Proyecto Fin de Carrera
Ingeniería Electrónica, Robótica y Mecatrónica

Simulación del robot KUKA YouBot en el entorno de CoppeliaSim

Autor: Francisco José Rivero González

Tutor: Manuel Vargas Villanueva

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Carrera
Ingeniería Robótica, Electrónica y Mecatrónica

Simulación del robot KUKA YouBot en el entorno de CoppeliaSim

Autor:

Francisco José Rivero González

Tutor:

Manuel Vargas Villanueva

Profesor titular

Dpto. de Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Carrera: Simulación del robot KUKA YouBot en el entorno de CoppeliaSim

Autor: Francisco José Rivero González

Tutor: Manuel Vargas Villanueva

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mi familia

A mis maestros

A mis amigos

A todos aquellos que han estado en los buenos momentos y, sobre todo, en los malos.

Agradecimientos

Una vez que empiezas a escribir esto te das cuenta de que una etapa termina después de duros años de esfuerzo y sacrificio. En estos momentos importantes es cuando miras atrás y a los lados para ver quienes siguen contigo y quienes ya no están. También es momento de agradecer por todos los que ayudaron en el camino que me ha llevado hasta aquí.

Por supuesto mi mayor agradecimiento es para mis padres que con su enorme esfuerzo y cariño me han apoyado en todo. No hay palabras para describir mi gratitud hacía ellos, sin ellos yo no estaría ahora mismo redactando estas palabras. Siempre han sido un pilar en los momentos duros y difíciles que han surgido a lo largo de estos años de universidad. Nunca han faltado sus consejos y advertencias, pero siempre dándome mi libertad para madurar y tomar decisiones. Tampoco su motivación a seguir luchando cuando surgían esos momentos de frustración, tristeza o falta de fuerzas. Por todo, infinitamente gracias.

En segundo lugar, no puedo olvidar a mis abuelos y mi familia. Siempre han estado para lo bueno y lo malo y estoy seguro de que se sienten muy orgullosos de lo que he conseguido y de la persona que soy, al fin y al cabo, ser una buena persona es el principal objetivo. Mis abuelos me han ayudado desde pequeño y tengo la suerte de aún tenerlos a mi lado. Nunca olvidaré los *"tuppers"* de comida casera que me ayudaron durante mi primer año fuera de casa o en las semanas difíciles de exámenes, que me llenaban de fuerza para seguir.

Por último, que no menos especial, debo agradecer la suerte que he tenido de conocer a maravillosas personas que hoy en día son amigos de corazón. Esa pequeña familia, como nosotros decimos, que estábamos juntos cuando sufríamos las jornadas intensas de estudios, las alegrías de los aprobados y por supuesto, las penas de los suspensos. Como suelo decir, los amigos son la familia que uno elige y yo no puedo estar más agradecido de los míos. En especial quiero agradecer el apoyo a Kevin, Javi, David y Dani "El canario". Con algunos de ellos he tenido y tengo el placer de vivir juntos, además de compartir innumerables experiencias. Todos ellos y los que no están nombrados siempre tendrán un hueco en mi corazón.

Gracias a todos los que han aportado algo durante estos largos años y me han regalado enseñanzas que siempre llevaré conmigo, ya sean académicas o personales.

Francisco José Rivero González

Sevilla, 2021

Resumen

Si juzgas a un pez por su habilidad para trepar árboles, pensará toda la vida que es un inútil.

Albert Einstein

Este proyecto se centra en controlar el robot manipular *KUKA YouBot* en el entorno de simulación *CoppeliaSim*. El software de *CoppeliaSim*, antes conocido como *V-REP*, es un potente simulador de robótico cuyo uso es gratuito y completo en su versión educativa. Durante este proyecto se conocerán algunas de sus funcionalidades y características que pueden sentar las bases de otros desarrollos. Para el caso de este proyecto, se ha elegido el robot *YouBot* perteneciente a los laboratorios *KUKA*. Es un modelo desarrollado e implementado en el software por el propio fabricante. Este robot es de tipo manipulador móvil, es decir, presenta una base móvil con un brazo robótico. Si se entra más en detalle, el brazo presenta 5 grados de libertad y su efector final es una pinza. Por otro lado, la base móvil está compuesta por cuatro ruedas omnidireccionales del tipo *Mecanum*, esto permite un mayor rango de movimiento y versatilidad. Además, se han incorporado dos sensores láser a la base de este para aumentar sus posibilidades de interacción con el entorno. A la hora de programar, se han usado los *scripts* embebidos y han sido programados en lenguaje *Lua*. Este lenguaje es fácilmente interpretado *CoppeliaSim* y presenta una extensa gama de funciones que pueden ser consultadas en el manual de referencia de este software.

Abstract

This project focuses on controlling the *KUKA YouBo*t manipulator robot in the robot simulator *CoppeliaSim*. The *CoppeliaSim* software, before known as *V-REP*, is a powerful robotics simulator that is free to use and complete in its educational version. During this project we will learn about some of its functionalities and features that can be used as the foundations for other developments. In the case of this project, the *YouBot* robot belonging to the *KUKA* laboratories has been chosen. It is a model developed and implemented in the software by the manufacturer itself. This robot is a mobile manipulator type, which means it has a mobile base with a robotic arm. If we go into more detail, the arm has 5 degrees of freedom and its end-effector is a gripper. On the other hand, the mobile base is composed of four *Mecanum* omni wheels, which allows for a greater range of movement and versatility. In addition, two laser sensors have been incorporated into the base to increase the possibilities of interaction with the environment. When it comes to programming, embedded scripts have been used and have been programmed in *Lua* language. This language is easily interpreted by *CoppeliaSim* and presents a wide range of functions that can be consulted in the reference manual of this software.

Índice

ÍNDICE DE FIGURAS

Notación

CAD	Diseño asistido por ordenador
DLS	*Damped least squares*
tan	Función tangente
sin	Función seno
cos	Función coseno
IK	*Inverse Kinematic* (Cinemática Inversa)

1 INTRODUCCIÓN

Los países con la mayor densidad de robots tienen también las tasas de desempleo más bajas. La combinación correcta de tecnología y humanos impulsarán la prosperidad.

Ulrich Spiesshofer, presidente y CEO de ABB

Hoy en día cada vez es más frecuente ver robots en nuestro día a día y en nuestros trabajos. Los robots ya no son solo industriales y, por lo tanto, se pueden ver en un restaurante, bar o cafetería sirviendo bebidas en la barra o en la entrada de un museo para interactuar con los visitantes a modo de recepcionista.

Este trabajo pretende mostrar la simulación de un robot real, llamado *KUKA youBot*, y que se encuentra en el mercado, a través del entorno de simulación de *CoppeliaSim*. Este software es gratuito y cuenta con una versión dedicada a la educación pensada para el uso de profesores, estudiantes, escuelas y universidades.

1.1. Motivación

Personalmente he elegido este trabajo por varios motivos, uno de los principales era el reto de aprender a usar un entorno de simulación totalmente desconocido para mí como es *CoppeliaSim* y programarlo usando el lenguaje *Lua*. La curva de aprendizaje de este programa puede ser alta al principio debido a su alta complejidad y multitud de opciones. Por otro lado, también tenía la intención de poder simular y realizar un trabajo sobre un robot real, es decir, que exista en el mercado. Esto era algo que me llamaba la atención y me apetecía por poder ver cómo controlar un robot ya diseñado y fabricado. Además, durante mis prácticas en un entorno industrial que trabaja con elementos, descubrí que la logística (transporte, almacenaje, etc.) era una parte muy importante del proceso, por ello quería usar en este trabajo un robot manipulador móvil. Este tipo de robots combina una base móvil con un brazo robótico. Esta configuración permite desplazar al brazo robótico por distintas zonas aumentando su área de trabajo y versatilidad.

Después de documentarme sobre el software *CoppeliaSim* y sus posibilidades descubrí, en la versión utilizada, el modelo del robot *youBot.* Este es un robot manipulador móvil que la empresa de robots *KUKA* había añadido para su simulación en este entorno. Por otro lado, no encontré casi ningún proyecto que utilizara este robot en particular u otros manipuladores móviles. Esto despertó aún más mi curiosidad y ganas de trabajar sobre el control de este robot. Con este proyecto pretendo crear una base de conocimiento para que ayude tanto

a entender el software de *CoppeliaSim* como el uso de los robots manipuladores móviles, en concreto el robot *KUKA youBot* en este entorno. Todo lo que aquí se expone, puede desarrollar mejoras simuladas del robot que más tarde se puedan implementar en su versión real.

1.2. Estado del arte y objetivos

El estudio y manejo de robots se lleva realizando desde hace varias décadas y desde entonces se han ido usando nuevos materiales, lenguajes de programación y configuraciones. Al principio se creaban robots para realizar tareas repetitivas y costosas en entornos industriales. Por ejemplo, son muy conocidos los brazos robóticos en las líneas de montaje de coches.

Figura 1: Robots en la industria automovilística.

El brazo robótico es uno de los muchos robots que existen e intenta imitar el brazo humano, de hecho, siempre se habla de sus partes haciendo referencia al hombro, codo y muñeca dependiendo del número de articulaciones. Al igual que ocurre con un brazo humano, los brazos robóticos se mueven con articulaciones rotatorias y se pueden equipar con distintas herramientas dependiendo del trabajo que vayan a realizar. Algunos ejemplos pueden ser una pistola de pintura, un soldador o simplemente una pinza para coger objetos. A parte de esos brazos, hay otros que intentan ser más humanos, por ello se asemejan en apariencia y poseen dedos. Estos robots se suelen denominar humanoides y se diseñan para imitar los movimientos y la apariencia humana.

Por otro lado, se tienen los robots móviles que son aquellos capaces de desplazarse en un entorno de forma automática. Dentro de esta categoría se pueden distinguir entre los terrestres, marinos o acuáticos y aéreos. También existen robots híbridos que combinan algunas categorías anteriores y por supuesto, las configuraciones son muy variadas y a veces intentan imitar animales, tanto en aspecto como en movimiento. Centrándose en los robots móviles, se puede observar esto en robots con patas.

Figura 2: Spot, robot fabricado por Boston Dynamics.

Además, dentro de los robots móviles terrestres se encuentran aquellos que se desplazan con ayuda de ruedas. En el mercado se pueden encontrar multitud de configuraciones y ruedas distintas.

Últimamente, se está buscando combinar brazos robóticos con robots móviles para aumentar las capacidades y posibilidades de actuación. El objetivo de este trabajo consiste en programar el movimiento de un robot que combina una plataforma móvil con un brazo robótico. Dentro de los modelos disponibles en el entorno de simulación de *CoppeliaSim*, la opción elegida para el proyecto ha sido un robot llamado *KUKA youBot,* el cual es un robot manipulador móvil con ruedas omnidireccionales del tipo *Mecanum*. Se busca tanto programar el movimiento de la plataforma como el movimiento del brazo robótico. Para ello, se ha decidido que el robot siga una trayectoria en la que primero tomará un objeto de una mesa y posteriormente, la dejará en otra mesa antes de volver a su posición inicial. [1]

A parte, se han añadido dos sensores tipo rayo en sus laterales para que el robot se sitúe a una distancia predefinida de las mesas. Con esto, se quiere mostrar la posibilidad de incorporar sensores al robot, los cuales no vienen de fábrica, para su mejor interacción con el entorno de su alrededor.

Figura 3: KUKA YouBot.

Fuente: maxongroup

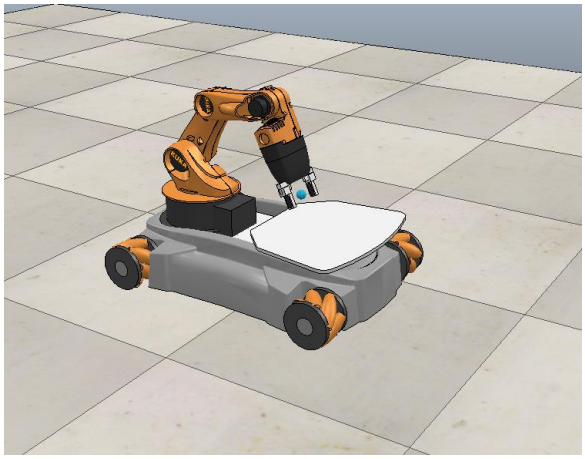

Figura 4: Modelo *KUKA youBot* en *CoppeliaSim.*

1.3. Estructura del trabajo

La realización de este proyecto se puede explicar siguiendo la cronología usada para su realización.

- Herramienta usada para la simulación: Conocer el entorno de *CoppeliaSim* y todas las funcionalidades necesarias. Esto implica crear objetos y programarlos a través del lenguaje de programación *Lua*.
- Movimiento del brazo robótico: En esta parte se comentará la morfología del brazo incorporado y la técnica empleada para lograr su movimiento. El objetivo de esto es lograr que sea capaz de recoger adecuadamente los objetos para su transporte.
- Movimiento del robot: Programar y desarrollar el código que permita el movimiento autónomo de la base móvil del robot y, por lo tanto, de cada una de sus ruedas omnidireccionales *Mecanum*. Se comentará en los próximos capítulos la morfología de las ruedas y su configuración, algo indispensable para el movimiento del robot.
- Mejoras y conclusión: La parte final de este proyecto ha sido la incorporación de los sensores laterales en el robot. Además, para concluir el proyecto se hablará de las conclusiones obtenidas a lo largo de todo el proceso y posibles mejoras.

2 ENTORNO DE SIMULACIÓN

Una de las partes fundamentales de este proyecto es conocer y manejar el entorno de simulación adecuadamente para conseguir los objetivos anteriormente expuestos en el apartado 1.1.

A continuación, se realizará una breve introducción de la herramienta de todas las funcionalidades que se han necesitado para realizar todo el trabajo. Se incluirán imágenes del proyecto y se explicará en detalle cómo se ha ido realizando la escena y todos los objetos añadidos.

2.1. Introducción a CoppeliaSim

El entorno de simulación donde se desarrolla este proyecto se llama *CoppeliaSim*, anteriormente era conocido como V-REP (*Virtual Robot Experimentation Platform*). El software es completamente gratuito y funcional en su versión educativa. Se encuentra disponible para cualquier sistema operativo (Windows, Ubuntu y Mac) en su página oficial [2]. Todas las simulaciones de este trabajo se han realizado usando la versión 4.0.0 EDU para Windows. Esta versión está destinada para usarla en un entorno académico y con ese propósito, sin embargo, no se encuentra limitada en ninguna de sus funcionalidades o capacidades. [3]

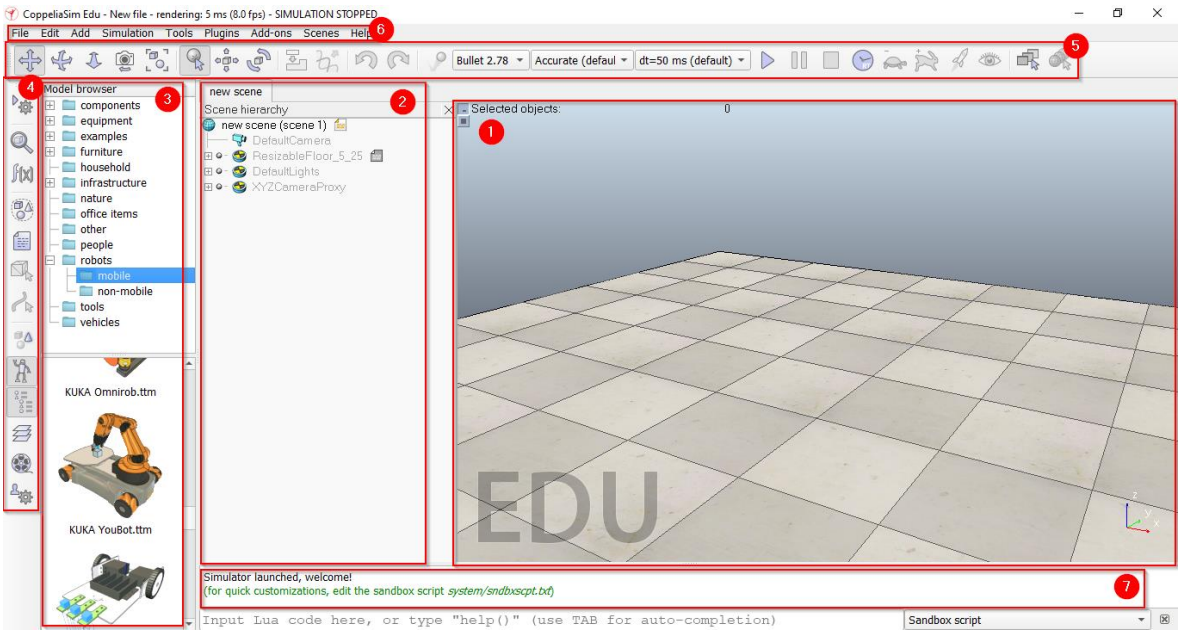

Figura 5: Ventana de inicio de *CoppeliSim*. 1: Escena, 2: Jerarquía de la escena, 3: Buscador de modelos, 4: Barra de herramientas 2, 5: Barra de herramientas 1, 6: Barra de menú, 7: Barra de estado.

Este simulador de robots no es el único que existe en el mercado, pero posee muchas funcionalidades y herramientas que hacen que sea muy versátil e ideal para simular muchas aplicaciones de robots. Toda la escena se puede personalizar con una gran variedad de elementos disponibles, además se pueden importar elementos realizados en cualquier programa CAD como puede ser *CATIA*. Esto permite diseños realistas según las necesidades de cada usuario.

Todos los objetos o modelos en la escena pueden ser programados y controlados individualmente con *scripts*, un plugin, *ROS* o un cliente API remoto. Con respecto a los lenguajes de programación, los controles pueden ser escritos en C/C++, *Python*, *Java*, *Matlab* o *Lua*. Este último ha sido el que se ha usado durante el proyecto. El lenguaje *Lua* es el que se suele usar con este software por su diseño para trabajar en scripts embebidos y presenta una extensa librería de funciones ya incorporadas. Estas librerías y su funcionalidad están completamente disponibles a través de la descarga del software o desde su página web [4]. Asimismo, todas las funciones, como se puede observar en el manual de referencia, están disponibles para su uso en C/C++.

Durante todo el proyecto se ha consultado este manual [5] ya que aclara el uso, además de los parámetros de entrada y salida de las funciones. Al principio este lenguaje puede resultar complicado por la poca información disponible, pero se puede observar un aumento de su uso, gracias a la comunidad que está surgiendo y colabora en el propio foro de la plataforma.

2.2. Escena y jerarquía

La escena del proyecto se compone de varios elementos como son formas primitivas (*shapes*), sensores (*sensors*) y caminos (*path*). Esto sin tener en cuenta el modelo del robot youBot.

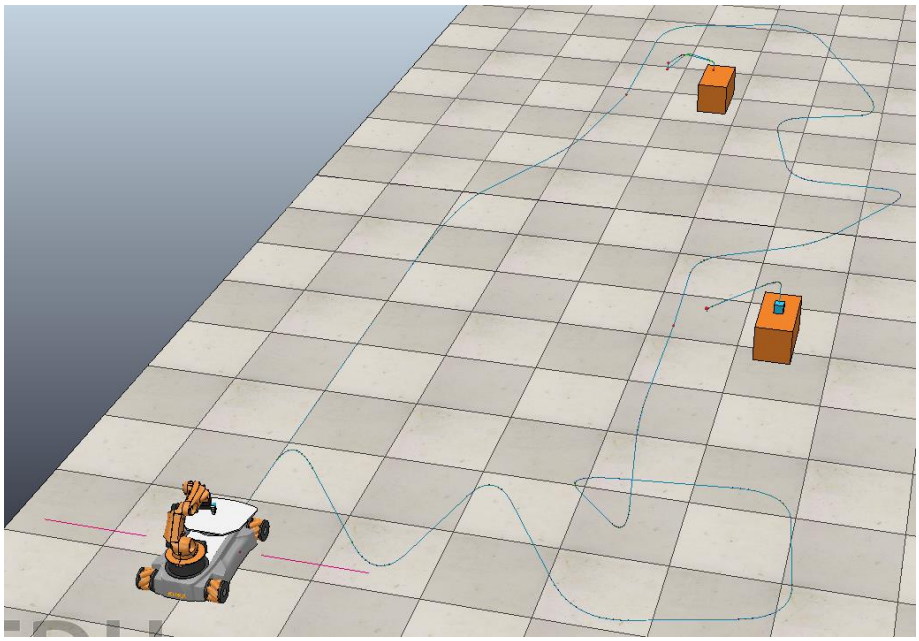

Figura 6: Escena del proyecto.

Lo primero que se ha modificado de la escena principal han sido sus dimensiones. Las dimensiones actuales son 5x10 metros.

El proceso de cambiar las dimensiones es muy sencillo. Como se muestra en la siguiente imagen solo es necesario hacer clic en el elemento que viene por defecto *ResizableFloor_5_25* y elegir las dimensiones en la ventana emergente.

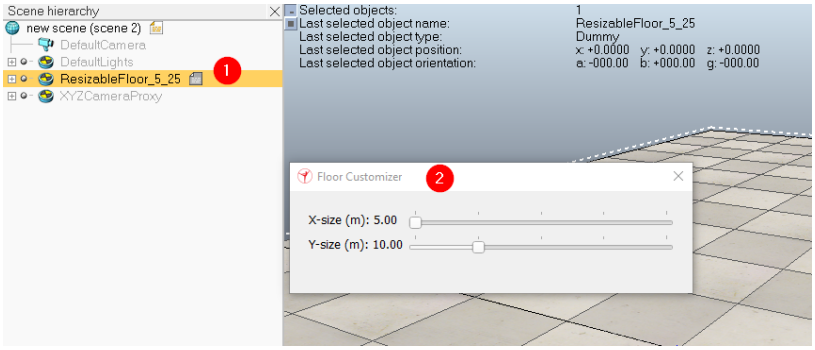

Figura 7: Vista de la customización del tamaño de escena.

Por otro lado, es importante destacar el papel de la jerarquía de objetos en la escena. Para *CoppeliaSim* es importante que estén definidos los padres e hijos de los elementos para su posterior simulación y propiedades. Como se puede observar en la siguiente imagen, la estructura tiene forma de árbol lo que permite ver la relación entre los diferentes objetos e incluso minimizarlos.

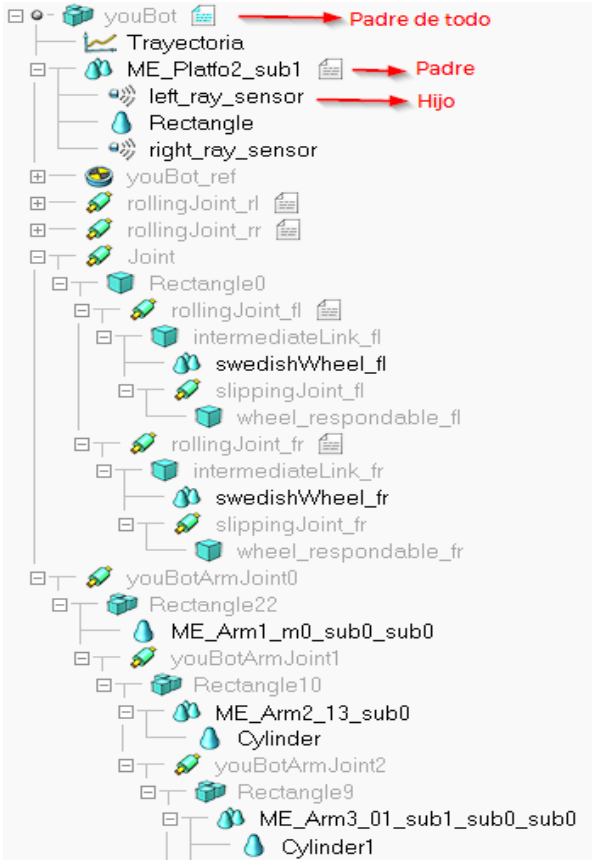

Figura 8: Vista de la jerarquía de escena.

Las jerarquías entre objetos deben estar bien definidas ya que son necesarias para realizar operaciones de traslación y rotación sobre las diferentes cadenas cinemáticas. Esto se observa a la hora de mover el robot, el movimiento se produce en las ruedas, pero se desplaza todo el robot en conjunto. El objeto padre puede acceder a las funcionalidades que ofrecen sus hijos, por ejemplo, llamadas a funciones de scripts de los hijos. En los brazos robóticos (como el usado en este proyecto) es necesario seguir una correcta jerarquía, para crear una cadena cinemática usando la estructura de eslabón – articulación. Esto es fundamental para usar los módulos de cálculo, como por ejemplo la cinemática inversa, que se aplican a los objetos con una relación jerárquica.

2.3. *Scripts* embebidos

Para la programación de los códigos de control de este proyecto, se han usado los distintos tipos de *scripts* embebidos que permite el software *CoppeliaSim*. El lenguaje de programación usado en estos scripts ha sido *Lua* [6]. Este lenguaje está basado en *C* y cuenta con una amplia librería de funciones. Mediante el manual de referencia del software, disponible en el software instalado o a través de su página web, se puede acceder a todas las funcionas y conocer su funcionamiento.

Los *scripts* [7] embebidos pertenecen a la escena donde están y son guardados con esta. En este proyecto se han usado los dos principales tipos: *main script* y *child script*. Al crear una nueva escena, se crea un *main script* que controla el bucle de la simulación principal y permite su funcionamiento. Este *script* no está pensado para que sea modificado y se desaconseja cualquier modificación. Un simple cambio puede provocar que los modelos no funcionen adecuadamente o se pierdan funcionalidades. Por otro lado, los *childs cripts* se usan para controlar los distintos elementos y el robot.

Todos los códigos de este proyecto se han implementado en varios *child scripts*. A su vez, existen dos tipos que son: *non-threaded child scripts* y *threaded child scripts*. Estos *scripts* se asocian a los distintos elementos de la escena y se puede acceder a ellos a través de la jerarquía de escena.

Los *non-threaded scripts* se representan con el símbolo en color negro, mientras que los *threaded child scripts* son de color azul. Ambos presentan distintas características que han sido necesarias para coordinar todos los elementos de control en la escena. Por ejemplo, los *threaded child scripts* se ejecutan más rápido y tienen una sincronización inherente con el bucle de simulación. Además, suelen ser asociados a actuadores o sensores por esta característica. A la hora de añadirlos, presentan distintas funciones principales como se observa en las imágenes siguiente.

```
function sysCall_init()
    -- do some initialization here
end

function sysCall_actuation()
    -- put your actuation code here
end

function sysCall_sensing()
    -- put your sensing code here
end

function sysCall_cleanup()
    -- do some clean-up here
end
```

```
function sysCall_threadmain()
    -- Put some initialization code here


    -- Put your main loop here, e.g.:
    --
    -- while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do
    --     local p=sim.getObjectPosition(objHandle,-1)
    --     p[1]=p[1]+0.001
    --     sim.setObjectPosition(objHandle,-1,p)
    --     sim.switchThread() -- resume in next simulation step
    -- end
end

function sysCall_cleanup()
    -- Put some clean-up code here
end
```

Figura 9: Vista de los distintos scripts; derecha: non-threaded child script, izquierda: threaded child script.

Para este proyecto se ha definido un *threaded child scripts* que maneja la simulación y las distintas tareas que se realizan. Dentro de este *script* se han definido varias funciones que reducen el código, mejorando su eficiencia y comprensión. Este código se ejecuta una vez y en comparación con el *non-threaded child script,* consume más recursos y puede perder algo de tiempo de procesamiento. Por ello, la implementación de ambos tipos de *scripts* en la simulación permite utilizar toda la potencia del software según las necesidades. Como se verá posteriormente, en la escena se han definido también varios *non-threaded child script* (algunos de ellos propios del modelo). Uno de estos scripts se usa para implementar el código de movimiento de la base del robot. El bucle de este *script* se ejecuta infinitamente mientras la simulación está activa y permite calcular las velocidades de las ruedas para que el robot se desplace adecuadamente. A esta funcionalidad se accede usando una cierta comunicación de rutina creada en el *threaded child script.*

2.4. Formas primitivas

El entorno de simulación permite crear elementos a partir de formas primitivas como puede ser un cubo o una esfera. Para este proyecto se han usado cubos que harán de mesas (color naranja) y uno pequeño azul que será transportado. A continuación, se muestra cómo crear estos objetos.

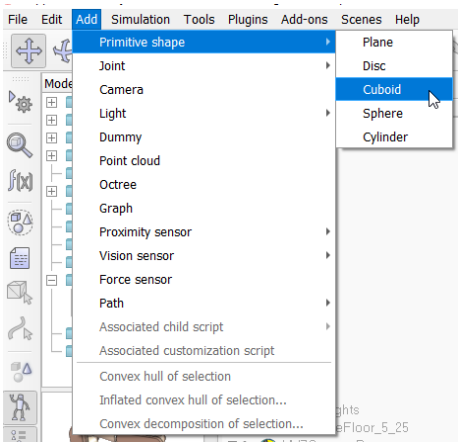

Figura 10: Selección de una nueva forma primitiva.

Los objetos creados, se pueden rotar y trasladar por la escena además se le pueden asociar scripts. Como se observa en la figura 11, se pueden personalizar en tamaño y color. Uno de los rasgos más importantes es definir su interacción con el entorno. Esto significa que se pueden modificar sus propiedades dinámicas y físicas. Los objetos pueden verse afectados por la física (gravedad), se pueden habilitar o deshabilitar las colisiones con otros objetos y decidir que el objeto pueda ser detectado por sensores, incluso que sensores. Esto permite una gran configuración y realismo en la simulación. Para acceder al cuadro de las propiedades solo es necesario hacer clic en la figura a través del recuadro de la jerarquía (figura 8). Al hacer esto, aparecerá una ventana emergente y seleccionamos el apartado *Common.* La figura siguiente muestra la ventana emergente. Para este proyecto es importante que la mesa sea detectable y dentro de las propiedades esté seleccionada la opción laser, ya que el sensor del robot es de este tipo.

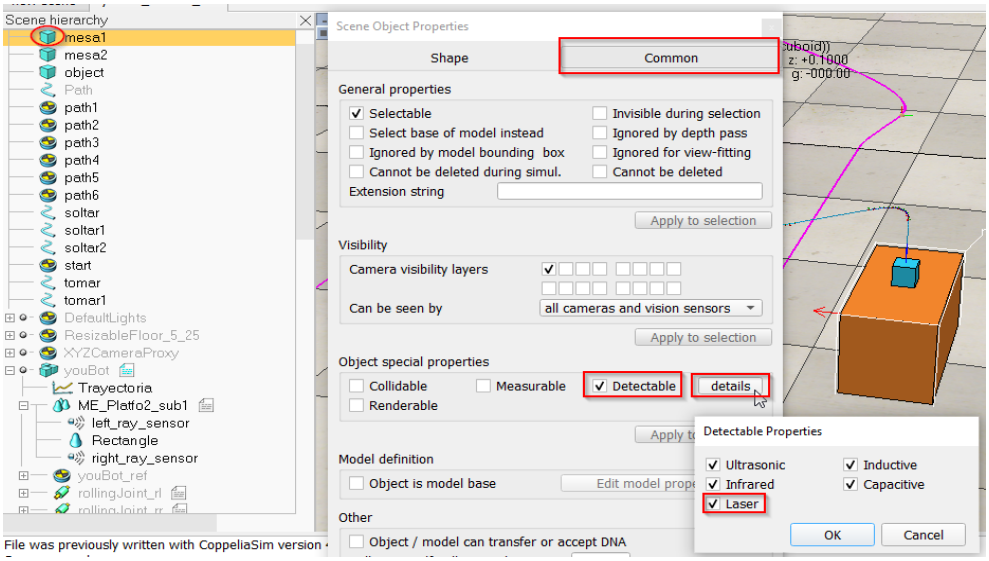

Figura 11: Vista de las propiedades del objeto.

Las propiedades especiales del objeto son:

- **Collidable**: El objeto puede ser comprobado para saber si está en colisión con otros objetos. Esto no implica que reaccione frente a una colisión, para ello el objeto debe ser *respondable*.
- **Detectable:** El objeto puede ser detectado por sensores de proximidad (ultrasonido, láser, etc.).
- **Renderable:** El objeto puede ser visto por sensores de visión (cámaras con capacidad de procesamiento).
- **Measurable:** El objeto puede ser usado para el cálculo de una distancia mínima.

2.5. Path

El camino o *path* en inglés es otro de los objetos que *CoppeliaSim* permite usar y cuyo uso se ha implementado en este proyecto. Este tipo de objeto se puede añadir a la escena a través de [*Menu bar* → *Add* → *Path*] y se representa como una sucesión de puntos con orientación en el espacio. El entorno de *CoppeliaSim* permite seleccionar entre dos tipos de caminos, circular y de tipo segmento. Todos los caminos usados en este proyecto son de tipo segmento. Una vez creado, se puede modificar y usar funcionalidades internas que permiten conocer la posición global de sus puntos entre otras, tal y como se verá a continuación. [8]

2.5.1. Trayectorias del brazo robótico

El movimiento del brazo que lleva incorporado el robot *youBot* se comentará en el capítulo siguiente. Ahora, se mencionará la trayectoria que sigue el brazo para coger el objeto de la mesa y dejarlo en su plataforma. Para este proceso es indispensable el objeto *path.* Se han creado dos caminos en los dos puntos donde el brazo se activa, el primer punto es donde se recoge al cubo que se transporta y el segundo punto es donde el cubo es depositado. El funcionamiento de los dos caminos es idéntico a la hora de diseñarlo, funcionalidad y propósito, por lo tanto, solo se mostrará la funcionalidad usando de ejemplo el camino creado en el primer punto.

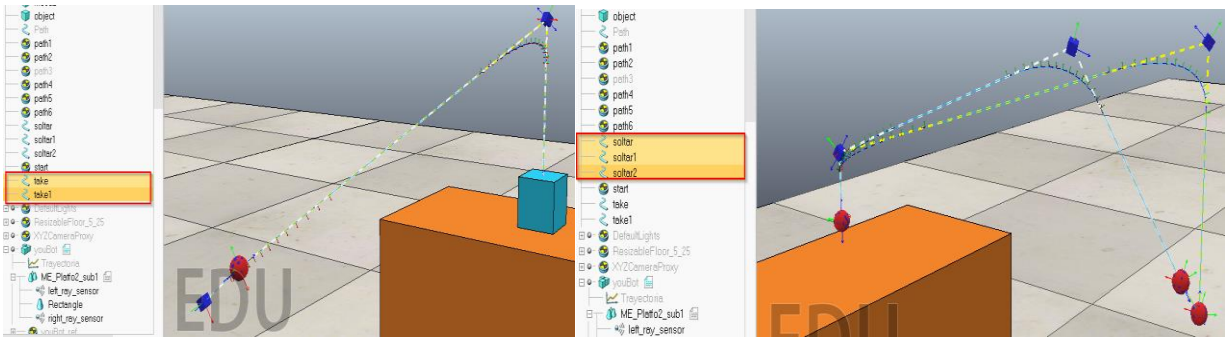

Figura 12: Primera trayectoria izquierda; segunda trayectoria derecha.

En el primer punto se ha divido la trayectoria total en dos caminos llamados *take* y *take1*. El primer tramo (*take*) comienza cuando llega el robot al punto y termino una vez que coge el cubo. El segundo tramo (*take1*) es la continuación del anterior y termina cuando se deja el cubo en la plataforma del robot.

Una vez que el robot llega al segundo punto se divide el proceso en tres tramos, estos son *soltar*, *soltar1* y *soltar2*. Lo primero en este caso es coger el cubo de la base móvil (*soltar*), dejarlo en la mesa (*soltar1*) y por último volver a su posición inicial (*soltar2*).

Para definir las propiedades del objeto *path* es necesario seleccionarlo después de crearse y hacer clic en el modo de edición de caminos. Este proceso se puede observar en la siguiente figura.

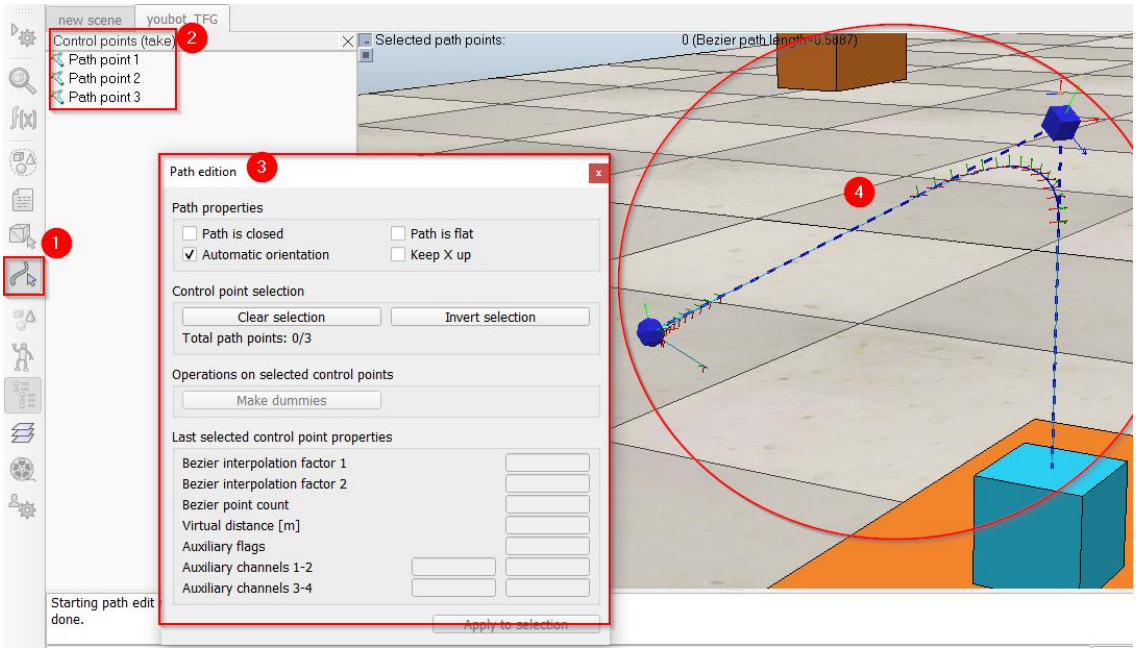

Figura 13: Vista de la edición del elemento path. 1: Modo edición de caminos, 2: Puntos de control, 3: Cuadro de edición, 4: puntos de control y camino representado en la escena.

Las propiedades elegidas para ambos caminos se pueden ver en la figura anterior. Como se puede observar ambos caminos están en el espacio por lo que tienen altura, esto permite movimientos en el espacio 3D. Lo importante del camino es definir correctamente los puntos de control y el orden de estos. Cuando se crea un camino aparecen dos puntos de control indicando posición inicial y final. Para añadir más puntos y poder crear diferentes formas y curvas, solo es necesario seleccionar un punto y usar copiar (Ctrl+C) y pegar (Ctrl+V). Los puntos seleccionados cambian de color en la escena para poder localizarlos mejor, luego se trasladan y rotan como un objeto de la escena, manteniendo en este caso el enlace con el punto anterior y posterior.

Hay que planificar y conocer la ruta que se quiere hacer y en qué sentido. Una de las condiciones que se necesitan para mover el brazo es seguir la trayectoria, antes descrita, siguiendo un orden. Los caminos se usan para describir la trayectoria de un punto durante la simulación. Gracias a unas funciones internas se puede determinar durante la simulación y modificar propiedades de ese punto como su posición y velocidad. El concepto usado para mover el brazo es el de seguir, mediante cinemática inversa, a ese punto mientras se desplaza por el camino. La trayectoria de un punto de referencia que se sitúa en la pinza del brazo será idéntica a la del punto en movimiento, siempre y cuando la configuración del brazo permita alcanzarla.

2.5.2 Trayectoria de la base móvil

La trayectoria de la base móvil y por lo tanto del robot en la simulación, sigue la misma mecánica que la del brazo. En este caso se compone de tres elementos *path* denominados *ruta1*, *ruta2* y *ruta3*. Al igual que para el brazo habrá un punto que recorra el *path* a una cierta velocidad. Ese punto se usará como referencia para los parámetros de entrada que necesita la función que mueve cada una de las ruedas del robot. Al ser un robot con

ruedas omnidireccionales del tipo *Mecanum*, el movimiento de cada rueda es independiente del resto. El código específico que permite conocer que velocidad angular hay que aplicar en cada momento a cada rueda, junto con las características de la base, se explican en detalle en el capítulo 4.

2.6. Sensores

Este apartado está dedicado a los sensores que CoppeliaSim permite integrar en la escena. Dentro de este programa existen tres grupos de sensores: visión, proximidad y fuerza. Aunque solo se ha implementado el sensor de proximidad, se hará una breve introducción de los sensores de visión.

2.6.1 Visión

Los sensores de visión pueden ser de tipo ortográfico (campo de visión rectangular) o perspectiva (campo de visión trapezoidal). Estos sensores simulan cámaras que pueden ser procesadas y obtienen imágenes de los objetos, siempre que entre sus propiedades este seleccionada esta opción. Las vistas de estos sensores pueden ser mostrada durante la simulación. El programa tiene un elemento "Cámara" que permite ver diferentes vistas de la escena, no hay que confundirlo con los sensores de visión.

2.6.2 Proximidad

Los sensores de proximidad simulan sensores del tipo infrarrojo, capacitivo, ultrasonido o láser, que es el elegido en este proyecto. Su función, además de detectar a los objetos que se encuentren en su volumen de detección, es medir la mínima distancia con ellos. Según el volumen se pueden distinguir seis tipos, tal y como aparecen en la siguiente figura.

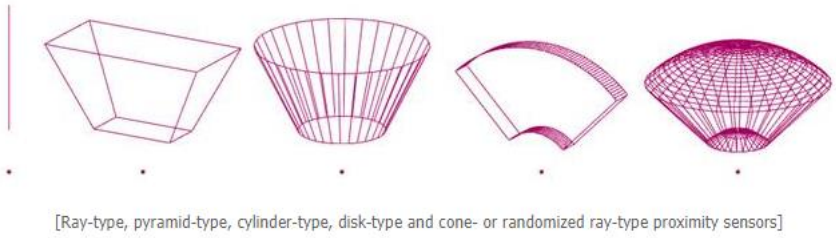

Figura 14: Diferentes tipos de sensores de proximidad.

Fuente: https://www.coppeliarobotics.com/helpFiles/en/proximitySensorDescription.htm

Todos los sensores tienen las mismas funcionalidades y propiedades. Lo que los diferencia es el volumen detectable que puede ser personalizable dentro de los diferentes tipos que existen. Cada uno de ellos pueden servir para simular diferentes sensores reales, por ejemplo, para un sensor de ultrasonidos se usaría un sensor tipo cono, y dentro del mismo, se podrían seleccionar su rango de detección, *offset* y su radio.

2.6.2.1 Tipo Haz

El sensor que se ha implementado en este proyecto es de tipo haz o rayo. Se ha colocado uno a cada lado del robot y simulan dos sensores láser. Estos sensores surgen como una mejora del movimiento de la base móvil. Con su uso se consigue que el robot detecte las mesas y se acerque o aleje una distancia dada previamente. El código asociado al sensor se explica en el capítulo 4.

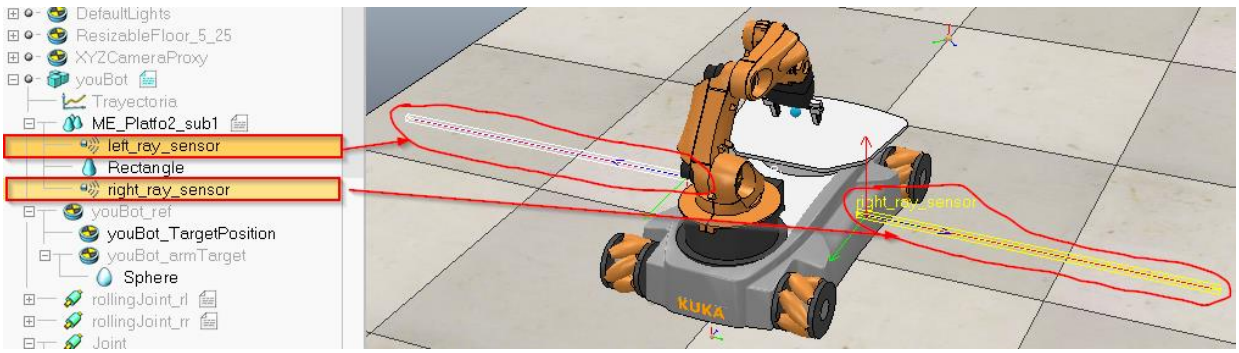

Figura 15: Sensores integrados en el robot *youBot.*

Este sensor se crea con [*Menu bar* → *Add* → *Proximity sensor* → *Ray type*]. Una vez creado se mueve con la opción de traslación hasta la posición deseada en el robot. Para que el sensor se mueva junto con el robot es importante definir la jerarquía de ambos. Como se observa en la figura anterior, los dos sensores son hijos de la plataforma que, a su vez, es hija del modelo completo del robot (se representa como *youBot* en la jerarquía).

Una vez realizado este proceso se han modificado ciertas propiedades. Haciendo doble clic sobre el sensor en la jerarquía de la escena se accede a sus propiedades. Hay que hacer clic en la figura pequeña, si se hace en el nombre, solo permite editarlo.

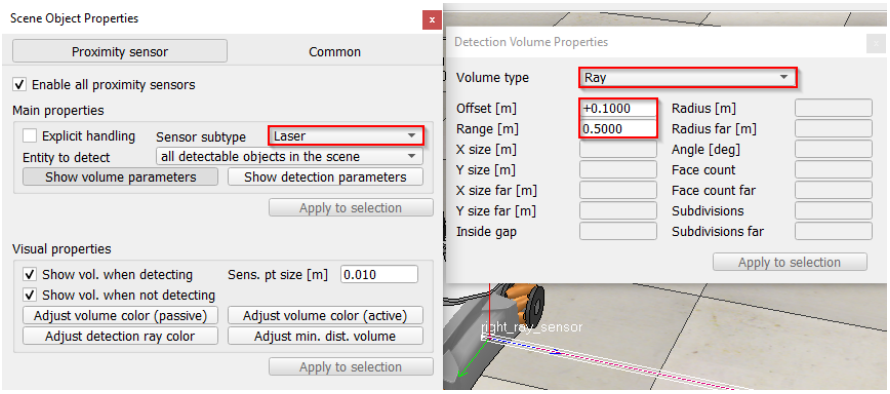

Figura 16: Propiedades de los sensores.

En la figura anterior, aparecen los dos cuadros de propiedades del sensor. Es importante que se encuentren marcados las opciones de *Laser* (*sensor subtype*) y *Ray* (*volumen type*).

3 BRAZO ROBÓTICO

Este capítulo trata sobre el brazo robótico y su control, una de las partes principales del robot *KUKA youBot*. A continuación, se detallan características del robot y se compara el robot real con su modelo en *CoppeliaSim*. Las técnicas de control explicadas para el funcionamiento de este brazo robótico pueden ser extrapoladas a otros brazos robóticos dentro del entorno de simulación.

3.1. Morfología

El brazo integrado en el robot *KUKA youBot* tiene un total de 5 grados de libertad. Se compone de 5 articulaciones de rotación y sus correspondientes eslabones en serie. Es una mecánica típica en un brazo robótico. El efector final de la muñeca es una pinza que permite coger objetos de cierto tamaño. Las pinzas, a diferencia del brazo, tienen articulaciones prismáticas que permiten su cierre y apertura. Los rangos de movimiento y las dimensiones del robot se pueden observar en las siguientes figuras.

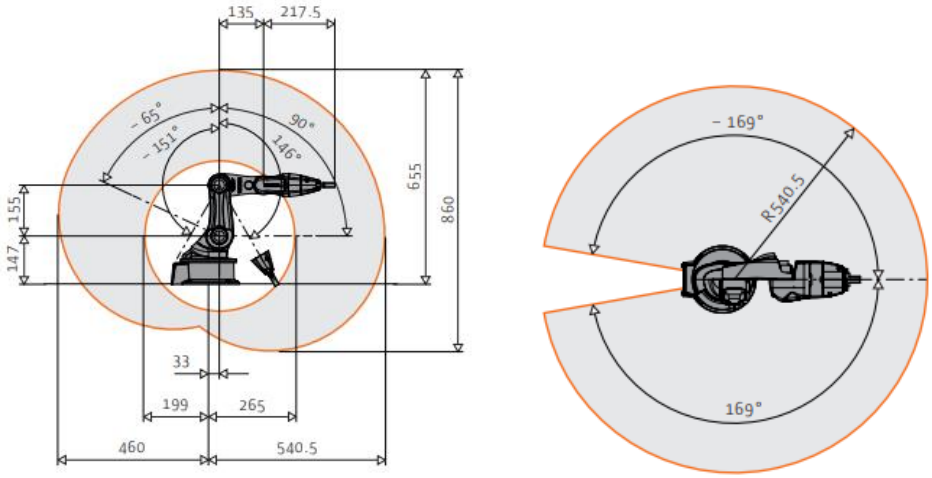

Figura 17: Planos del brazo robótica de KUKA YouBot

Fuente: https://www.generationrobots.com/img/Kuka-YouBot-Technical-Specs.pdf

Todas estas especificaciones son trasladas al modelo que se encuentra en *CoppeliaSim*. Para obtener el modelo del robot, es necesario seleccionarlo en [*Model browser* → *robots* → *mobile* → *KUKA YouBot.ttm*]. En la figura siguiente se muestran las distintas articulaciones sobre el modelo. El modelo ya incluye las restricciones de rango de movimiento.

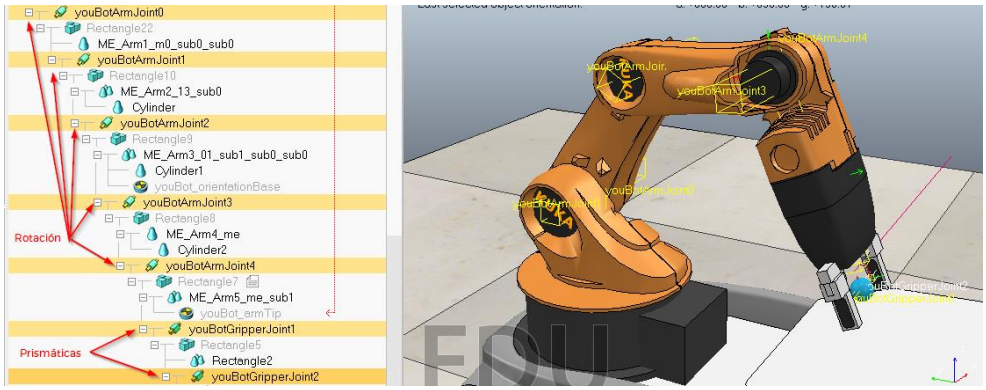

Figura 18: Modelo del brazo en CoppeliaSim

Para el estudio de los brazos robóticos y su modelo cinemático, tradicionalmente se ha recurrido al método de *Denavit-Hartenberg*. Este método permite establecer los sistemas de referencia de los eslabones que forman el brazo robótico. Sin embargo, para este trabajo se ha optado por usar el módulo de cinemática inversa que incluye el software de *CoppeliaSim*. De este modo, no es necesario definir el modelo cinemático ni los parámetros del brazo. Lo único necesario es definir correctamente una serie de propiedades de los objetos que forman el brazo para acceder a su control posteriormente. Tal y como se observa en la figura anterior, es fundamental crear correctamente la jerarquía para evitar fallos y definir correctamente la cadena cinemática.

3.2. Uso del módulo de cinemática inversa en CoppeliaSim

Antes de explicar cómo usar el módulo que integra *CoppeliaSim* de la cinemática inversa, es importante conocer el funcionamiento y lo que se pretende conseguir.

La cinemática inversa es un conjunto de ecuaciones que permiten conocer la posición del robot (cada una de sus articulaciones) para alcanzar un punto con su efector final [9]. Por este motivo, es tan importante definir la cadena del robot al que se le va a aplicar este módulo. La base y el efector final deben empezar y finalizar la jerarquía en forma de árbol. La tarea de resolver la cinemática inversa o IK (*Inverse Kinematic*) es definida por un grupo IK que contiene uno o varios elementos IK [10].

Para resolver la cinemática de una simple cadena es necesario definir un grupo IK con algún elemento IK. Este grupo se encarga de definir las propiedades generales. Cada elemento IK representa una cadena cinemática simple con algunos de sus eslabones definidos en modo cinemática inversa (*IK mode*) [11]. Básicamente estas cadenas están formadas inicialmente por una base, una serie de eslabones y una muñeca o efector final. Este

objeto debe estar enlazado con un objetivo. El elemento objetivo representa la posición y/u orientación que debe alcanzar el efector final tras resolver la cinemática inversa.

3.2.1 Configuración del modo IK

Lo primero es seleccionar los eslabones desde el 0 (*youBotArmJoint0*) hasta el 3 (*youBotArmJoint3*) y acceder a sus propiedades para elegir el modo de cinemática inversa. Se accede a las propiedades a través de la "lupa" que se encuentra a la izquierda. Este proceso se muestra en la siguiente imagen:

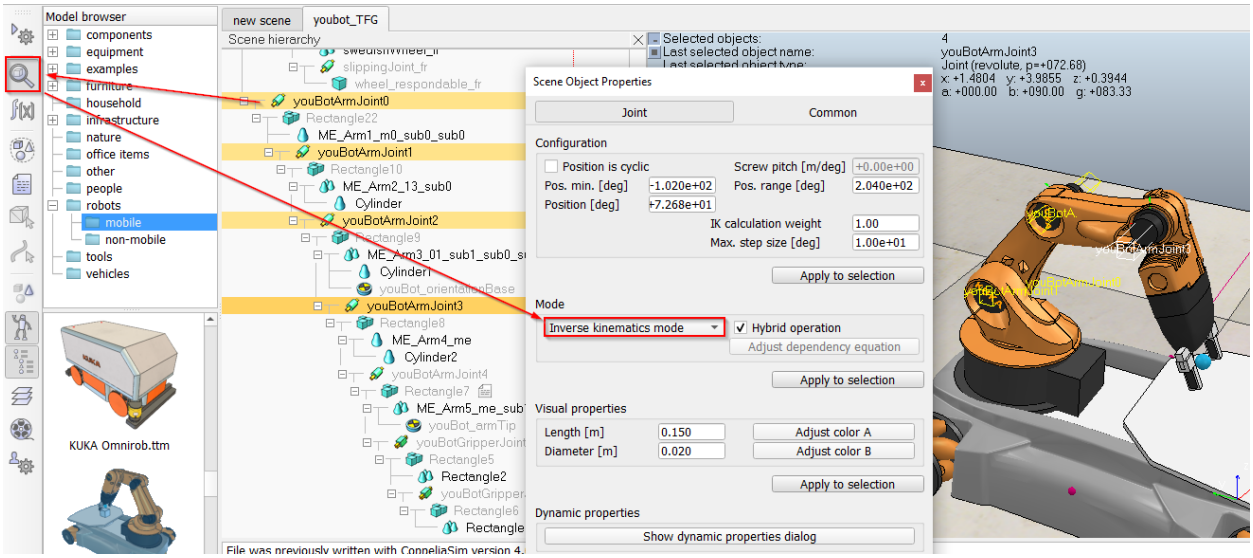

Figura 19: Selección del *IK mode* en los eslabones

Para el caso de los eslabones prismáticos de la pinza, se debe seleccionar el modo par/fuerza de la misma manera que se ha hecho en el caso anterior. Esto permitirá controlar la fuerza en lugar de la posición del eslabón, justo lo que se necesita para la apertura y cierre de la misma.

3.2.1.1 Enlazar la muñeca con el objetivo

Ahora se crearán dos elementos en la escena denominados "*Dummy*". La función de estos objetos es la de hacer de objetivo y posición final, respectivamente, de la cadena cinemática del brazo. Las propiedades de estos objetos permiten que puedan ser desplazados durante la simulación y, además, se puede acceder a su posición y orientación en todo momento. El objetivo es que un *dummy* sea perseguido por el otro. El *dummy* definido como posición final (será el punto que persiga), se coloca en medio de las pinzas y se representa visualmente como una esfera. Esto es así, para lograr un mejor agarre de los objetos a la hora de cogerlos y programar el movimiento.

Después de crear estos objetos de referencia y posicionar el de la muñeca, se procede a enlazarlos. Seleccionando el *dummy* de la muñeca (*youBot_armTip*) se hace clic en el botón izquierdo con forma de lupa.

En la ventana emergente se selecciona el otro objeto que queremos enlazar, en este caso se llama *youBot_armTarget*. Finalmente, para comprobar que se han enlazado correctamente debe aparecer una flecha roja bidireccional uniendo los dos elementos en el esquema árbol de la jerarquía. La siguiente imagen muestra el proceso y se puede observar el *dummy* entre la pinza, representado como una esfera. [12]

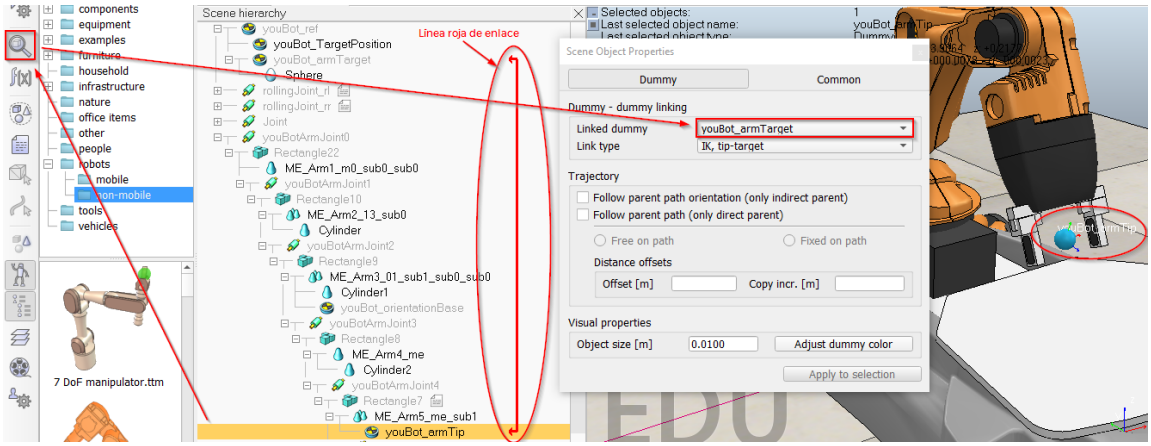

Figura 20: Vista del proceso para enlazar el efector final y el objetivo

3.2.1.2 Añadir el grupo IK y el elemento IK

Al principio, activamos todas las articulaciones con el modo IK. Ahora tenemos que lograr que se muevan a la vez para lograr la posición correcta. Para ello hay que crear un grupo IK usando el botón “f(x)” en la parte izquierda, como se muestra en la figura siguiente:

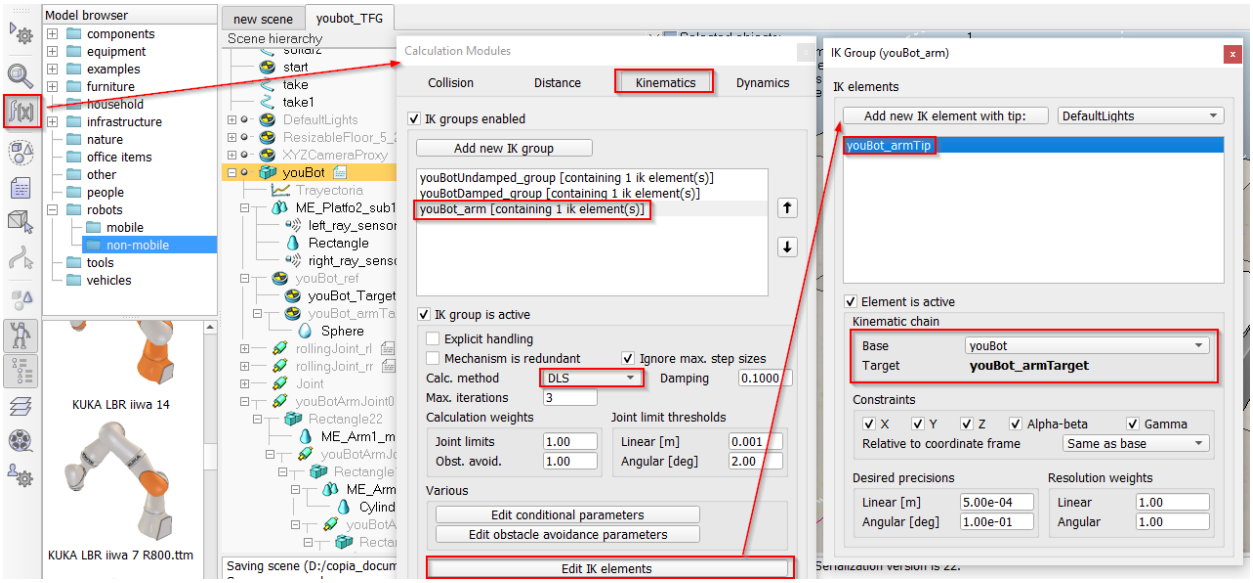

Figura 21: Vista de las pantallas para añadir el grupo IK

Una vez se despliega la ventana se selecciona el modo de cinemática (*kinematics*) y se añade un nuevo grupo IK. El nombre se puede modificar una vez creado, en este caso se llama: *youBot_arm*. Dentro de los métodos disponibles se ha optado por el método de mínimos cuadrados amortiguados o DLS por sus siglas en inglés. Este método es el más utilizado y presenta como ventaja, que se puede usar cuando no es posible alcanzar un objetivo (fuera del alcance o cerca de una configuración singular). La amortiguación puede dar lugar a cálculos más estables, pero hay que tener en cuenta que la amortiguación siempre ralentizará los cálculos (se necesitarán más iteraciones para llevar la punta a su sitio). Posteriormente, se pulsa sobre el botón para editar elementos IK y se añade un nuevo elemento IK con punta (*tip*) y a la derecha se selecciona, en la lista despegable, el *dummy* asociado a la punta. En la cadena cinemática se selecciona como base el robot (youBot). El objetivo aparece por defecto con el elemento enlazado a la punta. Finalmente, la punta del brazo intentará alcanzar el punto objetivo una vez comience la simulación.

3.3. Control del brazo robótico

El control del brazo está programado usando los *scripts* asociados a la pinza (*Rectangle7*) y al robot (*youBot*). Como se comentó en el primer capítulo, se han usado varios tipos de *scripts* y el lenguaje utilizado ha sido *Lua*. Lo primero que se debe hacer en el código es declarar las variables [13]. Estas adquieren el valor del manejador de cada objeto que es un número que *CoppeliaSim* crea internamente cada vez que se añade un nuevo elemento a la escena. Una vez que las variables tienen almacenadas el manejador (número desconocido a priori), basta con usarlas en cualquier parte del código para controlar los distintos elementos en la escena. Aquí se muestra una parte del código inicial.

```
function sysCall_threadmain()

-- Initialization for YouBot
    -- Get YouBot Handle
    you_bot = sim.getObjectHandle('youBot')
    youBot_reference = sim.getObjectHandle('youBot_ref')
    youBot_target = sim.getObjectHandle('youBot_TargetPosition')
    arm_target = sim.getObjectHandle('youBot_armTarget')
    arm_tip = sim.getObjectHandle('youBot_armTip')
```

Este trozo de código es una parte de la inicialización de variables que se encuentra dentro del *threaded child script* asociado al elemento "*youBot*". Las funciones definidas por el usuario se desarrollan antes del bucle principal, el cual empieza con la función *sysCall_theadmain()* y termina en un *end*. Las variables se asocian al manejador usando la función *sim.getObjectHandle('')* con el nombre del objeto dentro de las comillas simples. Este nombre será el definido en la jerarquía de escena y puede ser modificado. Si se quiere ver el manejador de los objetos y comprobar su número, tan solo es necesario escribir *print (nombre_de_la_variable)* y se vería en la barra de estado al simular la escena. En el código anterior se muestran solo, a modo de ejemplo, algunas de las variables definidas que se corresponden con el manejador de un objeto en la escena.

La posición inicial del brazo se puede establecer definiendo el ángulo de cada articulación. Esta asignación se puede realizar de forma manual en el código si se quiere alguna postura en concreto. Sin embargo, para este caso en el que se tiene el módulo IK implementado, se ha colocado el elemento "*target"* en la posición donde se quiere que empiece la pinza. A la hora de definir los ángulos, *CoppeliaSim* siempre necesita que la unidad sea radianes. Para hacer la conversión directamente en el programa, se puede usar la función matemática *math.pi,* que es igual que operar con el número π y dividir entre 180.

3.3.1 Movimiento coger/soltar pieza

Ahora se procede a controlar el movimiento del brazo en los puntos de la escena deseados. Como el proceso de coger y soltar el objeto se programa siguiendo la misma idea y funciones se va a explicar el movimiento de soltar. Ambos estarán disponibles para consulta en el anexo de códigos final. Todas las funciones que empiezan por "*sim"* son propias de la librería que viene por defecto en el software. Estas funciones pueden ser consultadas en el manual de referencia y al escribirlas en el *script* van apareciendo las distintas opciones. Basta con usar la tecla del tabulador para que se autocomplete la función que seleccionamos en el listado. Aquellas funciones que no empiezan por *sim* son definidas por el usuario al principio, en el código anterior se corresponden con *openGripper()* y *closeGripper()* que serán explicadas en el siguiente punto. A continuación, se explicarán en detalle cada función del código, pero antes se hará una breve explicación de la tarea realizada.

```
openGripper()
sim.setObjectParent(arm_target,-1,true)
sim.wait(2)
sim.followPath(arm_target, soltar, 3, 0, 0.1, 0.05)
closeGripper()
sim.followPath(arm_target, soltar1, 3, 0, 0.1, 0.05)
openGripper()
sim.followPath(arm_target, soltar2, 3, 0, 0.1, 0.05)
sim.setObjectParent(arm_target,youBot_reference,true)
sim.wait(1)
```

Empezando por arriba el brazo abre la pinza, el objeto "*target"* deja de tener padre y se espera 2 segundos antes de recorrer el primer camino definido con ciertas especificaciones. Se cierra la pinza y se vuelve a recorrer otro recorrido hasta el final y se abre la pinza. Finalmente se recorre el último trayecto hasta la posición inicial y se espera 1 segundo.

Con esta breve explicación se entenderá mejor las funcionalidades de las distintas funciones.

sim.setObjectParent (arm_target, -1, true): La variable *arm_target* se corresponde al punto *target* que mueve la posición del brazo al ser perseguido. Esta función asocia un elemento usando su manejador a un objeto padre en la escena. Sin embargo, en este caso, la variable *arm_target* se queda sin padre (-1) o libre. El elemento

booleano *true* indica que la posición y orientación absolutas del objeto deben permanecer iguales. Para el caso último, *sim.setObjectParent (arm_target, youBot_reference, true),* el punto objetivo se vuelve hijo de la referencia del robot para que cuando el robot se mueva, este punto siga manteniendo la distancia relativa con el brazo y se mueva durante la simulación. Si no se hace esto el punto *target* se quedaría quieto y al mover el robot con la base móvil, el brazo intentaría alcanzar el punto, pero ya está fuera de su alcance. No hay que olvidar que el enlace creado entre la muñeca y el punto *target* se mantiene durante toda la simulación, por lo tanto, la muñeca siempre va a estar siguiendo en todo momento al *target*.

sim.wait (2): Espera una cierta cantidad de tiempo durante la simulación. El tiempo está definido en segundos.

sim.followPath (number objectHandle, number pathHandle, number positionAndOrientation, number relativeDistanceOnPath, number velocity, number accel): Esta función mueve un objeto a través de un objeto *path.* El tercer argumento de entrada es un valor entre 1 y 3 (1: solo se modifica la posición, 2: solo se modifica la orientación, 3: se modifica la posición y la orientación). El cuarto argumento es un valor entre 0 y 1, donde 0 es el inicio del camino y 1 el final. Los últimos dos argumentos, se refieren a la velocidad del objeto durante su recorrido y su aceleración.

3.3.2 Apertura/cierre de la pinza

Ahora se van a explicar las dos funciones anteriores que permiten cerrar y abrir la pinza. Estas funciones son definidas al principio del script antes del *function sysCall_theadmain().* Para llamar a la función en el código solo es necesario indicar el nombre de la función y añadir dos paréntesis sin ningún argumento de entrada. Para usar estas funciones es necesario tener un *script* del tipo *non-threaded child script* que viene implementado con el modelo del brazo. Este *script* está asociado al objeto *Rectangle7* que corresponde con la pinza. Lo primero que se encuentra es la parte de inicialización. Para poder mandar señales entre los *scripts* se crea un canal de comunicación dentro de CoppeliaSim usando la función *sim.tubeOpen()*. Un canal es similar a una vía de comunicación bidireccional. Los mensajes escritos en un lado pueden ser leídos en el otro lado en el mismo orden en que fueron escritos.

```
openGripper=function()
    sim.tubeWrite(gripperCommunicationTube,sim.packInt32Table({1}))
    sim.wait(0.8)
end
```

```
closeGripper=function()
    sim.tubeWrite(gripperCommunicationTube,sim.packInt32Table({0}))
    sim.wait(0.8)
end
```

El código completo se muestra en el anexo final, pero se muestra a continuación la parte que sirve como actuador y a la cual se llama desde las funciones anteriores usando el canal de cominucación que se ha creado.

```
function sysCall_actuation()
    data=sim.tubeRead(communicationTube)
    if (data) then
        opening=sim.unpackInt32Table(data)[1]
    end

    if (opening==0) then
        sim.setJointTargetVelocity(j2,0.04) --closing
    else
        sim.setJointTargetVelocity(j2,-0.04) --opening
    end

    sim.setJointTargetPosition(j1,sim.getJointPosition(j2)*-0.5)
end
```

Las funciones de apertura/cierre escriben un 1 o 0 en el canal de comunicación. El *script* en la pinza siempre se está ejecutando y a mayor velocidad que el bucle principal, esta es una de las características de este tipo de *script*. Por lo tanto, una vez que detecta que se ha escrito algo por el canal usando la función *sim.tubeRead(),* extrae el paquete de datos y lo almacena en la variable *opening*. Como se sabe de antemano, este paquete de datos solo puede tener como valores posibles un 1 o un 0. Las variables *"j1"* y *"j2"* se corresponden con las articulaciones prismáticas que accionan la pinza. El movimiento de una articulación prismática es un movimiento relativo de traslación rectilínea a lo largo del eje de la articulación. Se ha definido que se cierre la pinza si el valor leído es 0. En este momento, se mueve la articulación hasta su otra posición a una cierta velocidad. La otra articulación repetirá el movimiento, pero hacia el lado opuesto. Para realizar le movimiento de la otra articulación se usa la función *sim.setJointTargetPosition(j1,sim.getJointPosition(j2)*-0.5).* Esta función fija el objetivo de una articulación si está en modo par/fuerza. El multiplicador "-0.5" no altera la velocidad a la que se mueve, sino hace que se muevan las dos a la vez. Esto es así por diseño en el programa ya que el rango de posicionamiento de la articulación *j2* es el doble que la de *j1,* y el signo negativo porque se mueven en direcciones opuestas.

4 PLATAFORMA MÓVIL

En este capítulo se estudiarán las características de la base móvil del robot *KUKA YouBot* y el control que se ha desarrollado para su movimiento en el entorno de simulación *CoppeliaSim*. La base móvil de este robot está compuesta por cuatro ruedas omnidireccionales y presenta una pequeña plataforma donde poder trasportar objetos. En las siguientes páginas, se mostrará el modelo de esta configuración. Además, se mostrará y explicará el código implementado para mover el robot por la simulación de forma autónoma. Por último, se incluye la implementación de los sensores en el robot y cómo se usan en este proyecto.

4.1 Características

La base de este robot está diseñada para poder transportar ciertos objetos usando su plataforma. El movimiento de esta base se produce gracias a su configuración de cuatro ruedas *Mecanum* que son del tipo omnidireccional. Este tipo de ruedas permiten alcanzar cualquier posición mediante una línea recta.

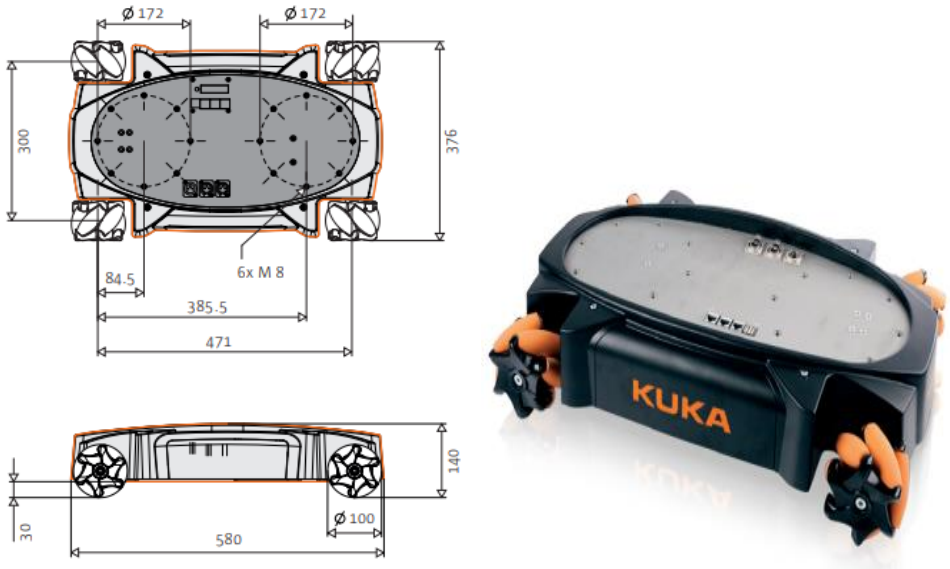

Figura 22: Planos e imagen de la base móvil

Fuente: Kuka-YouBot-Technical-Specs.pdf

Las ruedas *Mecanum* están formadas por una serie de rodillos montados con un ángulo de 45º [14]. Además, se mueven independientemente unas de otras, de forma que el robot no solo puede desplazarse hacia delante y hacia atrás, sino que también puede moverse de forma lateral, en diagonal o rotar sobre sí mismo.

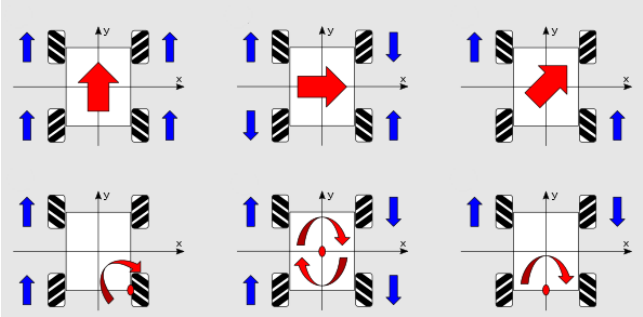

Figura 23: Configuración de accionamiento de las ruedas Mecanum para el movimiento del robot.

Fuente: Mecanum_wheel

4.2 Modelo cinemático inverso de la base móvil

Una vez que se añade el modelo a la escena, este tiene definidas las cuatro ruedas y cuáles son las delanteras y traseras. Esto se debe tener en cuenta para realizar el esquema que ayudará a obtener el modelo cinemático inverso.

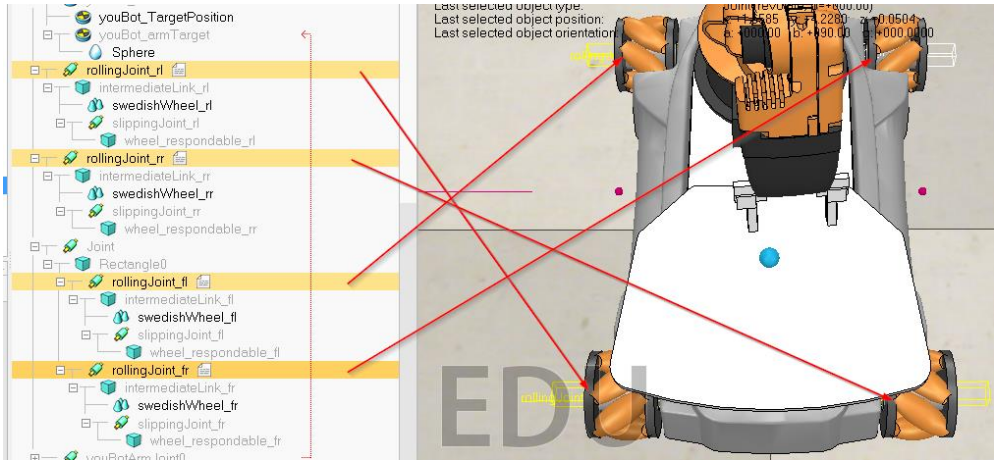

Figura 24: Ruedas en el modelo de CoppeliaSim

Según la configuración del robot, los rodillos de la rueda de la esquina superior derecha tienen la misma orientación que los de la rueda inferior izquierda, lo mismo ocurre para el caso de las otras dos, pero en sentido contrario. Cómo se observa en la figura siguiente que hace referencia al modelo del robot, las ruedas 1 y 3 son iguales y lo mismo ocurre entre las ruedas 2 y 4. Se calcula el movimiento en función al punto central del robot. Debido a la mecánica de la rueda *Mecanum*, cuando giran hacia adelante o hacia atrás también se

produce un movimiento lateral. Según el esquema, al girar las cuatro ruedas hacia delante las fuerzas o velocidades laterales se contrarrestan evitando el movimiento lateral. Esto es importante para realizar el modelo y para entender el porqué de esta distribución de las ruedas. Si quisiéramos un desplazamiento, por ejemplo, hacia la derecha, las ruedas 1 y 3 rotarían hacia atrás y las ruedas 2 y 4 hacia delante.

En este caso, se quiere obtener el modelo cinemático inverso de las ruedas. Esto permite obtener las velocidades angulares de cada rueda en función del movimiento del punto central [15].

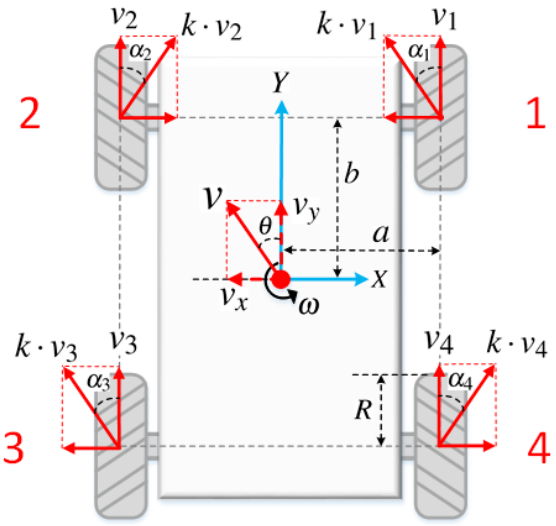

Figura 25: Modelo cinemático

Siguiendo como modelo el esquema anterior se obtiene que el movimiento del punto central se puede descomponer en una velocidad en el eje *X*, otra en el eje *Y* y una velocidad angular de rotación.

$$(v_x, v_y, \omega) \qquad (1)$$

donde se tiene la relación:

$$\begin{aligned} v_x &= v \sin\theta \\ v_y &= v \cos\theta \end{aligned} \qquad (2)$$

Por otro lado, al mover una rueda hacia adelante se sabe que también se produce un desplazamiento proporcional hacia un lateral. El movimiento de la rueda se puede igualar al movimiento del punto central de esta manera se obtiene la siguiente relación:

$$\begin{aligned} v_i + k v_i \cos\alpha_i &= v_y + a_i\omega \\ k v_i \sin\alpha_i &= v_x + b_i\omega \end{aligned} \qquad (3)$$

donde *i* representa el número de la rueda y α_i es la inclinación de los rodillos, en este caso, $\alpha_i = 45º$:

$$i = \{1,2,3,4\}$$

Aplicando una pequeña transformación en la ecuación (3) se obtiene:

$$tan\,\alpha_i = \frac{kv_i\,sin\,\alpha_i}{kv_i\,cos\,\alpha_i} = \frac{v_x+b_i\omega}{-v_i+v_y+a_i\omega} \qquad (4)$$

Se vuelve a transformar:

$$v_i = v_y + a_i\omega - \frac{v_x+b_i\omega}{tan\,\alpha_i} \qquad (5)$$

Ahora se sustituyen los valores para cada rueda. Particularizando para el ángulo de orientación de los rodillos en este caso:

$$\begin{aligned} v_1 &= v_y - v_x + a\omega + b\omega \\ v_2 &= v_y + v_x - a\omega - b\omega \\ v_3 &= v_y - v_x - a\omega - b\omega \\ v_4 &= v_y + v_x + a\omega + b\omega \end{aligned} \qquad (6)$$

La velocidad de la rueda se puede suponer:

$$v_i = \omega_i R \qquad (7)$$

Finalmente se puede obtener las ecuaciones que permiten relacionar el movimiento de cada rueda con el movimiento del punto central.

$$\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\\ \omega_4\end{bmatrix} = \frac{1}{R}\begin{bmatrix}-1 & 1 & (a+b)\\ 1 & 1 & -(a+b)\\ -1 & 1 & -(a+b)\\ 1 & 1 & (a+b)\end{bmatrix}\begin{bmatrix}v_x\\ v_y\\ \omega\end{bmatrix} \qquad (8)$$

Conociendo las dimensiones de la base móvil (*a* y *b*) y el radio de la rueda (*R*) se consigue la relación entre el movimiento de cada rueda y el punto central.

A partir de estas ecuaciones se pueden obtener de forma clara los principales movimientos y como se ajustan a lo espero. Para un movimiento hacia adelante o atrás todas las ruedas tienen la misma velocidad y mismo sentido. Para un movimiento de rotación las ruedas en el mismo lado tienen la misma velocidad, pero sentidos opuestos a los del otro lado. Para un movimiento lateral las ruedas en esquinas opuestas tienen la misma velocidad, pero sentidos opuestos de giro con las otras [16].

4.3 Implementación del modelo en Lua

El siguiente paso es implementar las ecuaciones del modelo que se han deducido, en el entorno de simulación. El movimiento del robot y el modelo se incluyen dentro de un *non-threaded child script* asociado al objeto plataforma (*ME_Platfo2_sub1*).

```
-- Inverse kinematics
function movementOmniwheel(vx, vy, omega, wheel_R, a, b)
omega_1 = (vy - vx + (a+b)*omega)/wheel_R
omega_2 = (vy + vx - (a+b)*omega)/wheel_R
omega_3 = (vy - vx - (a+b)*omega)/wheel_R
omega_4 = (vy + vx + (a+b)*omega)/wheel_R

-- set the right direction for each wheel
v_wheel_1 = -omega_1
v_wheel_2 = -omega_2
v_wheel_3 = -omega_3
v_wheel_4 = -omega_4
end
```

Las distintas constan

```
-- youBot size parameters
wheel_R = 0.05 -- 0.05 m
a = 0.15
b = 0.235
```

La velocidad de las ruedas se asigna a la velocidad angular con el signo negativo debido al modelo de la simulación. Presentan esta particularidad en el software y es necesario para que se muevan correctamente las ruedas.

4.3.1 Obtención de la velocidad angular de las ruedas

El siguiente paso tras implementar la función para obtener la velocidad angular de cada rueda, es calcular los valores de las variables de entrada de la función. Además, de las constantes (radio de las ruedas y medidas de la plataforma) es necesario conocer la velocidad lineal en los ejes X e Y y el giro del punto central del robot en cada instante. Esto con respecto a un punto de referencia que se encarga de marcar la trayectoria y que se mueve con velocidad constante, definida antes de iniciar cada tramo.

Esta implementación se desarrolla dentro del mismo *script,* pero en la función predefinida de actuación. A las funciones que aparecen cuando se crea este tipo de *script* se denomina *callback function,* son cuatro y la única obligatoria es la de inicialización. En este caso, la función de actuación se ejecuta en cada paso de simulación, durante la fase de actuación y se encarga de manejar los diferentes actuadores de la simulación, como es el caso de las articulaciones que mueven las ruedas.

```
function sysCall_actuation()

        delta_pos = sim.getObjectPosition(youBot_target, you_bot)

        delta_ori = sim.getObjectOrientation(youBot_target, you_bot)

        center_velocity = {delta_pos[2], delta_pos[3], delta_ori[1]}

        movementOmniwheel(center_velocity[1], center_velocity[2],
center_velocity[3], wheel_R, a, b)

        -- Apply the desired wheel velocities
        sim.setJointTargetVelocity(wheel_joints[1], v_wheel_1)
        sim.setJointTargetVelocity(wheel_joints[2], v_wheel_2)
        sim.setJointTargetVelocity(wheel_joints[3], v_wheel_3)
        sim.setJointTargetVelocity(wheel_joints[4], v_wheel_4)

end
```

El punto que se mueve siguiendo la trayectoria de la base móvil está declarado en la variable *youBot_target.* El movimiento de este punto se programa en el código de la rutina principal. Este punto se mueve con una velocidad constante que va a ser diferente en cada trayectoria. Existen tres trayectorias que van desde el inicio hasta la mesa con el objeto, luego de esta mesa hasta la segunda mesa donde lo deposita y, por último, de esta segunda mesa hasta la posición inicial.

La variable *delta_pos* almacena la posición relativa del objeto *youBot_target* con el robot *you_bot.* Por otro lado, la variable *delta_ori* almacena la orientación relativa del objeto *youBot_target* con el robot *you_bot.* La primera variable almacena la posición (x, y, z) y la segunda la orientación en ángulos de *Euler* (*alpha* (a), *beta* (b), *gamma* (g)). Estas variables nos permiten conocer los parámetros de entrada de la función, que son las velocidades lineales y el giro del punto central.

Ahora es importante elegir bien las variables que necesitamos, ya que tenemos tres valores de posición y tres valores de orientación, pero solo necesitamos tres valores de entrada. Para las velocidades *Vx* y *Vy* se utilizan *delta_pos[2]* y *delta_pos[3]* respectivamente. Esto puede resultar extraño porque son las posiciones de los ejes Y e Z, pero tiene la siguiente explicación: cuando se crea el elemento *path* este se crea con un sistema de coordenadas en el que el eje X es vertical y el eje Z es el que sigue la trayectoria. A continuación, se muestra la vista del camino:

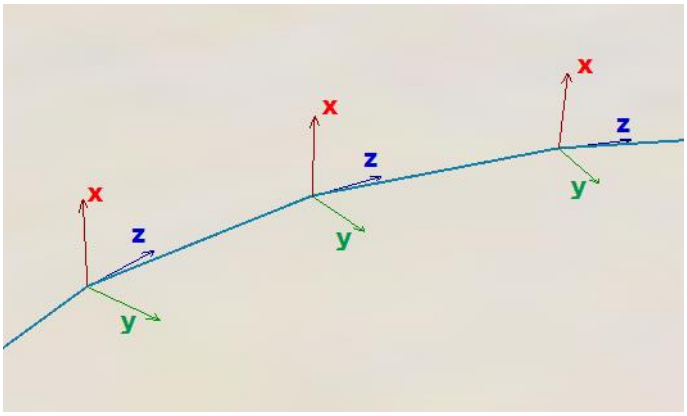

Figura 26: Vista de los ejes en la trayectoria

Para la variable *delta_ori* se toma su primer valor que hace a la rotación sobre el eje X.

Ahora es el momento de introducir estos valores en la función que calcula la velocidad angular de cada rueda y obtener, por tanto, los valores de *v_wheel_1, v_wheel_2, v_wheel_3* y *v_wheel_4.* Por último, se asocia cada valor a su rueda correspondiente usando la función *sim.setJointTargetVelocity(),* tal y como se observa en el código siguiente.

```
-- Apply the desired wheel velocities
sim.setJointTargetVelocity(wheel_joints[1], v_wheel_1)
sim.setJointTargetVelocity(wheel_joints[2], v_wheel_2)
sim.setJointTargetVelocity(wheel_joints[3], v_wheel_3)
sim.setJointTargetVelocity(wheel_joints[4], v_wheel_4)
```

Hay que tener en cuenta cada rueda del modelo de *CoppeliSim* a la hora de asociarlo a las definidas dentro de la función *movementOmniwheels(),* para no equivocarse con lo establecido en el esquema de la figura 25, que es de donde se han obtenido las ecuaciones. La asociación de las variables con las ruedas del robot en la simulación se realiza al principio del *script,* dentro de la función *sysCall_init()*. El código empleado para ello es el siguiente:

```
-- Prepare initial values for four wheels
wheel_joints = {-1,-1,-1,-1} -- front right, front left, rear left, rear right
wheel_joints[1] = sim.getObjectHandle('rollingJoint_fr')
wheel_joints[2] = sim.getObjectHandle('rollingJoint_fl')
wheel_joints[3] = sim.getObjectHandle('rollingJoint_rl')
wheel_joints[4] = sim.getObjectHandle('rollingJoint_rr')
```

Se observa que para evitar confusión se han asociado usando el esquema teórico, la rueda uno se asocia con la delantera derecha (*fr, front-right*), la dos con la delantera izquierda (*fl, front left*) y así hasta asociar las cuatro.

4.4 Control de las ruedas

El modelo posee unos *scripts* asociados a cada rueda cuando se crea el modelo en escena. Estos *scripts* son idénticos, la única diferencia son las variables a las que hacen referencia que son las propias de la rueda asociada. Su función es la de permitir el movimiento de las ruedas durante la simulación y, por consiguiente, el movimiento completo del robot.

Durante la realización de este proyecto surgieron varios errores al simular el robot. Tras un estudio del problema este se debía a un *bug* del software. Como solución se han implementado dos funciones que sustituyen a las funciones originarias en las ruedas (*sim.setObjectPosition()* y *sim.setObjectOrientation()*). Estos códigos se muestran a continuación y deben ser implementados si se usa este robot en una versión inferior a la 4.0.1. Como aclaración, este proyecto se ha realizado en la versión 4.0.0 pero ya existen versiones posteriores donde se ha corrige este error. Estas funciones han sido tomadas de esas versiones y la comunidad de *CoppeliaSim* como parche para esta versión.

```
function __setObjectPosition__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in
CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and
(sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectPosition(a,b,c)
end
function __setObjectOrientation__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in
CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and
(sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectOrientation(a,b,c)
end
```

4.5 Sensores asociados a la base

El modelo presente de *KUKA YouBot* en *CoppeliaSim* no viene con ningún sensor integrado. En la realidad, tampoco presenta ningún sensor. Por este motivo, se ha querido añadir como mejora dos sensores de tipo láser en la simulación [17]. Concretamente, los sensores se encuentran en la base móvil del robot y son hijos de esta. Aquí no se entrará en profundidad en explicar estos sensores ya que fueron explicados en el apartado 2.6.2.1 y se muestran en la figura 15.

Este apartado se centra en explicar el código que permite utilizar los sensores y usar su información para, en este caso, mover el robot. Todas las funciones y el código de los sensores se han programado en el *threaded child script* asociado al robot y que maneja la rutina que crea la secuencia de acciones del robot (movimiento y manipulación de la pieza).

Como siempre se empieza por inicializar las variables que hacen referencia a ambos sensores:

```
sensor_l = sim.getObjectHandle('left_ray_sensor')
sensor_r = sim.getObjectHandle('right_ray_sensor')
```

Los sensores están siempre leyendo información, pero solo la usaremos en los puntos donde están las mesas. Estos puntos coinciden con el final de la primera y segunda ruta. Una vez llegados a estos puntos, el robot se para y usa la información de los sensores activos. En la simulación se puede observar la activación de los sensores ya que al detectar un objeto el rayo pasa a parpadear en color amarillo.

Para saber el sensor que está activo (el entorno y la prueba creada solo usa un sensor en cada punto) se usa la función *sim.readProximitySensor()* que devuelve un 1 si el sensor detecta algo. Se muestra el código utilizado para saber si el sensor activo es el situado en el lado derecho o izquierdo.

```
if (sim.readProximitySensor(sensor_l)>0) then

   sensor_distance(sensor_l)

end
if (sim.readProximitySensor(sensor_r)>0) then

    sensor_distance(sensor_r)

end
```

Si el sensor se está activo se llama a una función definida para este proyecto que se llama *sensor_distance.* Esta función detecta a la distancia que se encuentra del punto y, aleja o acerca el robot hasta situarse a una cierta distancia predefinida. Además, se usa otra función creada para obtener la distancia que detecta el sensor, esta se denomina *getDistance()*. [18]

```
function getDistance(sensor)

   max_dist = 0.5 --0.5m distancia máxima asignada al sensor en la simulación
   local detected, distance
    detected,distance=sim.readProximitySensor(sensor)
    if (detected<0) then
        distance=max_dist
    end
    return  distance

end
```

Esta función devuelve la distancia en metros detectada por el sensor. Este dato se obtiene también de la función *sim.readProximitySensor()* que es propia del software de *CoppeliaSim*. Esta función es capaz de devolver varios datos, pero solo interesa, para este caso, saber si el sensor detecta algo (valor almacenado en la variable local *detected*) y la distancia a la que se encuentra el objeto detectado (*distance*).

Ahora se muestra la función *sensor_distance* en la cual se hace uso de la función anterior. Como se ve en el código, se usa la función *print()* que muestra las variables o texto por la barra de estado durante la simulación. Cada vez que se pone esta función se escribe una nueva línea. Luego se observa la distancia a la que se quiere que se sitúa el robot del objeto, para este proyecto se ha decidido que sea 0.1m. Para la simulación desarrollada, es importante que esta distancia sea adecuada para que el brazo robótico sea capaz de alcanzar el objeto de la mesa y luego, depositarlo en la otra.

```
sensor_distance=function(sensor)

        dist = getDistance(sensor)

        printf ("Distancia primera")
        print (dist)

        p_dist = dist - 0.1 --Distancia que se quiere conseguir 0.1

        p1=sim.getObjectPosition(youBot_target,-1)
        pfin = p1[1]-p_dist
        ptarget = {pfin, p1[2], p1[3]}
        sim.setObjectPosition(youBot_target, -1,  ptarget)
        waitReachTarget()
        dist = getDistance(sensor)
        printf ("Distancia final")
        print (dist)

end
```

Muchas de estas funciones han sido explicadas en capítulos anteriores, pero se hará una breve explicación del funcionamiento global de la función. Lo primero es calcular la distancia a la que está el robot del objeto para después, realizar el cálculo que permite saber cuánto se tiene que desplazar el robot para colocarse a la distancia establecida. Luego, este punto se asigna a la variable *youBot_target* para que el robot se mueva y siga a la nueva posición. En este caso, se ha implementado otra función para evitar cualquier problema en la transición entre movimiento de la plataforma y movimiento del brazo robótico, ya que se quiere mover el brazo solo cuando el robot este en su nueva posición. Esta función se ha denominado *waitReachTarget()* y su código es el siguiente:

```
waitReachTarget=function()
    repeat
        sim.switchThread()
        p1=sim.getObjectPosition(youBot_target,-1)
        p2=sim.getObjectPosition(youBot_reference,-1)
        p={p2[1]-p1[1],p2[2]-p1[2]}
        pError=math.sqrt(p[1]*p[1]+p[2]*p[2])
    until (pError<0.001)
end
```

Esta función permite que no se avance a la siguiente línea de ejecución hasta que el punto de referencia y el robot se encuentren en la misma posición con un error máximo de 0.001 m. En esta función, se puede observar el uso de funciones matemáticas en CoppeliaSim, como es el caso de la raíz cuadrada (*math.srqt*).

Una vez que se ha alcanzado el punto aparecerá también por la barra de estado (debajo de la escena) la nueva distancia que detecta el sensor (Distancia final). Esto permite comprobar que la distancia determinada ha sido alcanzada correctamente, tal y como se muestra en la imagen siguiente.

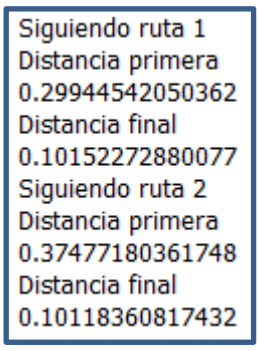

Figura 27: Captura de la barra de estado

Una vez ha sido alcanzado el punto, el robot se mantiene en la posición hasta que el brazo recoge o suelta la pieza en su correspondiente mesa.

4.6 Código de la rutina principal

Todos los códigos están disponibles al final de este documento para su consulta, sin embargo, se pretende explicar brevemente la secuencia que se sigue en el programa principal de la simulación.

El robot parte de una posición establecida y realizará un giro de 180º sobre sí mismo para cambiar su orientación. Esto también se realiza para mostrar una de las características de los robots con ruedas omnidireccionales *Mecanum,* la rotación sobre su propio eje. Una vez que ha realizado el giro, se empieza a mover siguiendo la primera ruta hasta la mesa 1. En este punto, se activa el sensor y se aproxima una cierta distancia para, posteriormente, recoger la pieza con la ayuda del brazo robótico. Tras depositar la pieza en la plataforma, el robot comienza a seguir la siguiente ruta que acaba en la mesa 2. Ahora, se repite la misma secuencia con la detección del sensor y la colocación de la pieza en su nueva posición. Una vez es dejada la pieza, el brazo robótico vuelve a su posición y comienza la trayectoria final hacia el punto de partida.

5 CONCLUSIONES Y FUTURAS MEJORAS

Durante el desarrollo de este proyecto han ido surgiendo complicaciones y muchas dudas. Una de las grandes dificultades ha sido el desconocimiento inicial de este software. Es cierto, que es un software abierto a muchas posibilidades de personalización y con una extensa capacidad de simulación, además de contar con muchas funcionalidades y herramientas. Todo esto también tiene el lado negativo de necesitar tener una buena base de conocimiento en su uso y en herramientas o lenguajes de programación externos. Por ejemplo, este software puede usarse con *ROS*, *Matlab* o *Python*. Esto además supone conocer cómo conectar ambos programas, como puede ser el caso de *Matlab-CoppeliaSim*.

En este proyecto se han fijado unas bases para conocer la herramienta y la programación de *LUA,* lenguaje que contiene una extensa librería de funciones. Una posible mejora de este proyecto podría ser usar una API externa e implementar funciones de control con Matlab.

Por otro lado, he tenido la dificultad añadida de usar el modelo de *KUKA YouBot*. Este robot manipulador móvil posee poca documentación y aún menos ejemplos de uso, debido a que los modelos se van incorporando poco a poco al software y no todos los usuarios lo usan. Por ello ha sido necesario hacer un trabajo para adaptar movimientos y funcionalidades de robots parecidos a este. Con este proyecto se buscaba dar a conocer el modelo y una base para su manejo, además se ha introducido una cierta mejora al integrar ciertos sensores. Por ello, pienso que futuras mejoras pueden consistir en implementar diferentes sensores y usarlos en escenarios de simulación distintos. También se puede mejorar el control del robot o modificarlo por otro en el que se apliquen otras técnicas.

La comunidad detrás de CoppeliaSim está creciendo y cada vez se verán más ejemplos de uso. También resulta de ayuda que su versión educativa sea gratuita y completamente funcional para que más estudiantes y profesores hagan sus simulaciones en esta plataforma. Por eso, aunque la curva de aprendizaje haya sido bastante pronunciada en el inicio, le veo mucho potencial a este software y su amplia variedad permite simular bastantes entornos, ya sean experimentales o reales dentro de una industria. También, otro paso importante que sería interesante de realizar es pasar del entorno de simulación al robot de verdad, para comprobar con seguridad que lo simulado se asemeja con la realidad.

ANEXO A - CÓDIGOS

Threaded child script (youBot)

```
openGripper=function()
    sim.tubeWrite(gripperCommunicationTube,sim.packInt32Table({1}))
    sim.wait(0.8)
end

closeGripper=function()
    sim.tubeWrite(gripperCommunicationTube,sim.packInt32Table({0}))
    sim.wait(0.8)
end

waitReachTarget=function()
    repeat
        sim.switchThread()
        p1=sim.getObjectPosition(youBot_target,-1)
        p2=sim.getObjectPosition(youBot_reference,-1)
        p={p2[1]-p1[1],p2[2]-p1[2]}
        pError=math.sqrt(p[1]*p[1]+p[2]*p[2])
    until (pError<0.001)
end

sensor_distance=function(sensor)

        dist = getDistance(sensor)

        printf ("Distancia primera")
        print (dist)

        p_dist = dist - 0.1 --Distancia que se quiere conseguir 0.1

        p1=sim.getObjectPosition(youBot_target,-1)
        pfin = p1[1]-p_dist
        ptarget = {pfin, p1[2], p1[3]}
        sim.setObjectPosition(youBot_target, -1,  ptarget)
        waitReachTarget()
        dist = getDistance(sensor)
        printf ("Distancia final")
        print (dist)

end


function getDistance(sensor)

    max_dist = 0.5 --0.5m distancia maxima asignada al sensor en la simulacion
    local detected, distance
    detected,distance=sim.readProximitySensor(sensor)
    if (detected<0) then
        distance=max_dist
    end
    return  distance
end
```

```
function sysCall_threadmain()

-- Initialization for YouBot
    -- Get YouBot Handle
    you_bot = sim.getObjectHandle('youBot')
    youBot_reference = sim.getObjectHandle('youBot_ref')
    youBot_target = sim.getObjectHandle('youBot_TargetPosition')
    object = sim.getObjectHandle('object')
    arm_target = sim.getObjectHandle('youBot_armTarget')
    arm_tip = sim.getObjectHandle('youBot_armTip')
    take = sim.getObjectHandle('take')
    take1 = sim.getObjectHandle('take1')
    soltar = sim.getObjectHandle('soltar')
    soltar1 = sim.getObjectHandle('soltar1')
    soltar2 = sim.getObjectHandle('soltar2')
    ruta1 = sim.getObjectHandle('ruta1')
    ruta2 = sim.getObjectHandle('ruta2')
    ruta3 = sim.getObjectHandle('ruta3')

    sensor_l = sim.getObjectHandle('left_ray_sensor')
    sensor_r = sim.getObjectHandle('right_ray_sensor')

    youbot_init_position = {1.5, 4, 0.1}
    sim.setObjectPosition(you_bot, -1, youbot_init_position)
    youbot_init_orientation = {-90*math.pi/180, 0, -90*math.pi/180}
    sim.setObjectOrientation(you_bot, -1, youbot_init_orientation)

    sim.wait(5)

    gripperCommunicationTube=sim.tubeOpen(0,'youBotGripperState'..sim.getNameSuffix(nil),1)
    closeGripper()
    printf ("Siguiendo ruta 1")
    sim.followPath(youBot_target, ruta1, 3, 0, 0.3, 1)
    sim.wait(3)

        if (sim.readProximitySensor(sensor_l)>0) then

            sensor_distance(sensor_l)

        end

        if (sim.readProximitySensor(sensor_r)>0) then

            sensor_distance(sensor_r)

        end
```

```
            openGripper()
            sim.setObjectParent(arm_target,-1,true)
            sim.wait(2)
            sim.followPath(arm_target, take, 3, 0, 0.1, 0.05)
            closeGripper()
            sim.followPath(arm_target, take1, 3, 0, 0.1, 0.05)
            sim.setObjectParent(arm_target,youBot_reference,true)
            openGripper()
            sim.wait(1)
            sim.setObjectPosition(arm_target, -1,  {-3.2388e-01, 1.5209e+00, 2.1800e-01})

    printf ("Siguiendo ruta 2")
    sim.followPath(youBot_target, ruta2, 3, 0, 0.4, 1)
    sim.wait(3)
```

```
        if (sim.readProximitySensor(sensor_l)>0) then

            sensor_distance(sensor_l)

        end

        if (sim.readProximitySensor(sensor_r)>0) then

            sensor_distance(sensor_r)

        end

            openGripper()
            sim.setObjectParent(arm_target,-1,true)
            sim.wait(2)
            sim.followPath(arm_target, soltar, 3, 0, 0.1, 0.05)
            closeGripper()
            sim.followPath(arm_target, soltar1, 3, 0, 0.1, 0.05)
            openGripper()
            sim.followPath(arm_target, soltar2, 3, 0, 0.1, 0.05)
            sim.setObjectParent(arm_target,youBot_reference,true)
            sim.wait(1)

    printf ("Siguiendo ruta 3")
    sim.followPath(youBot_target, ruta3, 3, 0, 0.5, 1)
    waitReachTarget()
    sim.wait(1)
    printf ("Fin de la ruta")

    simu_time = sim.getSimulationTime()
    print ("Tiempo simulacion (segundos): "..simu_time)
    sim.stopSimulation()
end

function sysCall_cleanup()
    -- Put some clean-up code here
end
```

Non-threaded child script (ME_Platfo2_sub1)

```
-- Initialization for YouBot
function sysCall_init()

    -- Get YouBot Handle
    you_bot = sim.getObjectHandle('youBot')
    youBot_reference = sim.getObjectHandle('youBot_ref')
    youBot_target = sim.getObjectHandle('youBot_TargetPosition')
    sim.setObjectPosition(youBot_target,sim.handle_parent,{0,0,0})
    sim.setObjectOrientation(youBot_target,sim.handle_parent,{0,0,0})
    sim.setObjectParent(youBot_target,-1,true)

    -- Prepare initial values for four wheels
    wheel_joints = {-1,-1,-1,-1} -- front right, front left, rear left, rear right
    wheel_joints[1] = sim.getObjectHandle('rollingJoint_fr')
    wheel_joints[2] = sim.getObjectHandle('rollingJoint_fl')
    wheel_joints[3] = sim.getObjectHandle('rollingJoint_rl')
    wheel_joints[4] = sim.getObjectHandle('rollingJoint_rr')

    omega_1 = 0
    omega_2 = 0
    omega_3 = 0
    omega_4 = 0

    -- youBot size parameters
    wheel_R = 0.05 -- 0.05 m
    a = 0.15
    b = 0.235

end
function sysCall_cleanup()


end

function sysCall_sensing()


end
```

```
-- Inverse kinematics
function movementOmniwheel(vx, vy, omega, wheel_R, a, b)
    omega_1 = (vy - vx + (a+b)*omega)/wheel_R
    omega_2 = (vy + vx - (a+b)*omega)/wheel_R
    omega_3 = (vy - vx - (a+b)*omega)/wheel_R
    omega_4 = (vy + vx + (a+b)*omega)/wheel_R

    -- set the right direction for each wheel
    v_wheel_1 = -omega_1
    v_wheel_2 = -omega_2
    v_wheel_3 = -omega_3
    v_wheel_4 = -omega_4
end
-- Control joints of YouBot
function sysCall_actuation()


        delta_pos = sim.getObjectPosition(youBot_target, you_bot)

        delta_ori = sim.getObjectOrientation(youBot_target, you_bot)

        center_velocity = {delta_pos[2], delta_pos[3], delta_ori[1]}

        movementOmniwheel(center_velocity[1], center_velocity[2], center_velocity[3], wheel_

        -- Apply the desired wheel velocities
        sim.setJointTargetVelocity(wheel_joints[1], v_wheel_1)
        sim.setJointTargetVelocity(wheel_joints[2], v_wheel_2)
        sim.setJointTargetVelocity(wheel_joints[3], v_wheel_3)
        sim.setJointTargetVelocity(wheel_joints[4], v_wheel_4)


end
```

Non-threaded child script (rollingJoint_rl)

```
function __setObjectPosition__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and (sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectPosition(a,b,c)
end
function __setObjectOrientation__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and (sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectOrientation(a,b,c)
end
function sysCall_init()
    rolling=sim.getObjectHandle('rollingJoint_rl')
    slipping=sim.getObjectHandle('slippingJoint_rl')
    wheel=sim.getObjectHandle('wheel_respondable_rl')
end
-- Following script resets the second joint on the omni-wheel (important to achieve the desired omni-wheel effect)


function sysCall_cleanup()

end

function sysCall_actuation()
    sim.resetDynamicObject(wheel)
    __setObjectPosition__(slipping,rolling,{0,0,0})

    __setObjectOrientation__(slipping,rolling,{math.pi/4,0,math.pi})
    __setObjectPosition__(wheel,rolling,{0,0,0})
    __setObjectOrientation__(wheel,rolling,{0,0,0})
end
```

Non-threaded child script (rollingJoint_rr)

```
function __setObjectPosition__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and (sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectPosition(a,b,c)
end
function __setObjectOrientation__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and (sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectOrientation(a,b,c)
end
function sysCall_init()
    rolling=sim.getObjectHandle('rollingJoint_rr')
    slipping=sim.getObjectHandle('slippingJoint_rr')
    wheel=sim.getObjectHandle('wheel_respondable_rr')
end
-- Following script resets the second joint on the omni-wheel (important to achieve the desired omni-wheel effect)


function sysCall_cleanup()

end

function sysCall_actuation()
    sim.resetDynamicObject(wheel)
    __setObjectPosition__(slipping,rolling,{0,0,0})
    __setObjectOrientation__(slipping,rolling,{-math.pi/4,0,0})
    __setObjectPosition__(wheel,rolling,{0,0,0})
    __setObjectOrientation__(wheel,rolling,{0,0,0})
end
```

Non-threaded child script (rollingJoint_fl)

```
function __setObjectPosition__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and (sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectPosition(a,b,c)
end
function __setObjectOrientation__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and (sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectOrientation(a,b,c)
end
function sysCall_init()
    rolling=sim.getObjectHandle('rollingJoint_fl')
    slipping=sim.getObjectHandle('slippingJoint_fl')
    wheel=sim.getObjectHandle('wheel_respondable_fl')
end
-- Following script resets the second joint on the omni-wheel (important to achieve the desired omni-wheel effect)


function sysCall_cleanup()

end

function sysCall_actuation()
    sim.resetDynamicObject(wheel)
    __setObjectPosition__(slipping,rolling,{0,0,0})
    __setObjectOrientation__(slipping,rolling,{-math.pi/4,0,0})
    __setObjectPosition__(wheel,rolling,{0,0,0})
    __setObjectOrientation__(wheel,rolling,{0,0,0})
end

```

Non-threaded child script (rollingJoint_fr)

```
function __setObjectPosition__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and (sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectPosition(a,b,c)
end
function __setObjectOrientation__(a,b,c)
    -- compatibility routine, wrong results could be returned in some situations, in CoppeliaSim <4.0.1
    if b==sim.handle_parent then
        b=sim.getObjectParent(a)
    end
    if (b~=-1) and (sim.getObjectType(b)==sim.object_joint_type) and (sim.getInt32Parameter(sim.intparam_program_version)>=40001) then
        a=a+sim.handleflag_reljointbaseframe
    end
    return sim.setObjectOrientation(a,b,c)
end
function sysCall_init()
    rolling=sim.getObjectHandle('rollingJoint_fr')
    slipping=sim.getObjectHandle('slippingJoint_fr')
    wheel=sim.getObjectHandle('wheel_respondable_fr')
end
-- Following script resets the second joint on the omni-wheel (important to achieve the desired omni-wheel effect)


function sysCall_cleanup()

end

function sysCall_actuation()
    sim.resetDynamicObject(wheel)
    __setObjectPosition__(slipping,rolling,{0,0,0})

    __setObjectOrientation__(slipping,rolling,{math.pi/4,0,math.pi})
    __setObjectPosition__(wheel,rolling,{0,0,0})
    __setObjectOrientation__(wheel,rolling,{0,0,0})
end
```

Non-threaded child script (Rectangle7)

```
function sysCall_init()
    j1=sim.getObjectHandle('youBotGripperJoint1')
    j2=sim.getObjectHandle('youBotGripperJoint2')
    communicationTube=sim.tubeOpen(0,'youBotGripperState'..sim.getNameSuffix(nil),1)
    opening=1
end

function sysCall_cleanup()

end

function sysCall_actuation()
    data=sim.tubeRead(communicationTube)
    if (data) then

        opening=sim.unpackInt32Table(data)[1]
    end

    if (opening==0) then
        sim.setJointTargetVelocity(j2,0.04) --closing
    else
        sim.setJointTargetVelocity(j2,-0.04) --opening
    end

    sim.setJointTargetPosition(j1,sim.getJointPosition(j2)*-0.5)
end
```

REFERENCIAS

[1] KUKA Laboratories GmbH, «generationrobots,» [En línea]. Available: https://www.generationrobots.com/img/Kuka-YouBot-Technical-Specs.pdf. [Último acceso: 2021].

[2] CoppeliaSim, «Coppelia Robotics (Downloads),» [En línea]. Available: https://www.coppeliarobotics.com/downloads. [Último acceso: 2021].

[3] CoppeliaSim, «coppeliarobotics,» [En línea]. Available: https://www.coppeliarobotics.com/. [Último acceso: 2021].

[4] CoppeliaSim, «Coppelia Robotics (Regular API reference),» [En línea]. Available: https://www.coppeliarobotics.com/helpFiles/en/apiFunctions.htm. [Último acceso: 2021].

[5] CoppeliaSim, «coppeliarobotics,» [En línea]. Available: https://www.coppeliarobotics.com/helpFiles/. [Último acceso: 2021].

[6] R. Ierusalimschy, L. Henrique de Figueiredo y W. Celes, «lua.org,» 2007. [En línea]. Available: https://www.lua.org/manual/5.1/es/manual.html. [Último acceso: 2021].

[7] CoppeliaSim, «coppeliarobotics,» [En línea]. Available: https://www.coppeliarobotics.com/helpFiles/en/childScripts.htm. [Último acceso: 2021].

[8] L. Armesto, «Youtube (Creación de trayectorias/caminos en el suelo | CoppeliaSim (V-REP)),» 11 02 2020. [En línea]. Available: https://www.youtube.com/watch?v=0dh9quzGpW8&list=PLjzuoBhdtaXOYfcZOPS98uDTf4aAoDSRR&index=28. [Último acceso: 2021].

[9] CoppeliaSim, «coppeliarobotics,» [En línea]. Available: https://www.coppeliarobotics.com/helpFiles/en/solvingIkAndFk.htm. [Último acceso: 2021].

[10] CoppeliaSim, «coppeliarobotics,» [En línea]. Available: https://www.coppeliarobotics.com/helpFiles/en/basicsOnIkGroupsAndIkElements.htm. [Último acceso: 2021].

[11] j. a. r. king, «Youtube (IK en CoppeliaSim (Tutorial)),» 8 06 2021. [En línea]. Available: https://www.youtube.com/watch?v=dB8ebrjUPdQ. [Último acceso: 2021].

[12] m. Ninja, «Youtube (Vrep (CoppeliaSim) Inverse kinematics tutorial (before version 4.2.0)),» 11 12 2018. [En línea]. Available: https://www.youtube.com/watch?v=eTd6mOk6Njw. [Último acceso: 2021].

[13] V.-R. Tutorials, «Youtube (Vrep script writing),» 5 10 2016. [En línea]. Available: https://www.youtube.com/watch?v=Zt21o1qsOB4. [Último acceso: 2021].

[14] B. Adamov, «Influence of mecanum wheels construction on accuracy of the omnidirectional platform navigation (on exanple of KUKA youBot robot),» de *International Conference on Integrated Navigation Systems (ICINS)*, San Petersburgo, 2018.

[15] Roboticoss, «Youtube (Robots Móviles Autónomos 5 : Modelo Cinemático y Simulación Robot Omnidireccional),» 31 10 2018. [En línea]. Available: https://www.youtube.com/watch?v=CSkHZhpEwsE. [Último acceso: 09 1 2021].

[16] N. Robotics, «Youtube (Modern Robotics, Chapter 13.2: Omnidirectional Wheeled Mobile Robots (Part 1 of 2)),» 10 05 2018. [En línea]. Available: https://www.youtube.com/watch?v=NcOT9hOsceE. [Último acceso: 2021].

[17] L. Armesto, "Youtube (Sensores de proximidad | CoppeliaSim (V-REP))," 3 02 2020. [Online]. Available: https://www.youtube.com/watch?v=vULeGmxff5c&list=PLjzuoBhdtaXOYfcZOPS98uDTf4aAoDSRR&index=13. [Accessed 2021].

[18] L. Armesto, «Youtube (Cómo Usar los Sensores de Proximidad para Permanecer entre Paredes | CoppeliaSim (V-REP)),» 12 02 2020. [En línea]. Available: https://www.youtube.com/watch?v=YKZmv7yA9jQ&list=PLjzuoBhdtaXOYfcZOPS98uDTf4aAoDSRR&index=26. [Último acceso: 2021].