UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIERÍA INDUSTRIAL VALENCIA

TRABAJO FIN DE GRADO EN INGENIERÍA DE LAS TECNOLOGÍAS INDUSTRIALES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESTUDIO DE ENTORNOS VIRTUALES PARA ROBOTS MÓVILES. DESARROLLO DE APLICACIONES DE CONTROL DE ROBOTS MÓVILES CON CONFIGURACIÓN DIFERENCIAL.

AUTOR: Ángel Vicente Cánovas.

TUTOR: Ángel Valera Fernández.

Curso Académico: 2019-20

Agradecimientos

a Blanca, compañera de la asignatura de LAC, por su ayuda y apoyo,

y a Ángel por su paciencia y orientación.

RESUMEN.

Motivado por lo que se conoce como Industria 4.0, recientemente el control de robots manipuladores y móviles es un área de interés para investigadores y desarrolladores industriales. La robótica va a detonar un importante avance científico y tecnológico en diversas áreas de la mecánica, control, electrónica, computación, salud y seguridad, entre otros (1). Por todo ello, no es de extrañar que sea una de las asignaturas más importantes y comunes en Ingeniería. En el presente Trabajo Fin de Grado (TFG) se proponía trabajar en esta área, en concreto con el desarrollo de aplicaciones relacionadas con la navegación de robots móviles y el desarrollo de misiones con dichos sistemas.

En este TFG se pretendía utilizar el Lego Mindstorms EV3, el robot móvil real disponible en el Laboratorio de Robótica del Dpto. de Ingeniería de Sistemas y Automática de la Universitat Politècnica de València. De hecho, se empezó a trabajar y se pudieron obtener ejecuciones reales con los robots físicos. Sin embargo, por los problemas provocados por la pandemia relacionada con el COVID-19, se tuvo que cambiar el enfoque el trabajo, por lo que se estuvieron estudiando diversas soluciones para poder trabajar con robots simulados. Así, se propone analizar y utilizar dos herramientas para trabajar con robots virtuales: la herramienta Robot Virtual Worlds, simulador original de RobotC, la plataforma usada en las prácticas de la asignatura de Laboratorio de Automatización y Control en el grado; y CoppeliaSim como segunda opción.

De esta forma, se trabajará con estas dos herramientas, explicando su funcionamiento y modo de empleo, estudiándose las características básicas y las ventajas e inconvenientes de ellas, comparándolas y desarrollándose diversas tareas de programación de robot móviles en ambos entornos como ejemplo, una de control cinemático de los robots móviles diferenciales y otra aplicación utilizando sensores. Además, se estudia LeoCAD como posible solución fácil al modelado del robot. Así daríamos una oportunidad para continuar trabajos de robótica en casa o a distancia y no volver a tener este problema.

Palabras clave:

Control por computador; control de robots; robótica móvil; navegación automática; control cinemático; robótica; simulación; MATLAB; Lego Mindstorms EV3; CoppeliaSim; Robot Virtual Worlds; RobotC.

ABSTRACT.

Motivated by the Industry 4.0, the control of robot manipulators and mobile robots has recently become an area of interest for industrial researchers and developers. According to several magazines, we can define robotics as a science that bring together various technological branches or disciplines, with the aim of designing robotic machines capable of performing automated tasks or simulating human or animal behaviours (1). For these reasons, it is not surprising that robotics is one of the most important and common subjects in engineering. This subject area is the purpose of this TFG, specifically the development of applications related to the navigation of mobile robots and the development of missions with these systems.

This project intended to use the Lego Mindstorms EV3, the real mobile robots available in the Robotics Laboratory of the Department of Systems and Automatic Engineering of the Polytechnic University of Valencia. In fact, work began and real executions with physical robots were successfully achieved. However, due to the problems caused by the pandemic COVID-19, the focus of the work changed. Therefore, many solutions have been studied in order to work with simulated robots. Consequently, analysing and using two tools to work with virtual robots has been proposed: Robot Virtual Worlds, environment developed by RobotC, which is the current platform used in the subject of Laboratory of Automatic and Control; and CoppeliaSim as second option.

Therefore, the TFG will work with these two tools, explaining their functions and method of use, studying the basic characteristics and the advantages of them, and developing various mobile robot control task in both environments, one of them is related to the kinematic control of mobile robots with differential configuration, and other application where sensors are used. In addition, LeoCAD is studied as possible solution to model the robot. This, an opportunity to continue robotic works at home is given to you.

Keywords:

Computer control; robot control; mobile robotics; automatic navigation; kinematic control; robotics; simulation; MATLAB; Lego Mindstorms EV3; CoppeliaSim; Robot Virtual Worlds; RobotC.

ÍNDICE.

ÍNDICE DE FIGURAS.

1. INTRODUCCIÓN.

Según varias revistas se puede definir la robótica como una ciencia que aglutina varias ramas tecnológicas o disciplinas, con el objetivo de diseñar maquinas robotizadas que sean capaces de realizar tareas automatizadas o de simular el comportamiento humano o animal (2). Combina diversas disciplinas como la mecánica, la electrónica, la informática, la inteligencia artificial, la ingeniería de control y la física y se ocupa del diseño, construcción, operación, estructura, manufactura, y aplicación de los robots (3). Según la Federación Internacional de Robótica (IFR), entre 2017 y 2020 se instalarán más de 1.7 millones de nuevos robots industriales en fábricas de todo el mundo. Para 2020 habrá más de 3 millones de robots industriales en uso en nuestras fábricas (4). La robótica está en un crecimiento imparable. En Wikipedia se puede ver que la robótica educativa es una subdisciplina de la robótica aplicada al ámbito educativo que se centra en el diseño, el análisis, la aplicación y la operación de robots. Se puede enseñar en todos los niveles educativos, desde la educación infantil y primaria hasta los posgrados. La robótica también se puede utilizar para fomentar y facilitar la instrucción en otras disciplinas, tales como la programación informática, la inteligencia artificial o la ingeniería de diseño (5).

Es conocimiento común que el objeto de estudio de la robótica es el robot. La definición en la RAE robot es “máquina o ingenio electrónico programable que es capaz de manipular objetos y realizar diversas operaciones” (6). Los robots pueden clasificarse de diversas maneras. Una de estas maneras es según su estructura, que divide a los robots en poliarticulados, móviles, androides, zoomórficos o híbridos. Aquí se va a trabajar con un robot móvil.

Un robot móvil es una máquina automática capaz de trasladarse en cualquier ambiente dado. Los robots móviles tienen la capacidad de moverse en su entorno y no se fijan a una ubicación física. Los robots móviles son un foco importante de la investigación actual y casi de cada universidad importante que tenga uno o más laboratorios que se centran en la investigación de robots móviles (7). Concretamente, el desarrollo de este trabajo está centrado en el control de un robot móvil con una configuración diferencial, es decir, no hay ruedas directrices, el cambio de dirección se realiza modificando la velocidad relativa de las ruedas a izquierda y derecha.

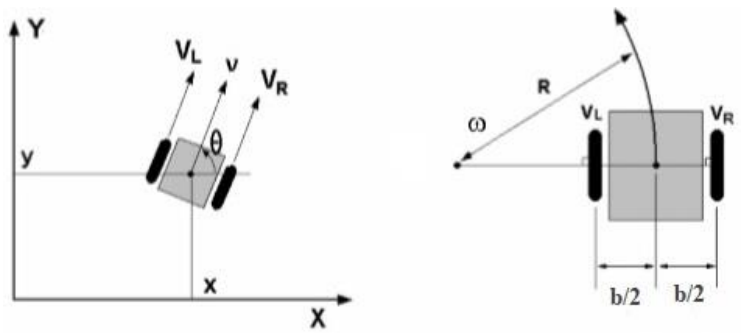


Figura 1. Configuración diferencial de un robot.

Con todo lo anterior dicho era evidente que esta ciencia debía de implantarse también en asignaturas del *Grado en Ingeniería en Tecnologías Industriales* de la *Universitat Politècnica de València* como lo es *Laboratorio de Automatización y Control (LAC)*. Por todo ello no es de extrañar que la robótica sea el área principal de este trabajo.

De esta forma, en el TFG se van a abordar dos tareas. La primera está relacionada con el control cinemático de los robots móviles. Para ello se tendrá que obtener el modelo cinemático directo e inverso de un robot móvil con configuración diferencial. A partir de estos modelos se podrá desarrollar un control cinemático de manera que se podrá indicar la trayectoria que el robot deberá realizar.

Una vez obtenido el control del movimiento del robot, se propone resolver una aplicación con el robot móvil. En concreto se pretende desarrollar un robot que, con un sensor de distancia, detecte los objetos y sea capaz de sacarlos de una determinada zona (o área de trabajo).

Cabe destacar que este TFG se planteó sobre el mes de febrero, y se pretendía utilizar los robots móviles reales disponibles en el Laboratorio de Robótica del *Dpto. de Ingeniería de Sistemas y Automática* de la *Universitat Politècnica de València*. Sin embargo, por los problemas provocados por la pandemia relacionada con el COVID-19, se tuvo que cambiar el enfoque el trabajo, por lo que se estuvieron estudiando diversas soluciones para poder trabajar con robots simulados. Así el enfoque final del trabajo es analizar y utilizar dos herramientas para trabajar con robots virtuales y solucionar este problema que se planteó en un principio y dar la oportunidad de poder continuar trabajos e investigaciones desde casa, sin necesidad de robot físicos o laboratorios.

La primera herramienta es un entorno de simulación perteneciente a RobotC, la actual plataforma con la que se trabaja en la asignatura de Laboratorio de Automatización y Control en la UPV para la programación de un robot Lego Mindstorms EV3. Esta plataforma bastante sencilla e intuitiva de usar es muy común en el ámbito de la educación. Además, también es muy usada para competiciones de no muy alto nivel. Por todo ello fue la primera opción de simulador que se escogió. Para ello se pueden utilizar diferentes escenarios y tipos de robots. Además, permite generar nuevos escenarios donde se pueden posicionar paredes, obstáculos móviles, etc. Sin embargo, su funcionalidad queda bastante limitada. Su utilidad está ambientada a lograr pequeñas pruebas y retos educativos. Fácil pero poco polivalente. Como resultado se decidió buscar una segunda opción.

La segunda herramienta es CoppeliaSim, un entorno de simulación de robots muy potente y utilizada en el área de la robótica. Se trata de un entorno de desarrollo con una arquitectura de control distribuida que permite modelar entornos y robots, lo que hace que sea un simulador muy versátil y potente para el desarrollo de aplicaciones multi robot. Un programa multiplataforma que te permite generar funciones con scripts internos o con diversos programas externos. En este trabajo se ha optado hacer una conexión a través de una API remota con MATLAB. Por el contrario, este polivalente y gratis simulador, aunque permite

el diseño de casi cualquier escenario y robots, es bastante más complejo y complicado de usar que RVW.

1.1 Objetivos.

Como se ha dicho anteriormente, este Trabajo Fin de Grado tenía en un comienzo otro propósito distinto desarrollando diversos controles del robot real que se usa en el laboratorio la asignatura anteriormente mencionada. Tras el COVID-19, esto no podía ser posible, pues no se permitía el acceso a dicho laboratorio impidiendo la realización del antiguo propósito e, incluso, de las prácticas de esta asignatura. Entonces, surge un nuevo problema que hasta ahora no había sido ninguna necesidad: como se podría llevar este trabajo a casa. El fin de este trabajo es dar una solución a ese problema ofreciendo posibilidades a los alumnos de poder continuar sus prácticas o trabajos en casa.

Los objetivos del trabajo son:

- Estudiar la opción de instalar la plataforma de programación usada en LAC, RobotC, en casa.
- Valorar, conocer y explicar el simulador que trae RobotC, Robot Virtual World, como posible solución a la falta del robot físico real.
- Valorar, conocer y explicar el simulador CoppeliaSim como segunda opción más compleja y versátil.
- Enseñar como conectar fácilmente CoppeliaSim con el programador externo MATLAB a través de una API remota y el funcionamiento del conjunto.
- Comparar ambos simuladores, viendo así cuando es más oportuno usar cuál.
- Enseñar un método para obtener o crear el diseño de un modelo 3D del robot EV3 usado en clase.
- Poner en práctica los dos simuladores mencionados mediante programas de controles cinemáticos o la función “Robot Limpiador”, propios de un nivel de prácticas de una asignatura de robótica en un grado de Ingeniería Industrial.
- Probar un comportamiento o escena un poco más compleja, como el algoritmo de Braitenberg, que solo pueda simular CoppeliaSim.
- Probar un comportamiento o escena un poco más compleja de seguimiento de paredes, que solo pueda simular CoppeliaSim.

Cabe destacar que en este trabajo la mayoría de la explicación de funcionamiento de ambos simuladores se hace mientras se resuelven ejemplos resueltos.

1.2 Estructura del documento.

La memoria de este trabajo presenta la siguiente estructura:

1. Introducción y objetivos.
2. Listado y breve descripción de cada uno de los programas y herramientas usadas para el trabajo.
3. Planteamiento y explicación de las tareas, aplicaciones y escenarios que se van a poner a prueba con los dos simuladores y obtención de trayectorias y gráficas teóricas y reales como referencia para la comparación.
4. Explicación de la funcionalidad, características y modo de empleo del simulador Robot Virtual Worlds con la simulación de dos funciones de ejemplo, una de control cinemático y otra función usando sensores.
5. Modelación del robot en LeoCAD y CoppeliaSim.
6. Explicación de la funcionalidad, características y modo de empleo del simulador CoppeliaSim con la simulación de dos funciones de ejemplo, una de control cinemático y otra función usando sensores. Además, probar este simulador con aplicaciones imposibles de realizar en RVW, como una probando el robot de Braitenberg u otra donde el robot desarrolla un seguimiento de paredes
7. Conclusiones.
8. Anexos.
9. Referencias.

1.3 Carpetas compartidas.

A continuación, se ha compartido un enlace que lleva a una carpeta compartida en Drive donde se puede ver y descargar todos los algoritmos, programas, funciones, escenarios, archivos, videos y demás que se han ido desarrollando durante este trabajo. Su contenido puede ser usado como referencia, ayuda o para comprobar lo resultados si se desea. A lo largo del trabajo se ira haciendo referencias a archivos de esta carpeta añadiendo la dirección de estos por si quieren acceder a ellos.

El enlace:

https://drive.google.com/drive/folders/1ntrfnRlZJRewaK7qiWV_TuQ9qjkweuqX?usp=sharing

2. PROGRAMAS Y HERRAMIENTAS UTILIZADAS.

Las plataformas y herramientas escogidas para la realización de este TFG, que más adelante se describirán, son:

- **Ordenador con Windows**. De todos modos, el método desarrollado en este trabajo para la simulación de robots es válido para MacOS y Linux también.
- **EV3 de LEGO Mindstorms.** (8) Robot móvil de LEGO utilizado en las prácticas de robótica de la asignatura LAC en la UPV a partir del cual se desarrolla nuestro trabajo.
- **ROBOTC for LEGO Mindstorms 4.X.** Plataforma para la programación de Robots, sencilla y fácil de usar con programación parecida a C++. También usada en las prácticas para la misma programación del EV3. (9)
- **Robot Virtual Worlds – LEGO 4.X.** Simulador que proporciona ROBOTC. Bastante sencillo e intuitivo. Sin embargo, como más adelante se explicará, su utilidad es limitada y reducida. (10)
- **CoppeliaSim Edu.** Este programa se conocía antes como V-REP. Simulador mucho más completo y polivalente que Robot Virtual Worlds, aunque también bastante más complicado de usar. (11)
- **MATLAB R2020a versión para estudiantes.** MATLAB si utilizará para la programación de los algoritmos se simularán en CoppeliaSim. (12)
- **LeoCAD.** Proveniente de LDraw. LDraw (13) es un estándar abierto para los programas CAD de LEGO que permite al usuario crear modelos y escenas virtuales de LEGO. Este programa nos permitirá crear nuestro modelo 3D de EV3 a pesar de carecer de grandes conocimientos gráficos. (14)
- **API remota del CoppeliaSim para permitir la conexión con MATLAB.**

2.1 LEGO Mindstorms.

Desde 1998 LEGO ha estado envuelto en el ámbito de la tecnología de la programación. Los robots que proporciona LEGO son muy usados en la educación. EV3 fue lanzado en 2013 e incorporado al laboratorio de LAC en los recientes años.

Este ladrillo inteligente (15) incorpora una interfaz iluminada (tres colores) de seis botones que indica el estado de actividad del ladrillo, pantalla de alta resolución (178 x 128 píxeles) que permite visualizar gráficas detalladas y observar los datos proporcionados por los sensores, un altavoz integrado, espacio para almacenar programas en placa (16 MB de memoria Flash y 64 MB de memoria RAM), un puerto USB y lector de tarjetas miniSDHC para ampliar la memoria en 32 GB. Tiene cuatro puertos de entrada para la adquisición de datos y cuatro puertos de salida para la ejecución de comandos. El ladrillo admite además comunicación USB, Bluetooth y Wi-Fi con un ordenador, y posee una interfaz de

programación que permite programar y registrar datos directamente en él. Es compatible con dispositivos móviles y funciona con pilas AA o con la Batería Recargable DC EV3. El ladrillo también está equipado con un procesador ARM 9 con sistema operativo basado en Linux, funciones de programación y registro de datos en el ladrillo compatibles con el software EV3 y concentrador USB 2.0 que permite conectar varios ladrillos en cadena.

En el laboratorio de LAC, además, se posee infinidad de piezas de LEGO Technic y múltiples sensores y motores para así montar nuestro robot móvil como el de la figura 2. Puedes ver el manual para el montaje de un robot como el que se usa en este trabajo en la carpeta de *Modelos* > *Manual Montaje EV3* de la carpeta compartida o en el enlace de la referencia (16) y bajando la página web hasta "*Building Instructions for Robot Educator".*

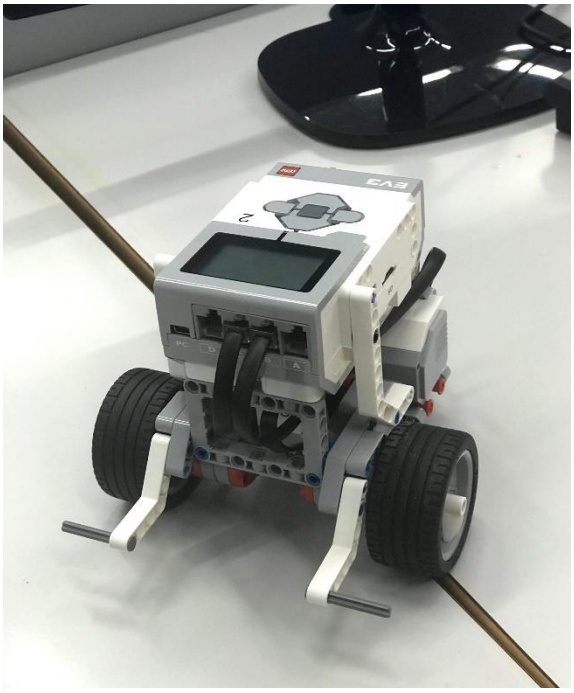


Figura 2. Robot móvil EV3 de prácticas sin sensores puestos.

A partir de aquí se dispone de servomotores interactivos, controles remoto, sensores de color, sensores de contacto, sensores de infrarrojos, giroscopios, etc. para añadir como queramos al robot (17). En las mismas direcciones mencionadas anteriormente también se dispone de manuales de ejemplo para colocar estos sensores.

Se puede encontrar una guía de uso con especificaciones técnicas y demás en la referencia (24)

2.2 ROBOTC Y Robot Virtual Worlds.

ROBOTC es un lenguaje de programación para el desarrollo de robótica educativa y concursos. Es un lenguaje de programación basado en C, con un entorno de desarrollo fácil de usar. Tiene un rápido firmware de alto rendimiento. Además, esta plataforma es la que se usa en las prácticas de clase.

Tiene diferentes versiones según que plataformas de robot quieres usar. Estas posibles plataformas son LEGO Mindstorms (EV3 y NXT), VEX Robotics (VEX EDR y VEZ IQ), CORTEX o Arduino. Entre estas, las más comunes y usadas son LEGO Mindstorms y VEX Robotics. Nosotros trabajamos con la primera: *ROBOTC 4.X for MINDSTORMS*.

Como el trabajo busca la posibilidad de llevarte el trabajo a casa, el precio de este es un factor a tener en cuenta, teniendo varias opciones. Una vez descargada la respectiva versión de ROBOTC tienes 10 días de prueba. Una vez pasada, la licencia cuesta lo siguiente:

	ANUAL	PERMANENTE
1 persona ROBOTC	43.24€	69.71€
6 personas ROBOTC	131.47€	263.82€
30 personas ROBOTC	263.82€	528.53€
1 persona ROBOTC + RVW	78.53€	122.65€
6 personas ROBOTC + RVW	219.71 €	440.29 €
30 personas ROBOTC + RVW	440.29 €	881.47€

La explicación de la interfaz gráfica de ROBOTC o cualquier tipo de explicación o instrucción sobre su uso y funcionamiento no entra en el ámbito de este trabajo, pues es algo que ya es dado en la asignatura de LAC. Lo que sí se va a explicar es el simulador que lleva este programa: Robot Virtual Worlds (RVW).

RVW también tiene distintas versiones y modos para usar y distintas licencias para descargar según la plataforma al igual que ROBOTC. Dentro de la página web de RVW (10), se le da a *PURCHASE* y ahí puedes acceder a las distintas plataformas para descargar. Como ROBOTC, las más importantes son *ROBOTC Robot Virtual Worlds for VEX*, *ROBOTC Robot Virtual Worlds for LEGO* y *Robot Virtual Worlds for Virtual Brick*. Otra opción interesante de licencia es *Robot Virtual Worlds Homework Packs* la cual permite que estudiantes tengan sus propias licencias individuales para usar Robot Virtual Worlds en casa. Para esto, se debe tener una licencia de aula válida para comprar un paquete de tareas. Las licencias de paquetes de tareas se pueden comprar por 180 días (4.40€) o 365 días (8.81€). Otra licencia a tener en cuenta es Ro*bot Virtual Worlds Building License* la cual obtienes un año de acceso a Robot Virtual Worlds para toda su escuela por solo 528.13€.

Una vez descargado RVW, los escenarios virtuales que te vienen son los del paquete *Challenge Pack for EV3*, pero se puede descargar distintos modos de escenarios virtuales navegando entre las distintas pestañas de la misma página web. *Challenge Pack for EV3* tiene muchos retos, pruebas y variedad de escenarios para practicar y mejorar tu programación del EV3. A continuación se va a describir el interfaz gráfica de este:

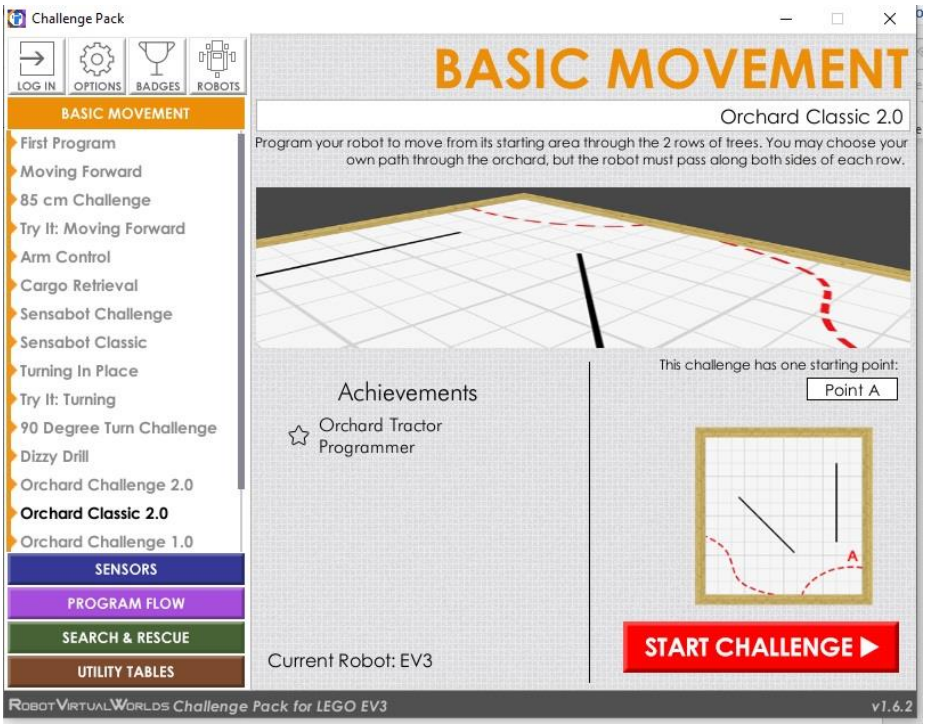


Figura 3. Interfaz gráfica del menú de Challenge Pack for EV3

A la izquierda se tiene el menú de escenarios, donde están todos los escenarios y retos disponibles. Además, esta ordenado según se quiera poner a prueba movimientos básicos, sensores y demás. En *Achievements* estarán los retos que te proponen que intentes. Abajo a la derecha se puede ver el escenario desde un plano de planta y seleccionar desde que punto de los disponibles saldrá el robot. Debajo de esto la opción de empezar la simulación. Arriba a la izquierda, en *ROBOTS*, se puede cambiar y elegir entre distintos tipos de formas del robot EV3.

Figura 4. Menú de robots en Challenge Pack for EV3.

Como se puede ver, solo se dispone de 4 tipos de robots distintos donde cambian la organización de algunos de sus sensores. En *Part Mapping* se puede comprobar que sensores tiene el robot seleccionado y donde. No tener libertad para modificar tu propio modelo de robot es una gran desventaja y limitación para este simulador, a diferencia de CoppeliaSim, como se verá más adelante. Debido a esto muchas funciones y algoritmos, como el de Braitenberg (algoritmo original sobre el que iba a ir el presente TFG), son imposibles de simular o de comprobar. Por ello más adelante se hará una comparación con CoppeliaSim, la segunda opción propuesta para llevarte el trabajo a casa.

De cualquier forma, el modo o pack de RVW que es más interesante para el trabajo es *RVW Level Builder*, situado en el menú de *Tools* de esta web. Lo interesante de este modo es que te deja construir y editar tu propio escenario.

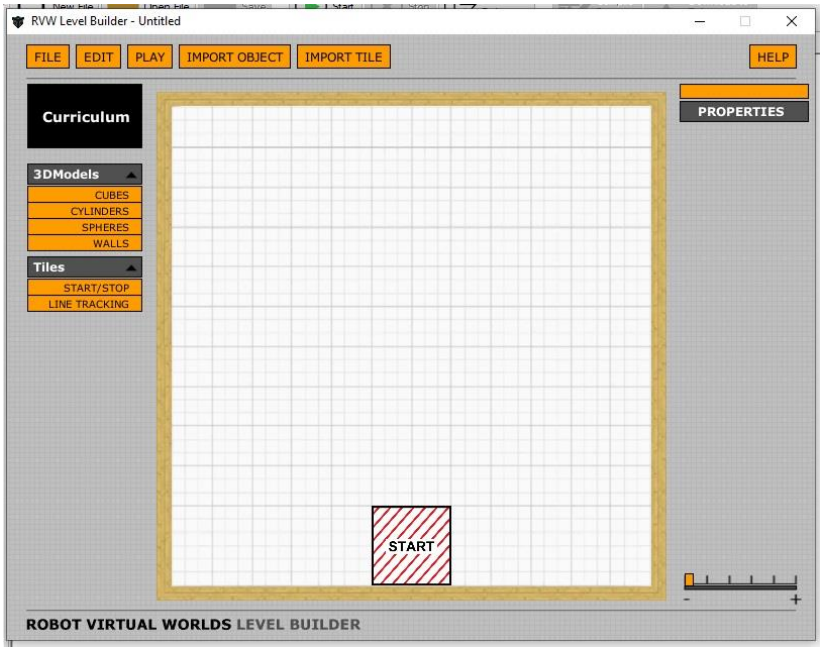


Figura 5. Interfaz gráfica de RVW Level Builder.

RVW Level Builder es muy fácil e intuitivo de usar. A la izquierda se tiene distintos tipos de modelos 3D y losas para diseñar el escenario. También deja importar tus propios modelos de obstáculos, aunque esto también está un poco limitado. Al igual que el otro modo, los robots que puedes usar son únicamente los que te ofrece el simulador. El funcionamiento y características de este simulador se explicará más afondo en el apartado 4 a modo ejemplo.

2.3 CoppeliaSim.

La elección de CoppeliaSim (antes llamado V-REP) se debe a que, a diferencia de RVW, es un simulador muy completo y polivalente. También dispone de una gama de diferentes sensores y actuadores para simular y modelar el comportamiento de casi cualquier robot y de un modo de edición de formas capaz de crear complejos robots. Además, a este simulador se le puede importar diferentes tipos de modelos 3D de numerosos programas de modelado y usarlos tanto como de robot como de obstáculos. Con lo cual se puede modelar o importar infinidad de distintos tipos de robots y escenarios para su simulación como se verá más adelante. El simulador se basa en una arquitectura de control distribuido, es decir, que cada objeto o modelo puede ser controlado independientemente y a la vez.

Este simulador es perfectamente ejecutable con Linux, MacOS o Windows se puede controlar con scripts internos en lenguaje LUA, nodos ROS, o con distintos lenguajes de programación externos como C/C++, Python, Java o MATLAB. Este último es el que se usa en este trabajo. La conexión CoppeliaSim-MATLAB se puede conseguir de manera sencilla con una API remota, lo cual se explicará en el apartado 5.5.1. (18)

Por último, CoppeliaSim tiene una versión educacional gratis muy completa. Esto ha sido esencial para la elección de este simulador. Se tiene pues, un simulador muy complejo, capaz de modelar todo tipo de robot y escenarios, y gratis. Para descargarlo solo se debe ir a su página web, darle a *downloads* en la barra de menús, elegir la plataforma que desees (en este caso Windows), y dar a descargar en la versión *Edu*.

A continuación, se va a explicar resumidamente la interfaz gráfica de CoppeliaSim:

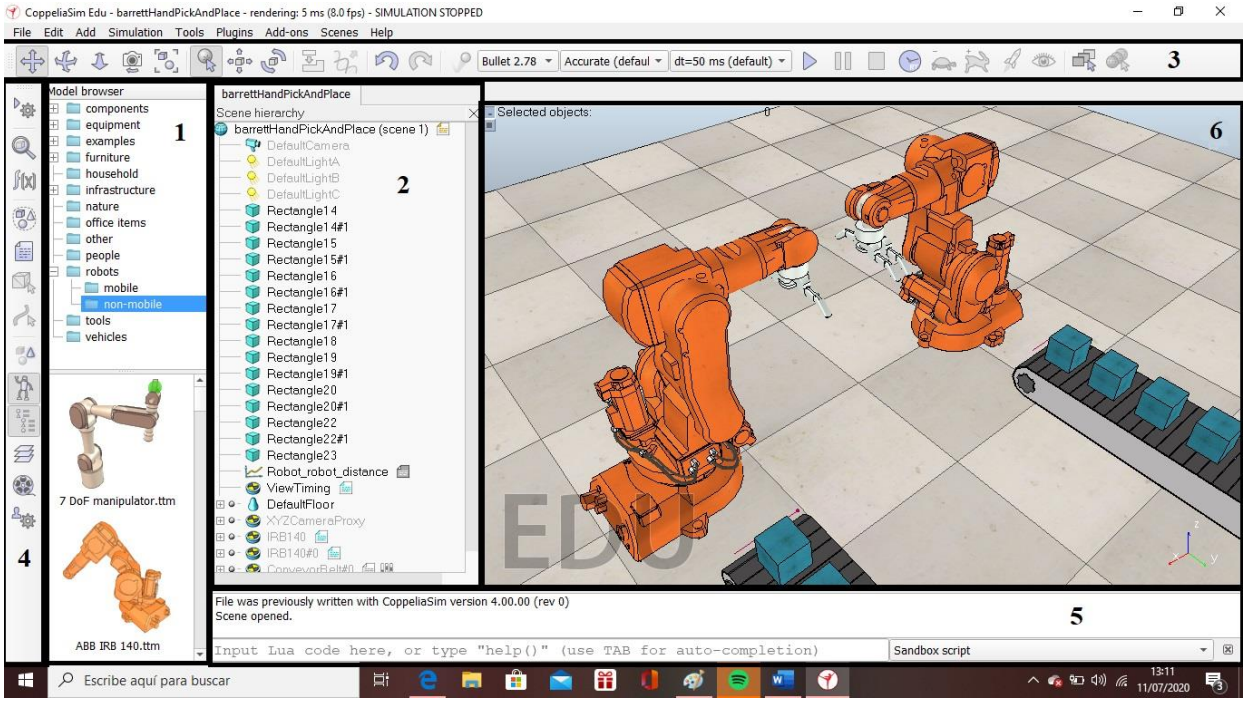


Figura 6. Interfaz gráfica de CoppeliaSim.

1. **Navegador de modelos**. Aquí se encuentran los modelos que ya proporciona CoppeliaSim para crear un escenario. Tan solo se tiene que arrastrar a la escena el que guste. Se pueden ver modelos desde conocidos robots o sensores hasta varios tipos de sillas. En la parte de arriba se sitúa una estructura de carpetas ordenando dicho elementos y en la parte de abajo se tiene pequeñas visualizaciones de los modelos que componen dichas carpetas.
2. **Jerarquía de escena**. Aquí se presenta el contenido de la escena en forma de árbol jerárquico. Algunos objetos se ponen como hijos de otros agrupando a dichos objetos y evitando que se suelten.
3. **Barra de herramientas horizontal.** Aquí aparecen ítems de navegación y exploración de elementos. Contiene herramientas que configuran los simuladores de cámara y la dirección de los objetos.

4. **Barra de herramientas vertical.** Esta barra la compone botones que son enlaces a las funcionalidades principales de CoppeliaSim que usarán sus objetos para, por ejemplo, modificar sus propiedades o configurar la simulación.
5. **Barra de estado.** Aquí se mostrará la información, comandos o mensajes de error o textos ejecutados por los scripts de las operaciones realizadas.
6. **Ventana principal de simulación.** En esta aparecen todos los elementos que conformen nuestro escenario de simulación, así como las simulaciones que vamos a realizar.

El funcionamiento y todas las características de este simulador se explicará más afondo a partir del apartado 5.

2.4 LeoCAD.

Como se ha mencionado anteriormente, según su web (13), LeoCAD es un modelador de LEGO que permite crear tus modelos o escenarios con una interfaz intuitiva que está diseñada para permitir a los nuevos usuarios comenzar a crear nuevos modelos sin tener que pasar demasiado tiempo aprendiendo la aplicación. Al mismo tiempo, tiene un amplio conjunto de características que permite a los usuarios experimentados crear modelos utilizando técnicas más avanzadas. También utiliza la biblioteca de partes LDraw, que tiene casi 10,000 partes diferentes y continúa recibiendo actualizaciones.

Las versiones nativas están disponibles para Windows, Linux y macOS para garantizar que los usuarios estén familiarizados con la interfaz del programa. LeoCAD es de código abierto, por lo que cualquiera puede contribuir con correcciones y características, y siempre será gratuito.

Esta herramienta libre, desarrollada por Leonardo Zide, tiene un uso bastante sencillo, principalmente basta con arrastrar y soltar piezas que luego son orientadas y configuradas según el gusto del diseñador. En el apartado 5.1 se puede ver como ejemplo el proceso de diseño del robot EV3.

En la siguiente figura 7 se muestra el entorno de este programa.

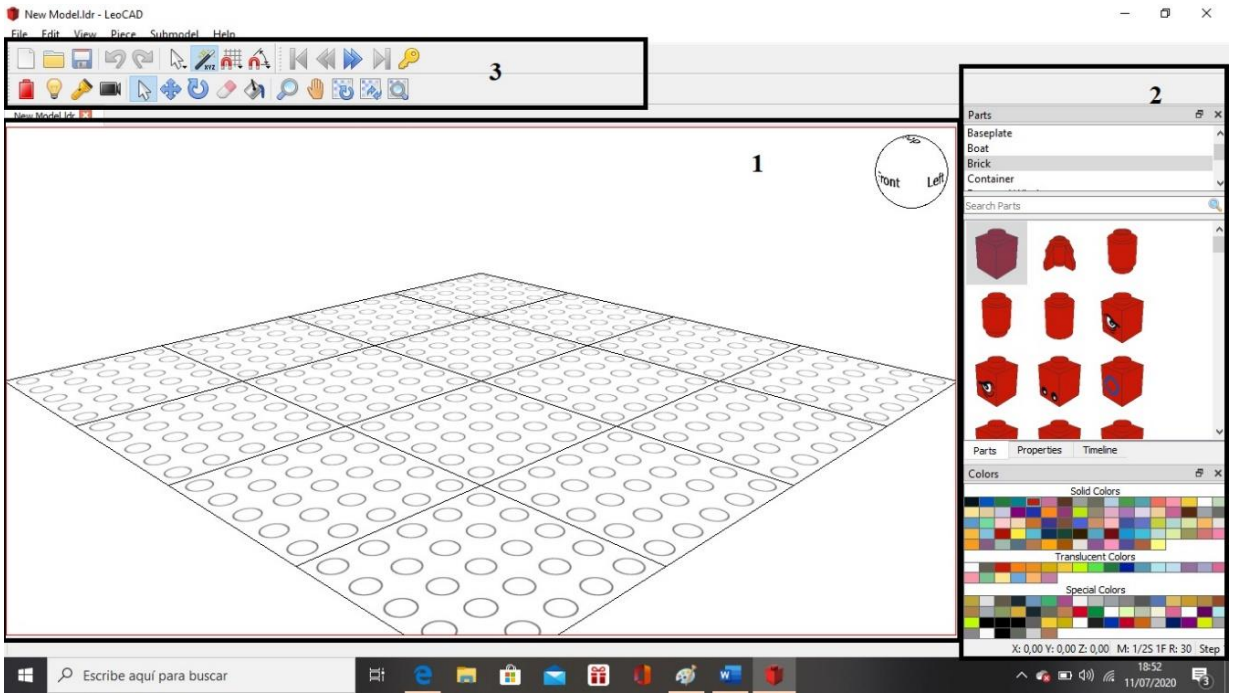


Figura 7. Interfaz gráfica de LeoCAD.

Como se puede observar, es una interfaz bastante sencilla de entender:

1. Ventana principal de diseño.
2. En esta ventana se muestran la infinidad de piezas que tiene este programa ordenadas por categorías que se muestran encima de estas. La mayoría de las piezas que se usan para la construcción de EV3 están en *Technic*. También tiene un buscador para ayudar en la búsqueda de la pieza deseada. Abajo tiene variedad de colores para poder personalizar aún más el diseño.
3. Barra de herramientas.

También se ha probado con otros modeladores de LEGO como LEGO Digital Designer pero solo exporta archivos LXT Files (.lxt) los cuales no son importables para CoppeliaSim.

3. DESCRIPCIÓN DE TAREAS Y ESCENARIOS.

Para probar los dos simuladores que se proponen en este trabajo se han realizado varias tareas o aplicaciones. El objetivo es implementar el control de un movimiento de un robot de LEGO a partir del movimiento de dos motores.

Las primeras de estas tareas están relacionadas con el control cinemático de los robots móviles. Se trata de desarrollar un control cinemático del robot móvil a partir de un control de trayectoria empleando programas de Simulink usando un punto descentralizado. Se comprueba las respuestas a partir de sus gráficas en Matlab. Después, se generan los ficheros en RobotC con estas mismas trayectorias para descargárselos al robot real y se comparan las trayectorias. Por lo tanto, se obtienen la trayectorias teóricas y reales que servirán para poder comparar estos resultados con los que se obtienen de los simuladores propuesto en este trabajo.

El control cinemático para robots móviles determina las acciones de control necesarias para llevarlo desde la posición actual hasta la posición deseada. El control dinámico lo hace para velocidades. Las dos principales formas de control cinemático son control de trayectorias y control de caminos. Aquí nos centramos en el seguimiento de trayectorias. En el control de trayectorias, el robot sigue una curva parametrizada en el tiempo como trayectoria. Esto quiere decir que el robot tiene que estar en un punto de la trayectoria determinado en un momento dado, es decir, los puntos describen una función del tiempo:

$$x(t);\ y(t);\ \theta(t)$$

Se controlará la trayectoria del robot sobre un círculo de 400mm de radio.

Además, también habrá un control dinámico para las velocidades de las ruedas con el método de diseño de reguladores.

En la segunda aplicación, se desarrolla un control del robot que, con un sensor de distancia, detecte los objetos y sea capaz de sacarlos de una determinada área. Además, también usa un sensor de color para intentar no salirse de esa misma área. Para la simulación de la segunda aplicación es necesario la creación del escenario, el cual se enseñará en este apartado y se explicará el procedimiento de su creación con cada una da de las plataformas de simulación propuestas más adelante.

También se realizarán unas tereas extra. Con el término “extra” se refiere a que esta tarea no es usada para comparar ambos simuladores, pues RVW no es capaz de llevarlas a cabo. Y justo ese es el motivo de estas tareas, poner a prueba CoppeliaSim con escenarios imposibles de realizar en el primer simulador propuesto. Además, dichas tareas se realizarán con el programador interno de CoppeliaSim, Lua, para aprovechar y ver también un poco de este. La primera de estas tareas es el “Robot de Braitenberg”, el algoritmo del cual era base el presente trabajo en su comienzo, y la segunda un seguimiento entre paredes saliendo de una espiral. Ambas se explican en su respectivos subapartados.

3.1 Primera tarea. Control cinemático.

La pandemia producida por el COVID-19 hizo cambiar totalmente el rumbo del presente trabajo, pues dejó de estar disponible el acceso al laboratorio. Por suerte, antes de que la cuarentena llegase, se recogieron algunos datos experimentales en dicho laboratorio, que ahora se podrá usar para compararlos con los simuladores propuestos en este TFG. Estos datos experimentales son sobre el control cinemático de trayectorias del EV3. Concretamente, respecto a una trayectoria circular. Se plantearon los esquemas de control, se comprobó a través de Simulink y, una vez obtenido el resultado, se crearon los programas de RobotC a partir de estos esquemas. El resultado que hizo el robot se puede apreciar en las siguientes gráficas.

Simulink

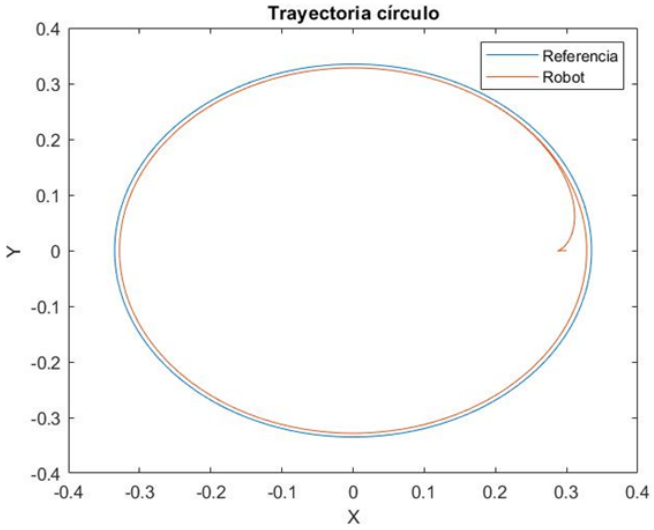


Figura 8. Gráfica resultado de la trayectoria circular en Simulink.

Robot real programado en RobotC

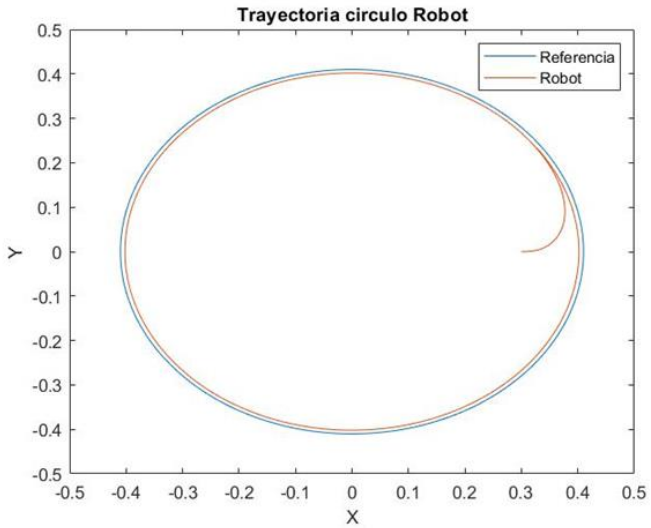
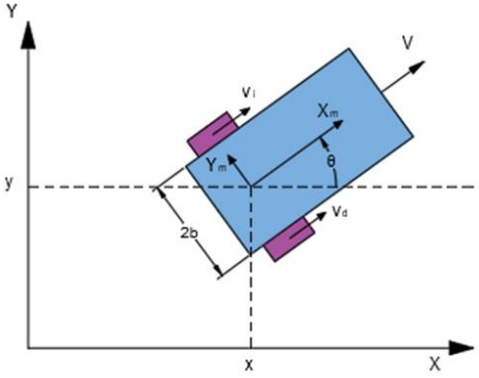


Figura 9. Grafica resultado de la trayectoria circular del robot real.

Todo el proceso experimental se puede ver en el ANEXO I. En este anexo se sitúa la resolución para el control cinemático la trayectoria, todos los esquemas propuestos para Simulink y la dirección y explicación de las funciones implementadas en RobotC para el robot real.

Las funciones en la que se han ido basando los cálculos del control cinemático en Simulink y RobotC y que servirán como base para la futura resolución del control en los apartados de los distintos simuladores se plantea a continuación.

El TFG empezó usando un robot físico real con una configuración diferencial, con ruedas de radio r, separadas entre sí una distancia 2b y que forman un ángulo de Θ grados con el eje x del plano en el que se trabaja.



r = 2.8 cm

b = 5.6 cm

Figura 10. Configuración diferencial del robot EV3

Fijándose en la configuración diferencial del robot (figura 10) se deduce que las velocidades lineales y angular están en función de las ruedas y la distancia entre ellas:

$$\begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \\ -1/2b & 1/2b \end{bmatrix} \begin{bmatrix} v_i \\ v_d \end{bmatrix} \tag{3.1.1}$$

Y a partir de estas se puede calcular la posición y orientación del robot:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} \tag{3.1.2}$$

Sustituyendo se obtiene la ecuación **cinemática directa** del robot diferencial:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos(\theta) & 0\\ \sin(\theta) & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}v\\ w\end{bmatrix} = \begin{bmatrix}\cos(\theta) & 0\\ \sin(\theta) & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}1/2 & 1/2\\ -1/2b & 1/2b\end{bmatrix}\begin{bmatrix}v_i\\ v_d\end{bmatrix}$$

(3.1.3)

El control de trayectoria se trata de un control sobre la curva temporal de cada una de las coordenadas del robot. El control, en este caso, se realiza a partir de la posición y la velocidad de un punto separado una distancia g del eje de tracción del robot (punto medio entre las dos ruedas), es decir, a partir de un punto descentralizado. Con lo cual, la posición del punto descentralizado sería:

$$\begin{bmatrix}x_p\\ y_p\end{bmatrix} = \begin{bmatrix}x + g\cos(\theta)\\ y + g\sin(\theta)\end{bmatrix}$$

(3.1.4)

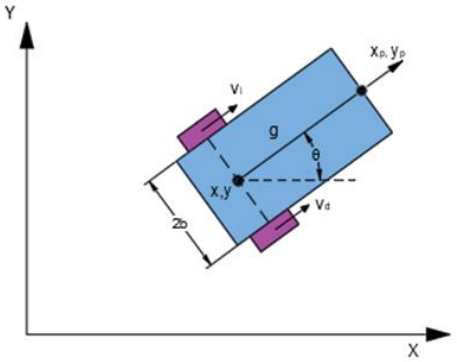


Figura 11. Punto descentralizado del robot.

El punto descentralizado se encuentra a una distancia g = 5.6cm.

La velocidad del punto descentralizado sería la derivada:

$$\begin{bmatrix}\dot{x}_p\\ \dot{y}_p\end{bmatrix} = \begin{bmatrix}\dot{x} - g\sin(\theta)\dot{\theta}\\ \dot{y} + g\cos(\theta)\dot{\theta}\end{bmatrix} = \begin{bmatrix}1 & 0 & -g\sin(\theta)\\ 0 & 1 & g\cos(\theta)\end{bmatrix}\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix}$$

(3.1.5)

Sustituyendo ahora la ecuación cinemática directa del robot diferencial en la derivada del punto descentralizado:

$$\begin{bmatrix}\dot{x}_p\\ \dot{y}_p\end{bmatrix} = \frac{1}{2b}\begin{bmatrix}b\cos(\theta) + g\sin(\theta) & b\cos(\theta) - g\sin(\theta)\\ b\sin(\theta) - g\cos(\theta) & b\sin(\theta) + g\cos(\theta)\end{bmatrix}\begin{bmatrix}v_i\\ v_d\end{bmatrix}$$

(3.1.6)

Despejando las velocidades lineales de las ruedas del robot se obtiene **el control de cinemática inversa:**

$$\begin{bmatrix} v_i \\ v_d \end{bmatrix} = \frac{1}{g} \begin{bmatrix} g\cos(\theta) + b\sin(\theta) & g\sin(\theta) - b\cos(\theta) \\ g\cos(\theta) - b\sin(\theta) & g\sin(\theta) + b\cos(\theta) \end{bmatrix} \begin{bmatrix} \dot{x}_p \\ \dot{y}_p \end{bmatrix}$$

(3.1.7)

Por último, para el cálculo de las velocidades de control se propone un control proporcional con prealimentación de la velocidad. **El control cinemático** queda:

$$\begin{bmatrix} \dot{x}_p \\ \dot{y}_p \end{bmatrix} = \begin{bmatrix} k_{r_v}\dot{x}_{ref} \\ k_{r_v}\dot{y}_{ref} \end{bmatrix} + \begin{bmatrix} k_{r_p} & 0 \\ 0 & k_{r_p} \end{bmatrix} \begin{bmatrix} x_{ref} - (x_m + g\cos(\theta)) \\ y_{ref} - (y_m + g\sin(\theta)) \end{bmatrix}$$

(3.1.8)

Para el control dinámico de la velocidad se considera el método de ajuste de reguladores, que consiste en obtener un conjunto de parámetros significativos del sistema a controlar, de forma que el regulador se obtiene de forma empírica a partir de dichos parámetros. Para obtener dichos parámetros significativos se utiliza el métodos empíricos de Ziegler-Nichols, método sencillo y clásico para determinar el valor de los parámetros del PID. Se usan dos métodos:

- Respuesta ante escalón. Se alimenta al sistema con un escalón y se obtenían los parámetros más importantes: la ganancia K y los tiempos L y T a partir de los tiempos de respuesta del sistema.
- Respuesta sostenida. Se establece un control proporcional del sistema, de forma que se va modificando la ganancia del regulador hasta que se consigue que la respuesta del sistema sea críticamente amortiguada u oscilatoria. Con esta respuesta tenemos que ver el tiempo de repetición Tc expresado en segundos y el valor de la ganancia Kc que hace que el sistema se comporte de esta forma

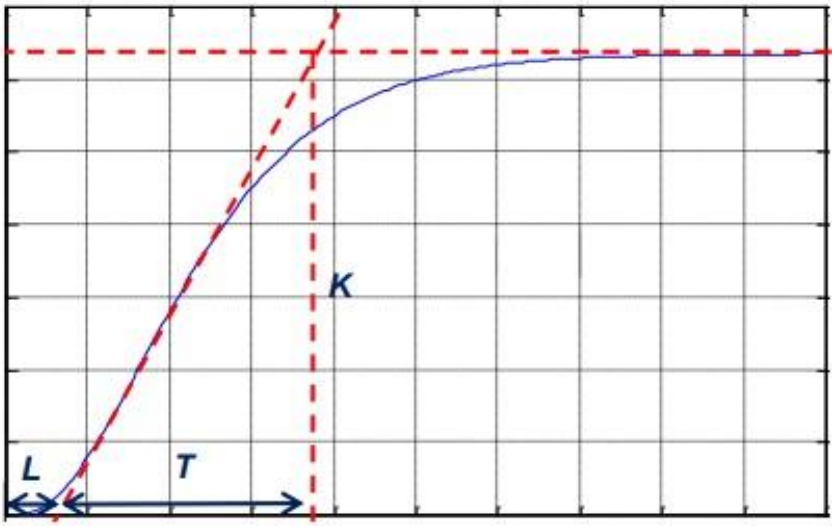


Figura 12. Parámetros significativos de Ziegler-Nichols.

Utilizando un periodo de muestreo de 0.02s, los parámetros significativos del motor en velocidad utilizando el método de la respuesta a un escalón son:

Parámetros en Velocidad	
K	0,1404
L	0,0529
T	0,0359

Para obtener los parámetros del motor en posición se utiliza el método de la respuesta sostenida. Utilizando el mismo periodo de 0.02S, éstos son:

Parámetros en Posición	
K_C	725
T_C	0,16

Así, parámetros de los reguladores Kpid, Ti y Td se podrán obtener a partir de estos valores, siguiendo la siguiente tabla:

Controlador	Parámetro	Respuesta Escalón	Respuesta Sostenida
P	K_P	$\frac{T}{K \cdot L}$	$0.5K_c$
PD	K_{PD}	$1.2\frac{T}{K \cdot L}$	$0.6K_c$
	T_d	$L/2$	$0.125T_C$
PI	K_{PI}	$0.9\frac{T}{K \cdot L}$	$0.4K_c$
	T_i	$3L$	$0.8T_C$
PID	K_{PID}	$1.2\frac{T}{K \cdot L}$	$0.6K_c$
	T_i	$2L$	$0.5T_C$
	T_d	$L/2$	$0.125T_C$

Una vez calculados los parámetros del controlador se ha ido jugando un poco con estos parámetros tratando de mejorar la respuesta, pudiendo ver los resultados en los distintos programas de control nombrados a lo largo de este trabajo. En la dirección *Laboratorio* > *RobotC* > *Pruebas PID* de la carpeta compartida puedes ver distintos archivos de programas de RobotC donde se prueba distintos reguladores proporcionales, derivados e integrales en las ruedas del robot.

Aun así, con un simple control proporcional se ha logrado un buen resultado en la mayoría de los controles, quedando la siguiente función para la acción de control, donde el error de la velocidad se multiplica por dicha constante proporcional.

$$\begin{bmatrix} acontrol_i \\ acontrol_d \end{bmatrix} = \begin{bmatrix} k_{m_p} & (w_{i_{ref}} - w_{i_{robot}}) \\ k_{m_p} & (w_{d_{ref}} - w_{d_{robot}}) \end{bmatrix} = \begin{bmatrix} k_{m_p} \times error\_vel\_ang\_i \\ k_{m_p} \times error\_vel\_ang\_d \end{bmatrix} \quad (3.1.9)$$

Con todo lo anterior dicho, el esquema resultante global que se sigue para pasar de unas variables a otras queda:

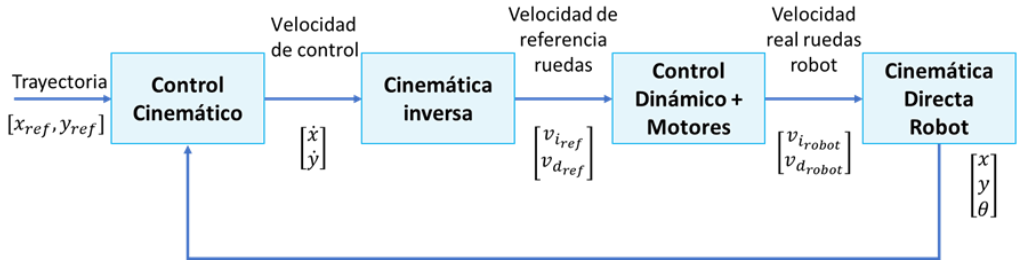


Figura 13. Esquema general de control de trayectorias.

Más adelante se implementará con los simuladores y se comparan los resultados.

3.2 Segunda tarea. Robot Limpiador.

La aplicación de esta tarea es sencilla. El robot debe limpiar las bolas de un tatami circular evitando salirse de este. Si tarda mucho se da como no logrado. Si sale del tatami se da como no logrado. Si no saca todas las bolas se da como no logrado

.

Figura 14. Esquema de la escena "Robot Limpiador".

La escena del trabajo de Robot Limpiador está formada por una tatami circular blanco con el borde negro con un diámetro de 2m. El tatami tiene 3 o más pelotas y un EV3 puesto todo de forma aleatoria (Figura 13). La idea fue que el robot de vueltas hasta encontrar una bola y en el momento que la encuentra sacarla. Si el robot da una o dos vueltas y no ve nada avanza un poco para buscar en otro sitio. En todo momento el robot debe evitar salir del tatami.

Esta aplicación requiere pues de la creación de una escena determinada además del uso de sensores, la cual nos servirá como base para explicaciones sobre las diferentes propiedades de los objetos, el diseño de escenarios y el modelado de algunas funciones en ambos simuladores.

3.3 Primera tarea extra. Robot de Braitenberg.

Como ya se ha comentado en reiteradas ocasiones, este trabajo tenía otro enfoque antes de los problemas creados por el COVID-19 que impedían realizar el acceso al laboratorio de la universidad y al robot real como consecuencia. Así pues, se intentó desarrollar el trabajo simulando en RVW, el simulador oficial de RobotC, que es la plataforma con la que se programa en ese laboratorio. Sin embargo, esto no tuvo existo pues este simulador no era capaz de simular este escenario. Debido a esto el TFG se enfocó en como poder simular los trabajos del EV3 en casa, dando solución a esto de la mejor manera posible.

Dicho esto, ese enfoque anterior tenía como base el comportamiento de Braitenberg. En la página web de Electric Bricks, se define el vehículo de Braitenberg como un robot móvil que puede moverse de forma autónoma por un entorno. Tiene sensores primitivos (capaces de medir estímulos), y ruedas (cada una manejada por su propio motor) que funcionan como actuadores o efectores. Un sensor, en su forma más simple, está directamente conectado a un efector, de manera que una señal o estímulo en dicho sensor produce una respuesta inmediata en el motor (moviendo la rueda, por ejemplo) (19).

Dependiendo de la forma en que estén conectados los sensores y motores, el vehículo mostrará distintos comportamientos. Parecerá que el vehículo intenta lograr llegar a ciertas situaciones y evadir otras, cambiando su trayectoria cuando la situación cambia. Podemos definir así dos comportamientos: temeroso y explorador. El temeroso, cuando el sensor detecta luz, la criatura huye de ella y aumenta la velocidad, de lo contrario disminuye su velocidad pues se siente más tranquilo, dando la sensación de que el robot huye de la luz ya que no le gusta, y busca sombra (20). El robot explorador sería lo contrario, dirigiéndose a la luz.

En esta tarea se busca el robot temeroso. Esto se llevaría a cabo con varios sensores a la izquierda y el mismo número a la derecha que estimulan las ruedas de sus respectivos lados, siguiendo la siguientes regla.

- Más luz en la derecha → más velocidad en la rueda derecha → robot gira para la izquierda.
- Más luz en la izquierda → más velocidad en la rueda izquierda → robot gira para la derecha

Dichos sensores que se colocan en cada lado del robot están ponderados, dando más importancia si detectan algo de frente para esquivarlo rápido y menos importancia cuanto más laterales o atrás estén.

Si se quisiera hacer el robot explorador solo habría que cruzar las conexiones de sensores y ruedas, como en la figura 15.

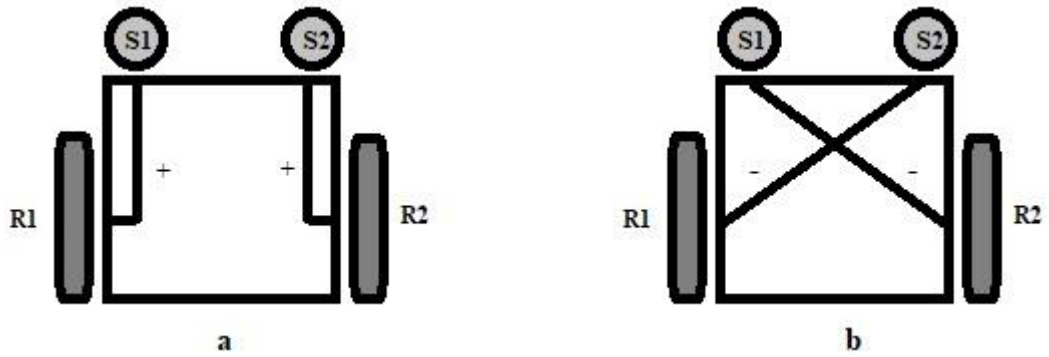


Figura 15. Configuración del robot diferencial de Braitenberg en modo a) temeroso / b) explorador (20).

En este trabajo el comportamiento se comprueba con luces, sin embargo, podría emplearse de otras maneras. Si se usaran en su vez los sensores ultrasónicos en vez de los de color, detectando así obstáculos, podría obtenerse una buena solución para la evitación de obstáculos que seguiría el mismo razonamiento.

3.4 Segunda tarea extra. Seguimiento entre paredes.

La segunda de las tareas extra que solo se resuelven en CoppeliaSim es un sencillo seguimiento de paredes. El robot sigue una trayectoria entre dos paredes por muy irregular que sea ésta, deteniéndose y disminuyendo la velocidad si se topa de frente con esta. Para su desarrollo, como en el robot de Braitenberg, se necesita una configuración diferente del robot añadiéndole sensores, cosa que no puede conseguirse en RVW. Para el escenario se ha elegido una simple espiral donde el robot tendrá que salir de ella sin tocar la pared.

El razonamiento es sencillo, basado en uno de los video-apuntes del profesor de la Universidad Politécnica de Valencia, Leopoldo Armesto (39). Se parte de una velocidad máxima impuesta que disminuye a medida que el robot se va acercando a una pared de manera inversamente proporcional. Para ello se define una distancia máxima de detección a partir del cual se va frenando y un distancia mínima a partir del cual el robot ya debería estar parado.

$$v = v_{max} \times \frac{d}{d_{max}}$$

(3.4.1)

De esta forma la velocidad máxima se va multiplicando por d/d_{max}, siendo el rango de *d* $d \in [d_{min}, d_{max}]$ y $d=0$ para $d<d_{min}$, siendo entonces el máximo valor de $v=v_{max}*1=v_{max}$, y el mínimo valor 0, quedando $v \in 0 \cup [v_{max} * d_{min}/d_{max}, v_{max}]$.

Por otro lado, para el seguimiento entre paredes se dispone al robot de dos sensores ultrasónicos de Mindstorms en cada lado midiendo distancias como en la figura 16.

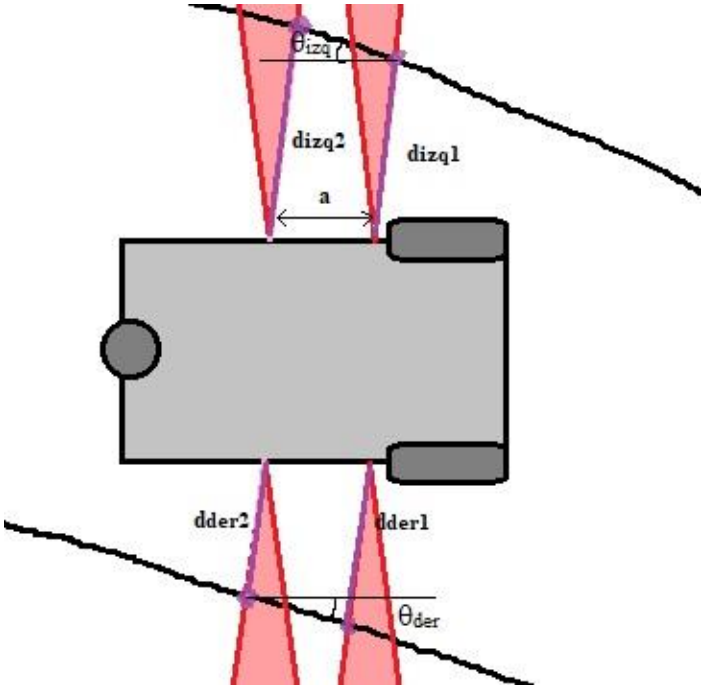


Figura 16. Configuración de los sensores del EV3 para el seguimiento de paredes.

Así pues, siguiendo este esquema se puede deducir el error angular de la orientación en ambos lados y las distancias medias:

$$\theta_{izq} = \text{atan}\left(\frac{d_{izq_2} - d_{izq_1}}{a}\right) \qquad d_{izq} = \frac{d_{izq_1} + d_{izq_2}}{2}$$

(3.4.2) y (3.4.3)

$$\theta_{der} = \text{atan}\left(\frac{d_{der_1} - d_{der_2}}{a}\right) \qquad d_{der} = \frac{d_{der_1} + d_{der_2}}{2}$$

(3.4.4) y (3.4.5)

Quedando así un error de orientación y error de posición, respectivamente, como:

$$\theta = \frac{\theta_{izq} + \theta_{der}}{2} \qquad d = K(d_{der} - d_{izq})$$

(3.4.6)

Donde K es una constante proporcional multiplicando la diferencia de distancias. En el trabajo se ha usado simplemente K=1, ya que esta funcionaba bien.

Sumando estos dos errores se obtiene el error de dirección de la trayectoria que el robot debería tener, como se muestra en la figura 17.

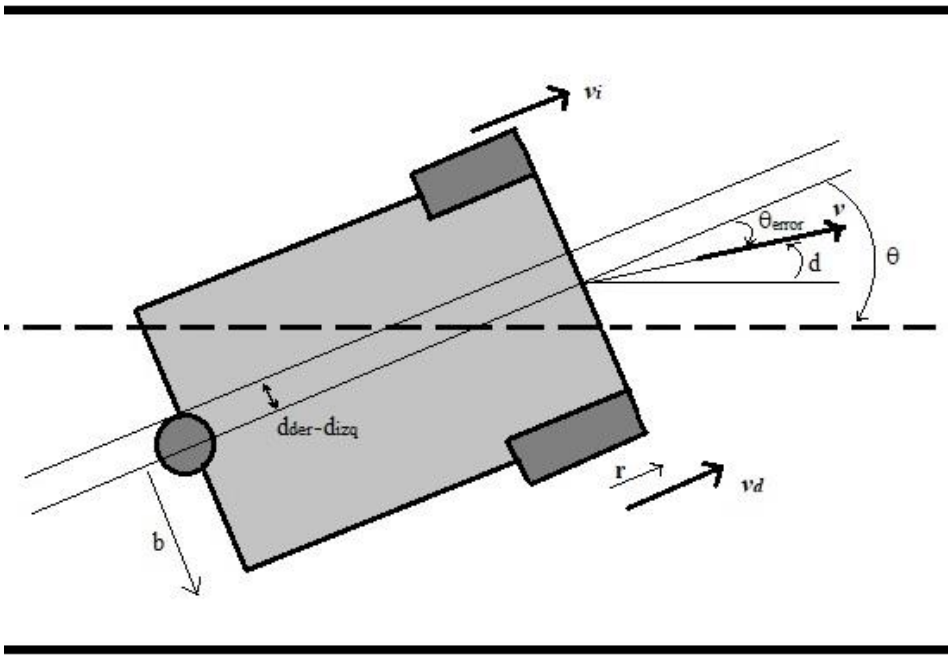


Figura 17. Errores de orientación y posición del robot diferencial en seguimiento entre paredes.

$$\theta_{error} = d + \theta$$

(3.4.7)

Quedando las velocidades lineales de cada rueda como:

$$v_{izq} = v\left(\cos(\theta_{error}) + \frac{b}{2r}\sin(\theta_{error})\right) \qquad v_{der} = v\left(\cos(\theta_{error}) - \frac{b}{2r}\sin(\theta_{error})\right)$$

(3.4.8) y (3.4.9)

Transformándolas en angulares dividendo por el radio de la rueda (0.028 m).

$$w_i = \frac{v_i}{r}$$

(3.4.10)

4. SIMULACIÓN EN RVW.

4.1 Creación de escenario en RVW Level Builder.

RVW no permite la creación de una escena idéntica a la propuesta. A continuación, se explicará los problemas por los cuales el simulador no puede trazar exactamente la misma escena de las prácticas y se diseñará un escenario lo más similar posible.

Para abrir el simulador de *RVW Level Builder* basta con compilar y ejecutar un programa en la pestaña *Robot > Compile and download program* o simplemente abrirlo en *Window > RVW Level Builder utility*. Con esta última forma no podrás ejecutar la simulación si previamente no has compilado el programa.

Una vez abierto, se entra en el modo *Build* y se comienza a crear el tatami. En *3DModels* se puede encontrar objetos proporcionados por el simulador con forma de cubos, cilindros, esferas y muros. En *Tiles* se encuentran losas para trazar el suelo (*line tracking*) y para marcar los puntos de salida o parada del robot (*start/stop*). A pesar de la variedad de líneas en *line tracking*, ninguna de estas sirve para crear un círculo de 2m, por lo que tendremos que importar nuestras losas con el menú *IMPORT TILE*. El problema que se presenta es que el simulador no deja poner un obstáculo u objeto encima de una losa, como se puede comprobarse. Por este motivo hemos tenido que dividir el circulo en porciones de losa para conseguir su perímetro sin ocupar el centro y así poder posicionar las pelotas y el robot en él (figura 18).

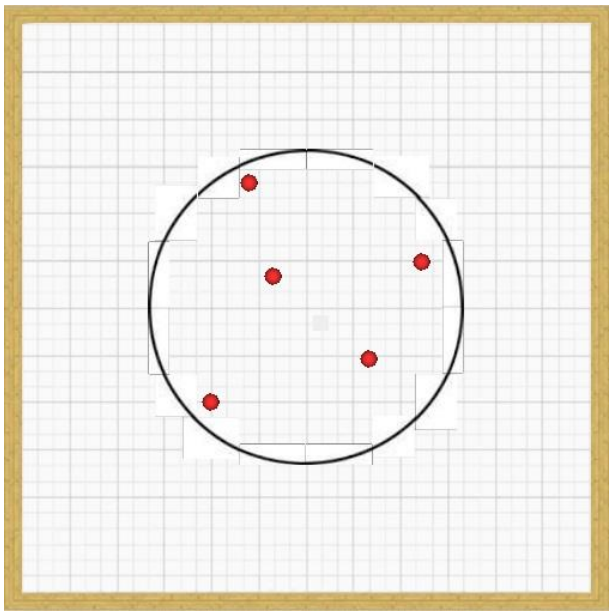


Figura 18. Escena de RVW para la simulación del programa "Robot Limpiador".

Para importar las losas que constituirán la circunferencia primero se ha descargado una foto cualquiera de una circunferencia de internet como la de la dirección *Escenas > RVW Level Builder > Imágenes Circulo*, se ha abierto en Paint, se ha activado la cuadrícula para que ayude como referencia a la hora de recortarla y se ha recortado como en la figura 19.

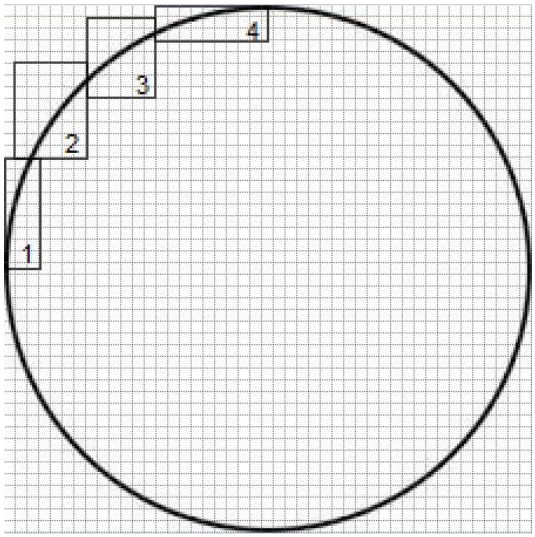


Figura 19. Rectángulos de referencia para recortar en Paint la circunferencia empleada para el Robot Limpiador

Después, contando los cuadrados y sabiendo que la circunferencia tiene 1m de rayo se ha hecho un cálculo simple para obtener la distancia equivalente de esas secciones. Por ejemplo, la sección 1 tiene 3 cuadrados de ancho y 9.5 cuadrados de largo. Si 22.5 son la distancia en cuadrados de radio (1m), entonces con una regla de tres, 9.5 cuadrados equivalen a 0.4222m y 3 cuadrados equivalen a 0.1333m. Continuando con las demás secciones queda:

SECCIÓN	X (cuadros)	Y (cuadros)	X (m)	Y (m)
1	3	9.5	0.13333	0.42222
2	6	8	0.26667	0.35556
3	6	6	0.26667	0.26667
4	9.5	3	0.42222	0.13333

A continuación, en RVW Level Builder se abre IMPORT TILE y se sube cada sección definiendo la altura y la anchura que se ha calculado y la carpeta donde se quiere que aparezca (figura 20).

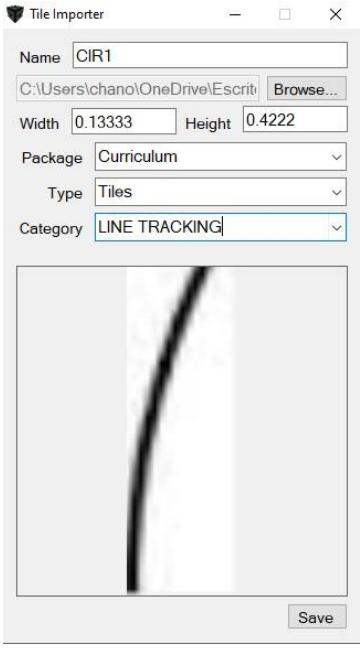


Figura 20. Pestaña de IMPORT TILE.

Ahora solo quedaría montarlo con cuidado para que quede la circunferencia como la de la figura 18.

Ahora se tienen que elegir las pelotas que vamos a usar y colocarlas en el tatami de forma aleatoria. En el trabajo se ha usado la 5ª esfera de color rojo ya proporcionada por RVW en *3DModels > Spheres*, pues es bastante similar a las reales usadas en la práctica, pero si se prefiere cambiar alguna propiedad o algo más personalizado simplemente bastaría con importarlo con IMPORT OBJECT (21). El menú de importar modelos es como el de la figura 21.

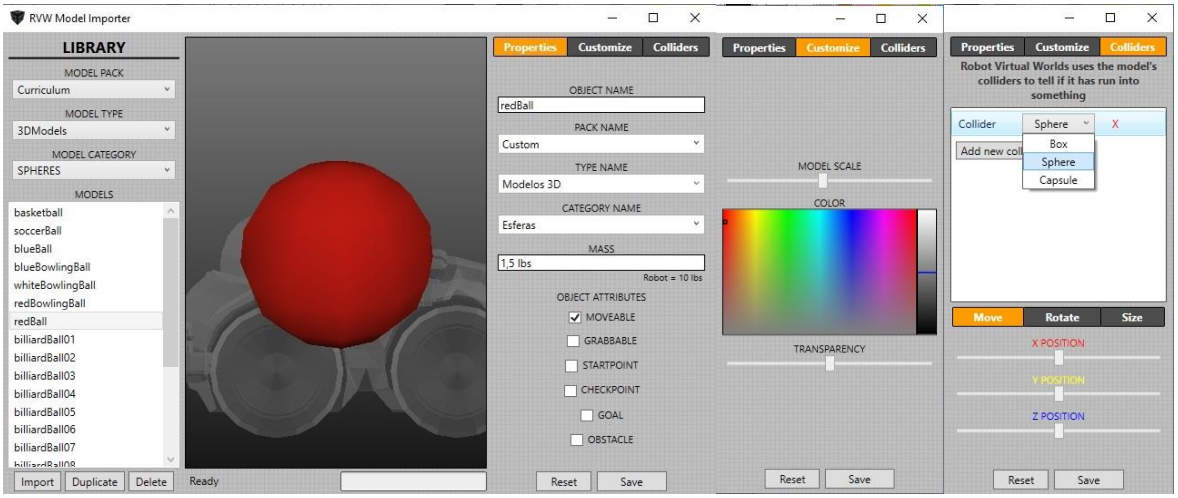


Figura 21. Menú IMPORT OBJECT del RVW Level Builder.

A la izquierda se observa la librería de modelos de RVW Level Builder donde se puede coger algún modelo como base. La parte derecha se divide entre los siguientes menús:

- En **Properties** se elige o se crea la carpeta donde se guardará el nuevo modelo y se le asignan propiedades o atributos.
    - En la opción **MASS** se pone el peso del modelo.
    - **MOVEABLE** habilita o deshabilita que al objeto le afecten las fuerzas y pueda ser movido.
    - **GRABBABLE** permite al objeto ser recogido por el robot.
    - **STARTPOINT** le permite la opción de poder ser punto de partida.
    - **CHECKPOINT** convierte al objeto en un punto de paso den la ruta hacia la meta.
    - **GOAL** hace al objeto punto final de la simulación.
    - **OBSTACLE** vuelve al modelo en un obstáculo de manera que si el robot choca con el termina la simulación.
- El menú de **Costumize** permite personalizar el modelo cambiándole el tamaño comparándolo con el tamaño de un robot, eligiendo entre una gama de colores o cambiando el nivel de opacidad.
- En **Colliders** se puede volver al objeto colisionar cuando entra en contacto con otro modelo, reaccionar a el choque. La forma pura de colisión se puede modificar con las opciones de este menú.

Por último, queda poner también aleatoriamente la salida del robot en la simulación. Para ello, se abre *IMPORT TILES* y se importa cualquier imagen en blanco para no interferir con el simulador de color del robot. Una vez hecho esto, se abre *IMPORT OBJECT* para buscar

la imagen blanca anteriormente importada y volver a importarla con el atributo de punto de salida. Esto se hace así debido a que el importador de losas no te da la opción de ponerles tales atributos, y si directamente se abre el importador de modelos y se escoge una losa de salida y se le cambia da problemas y no carga bien el color.

Con todo esto hecho, el escenario debería quedar como el de "Robot Limpiador" de la dirección *Escenas > RVW Level Builder*.

Si se quisiera importar algún modelo 3D distinto creado por un programa externo, solo se tendría que dar a *Import* dentro del menú de importación de modelos. Este simulador solo admite archivos STL Files (.stl) que pueden venir de programas como Autodesk Inventor o SolidWorks.

4.2 Programación y simulación.

La programación de RobotC es una programación bastante sencilla basada en C. Usa la mayoría de comandos lógicos de C. Los comandos específicos de RobotC por su uso destinado a la programación de un robot, como comandos de motores o sensores, se puede encontrar en el navegador de funciones *Text Functions* que se podrá ver a la izquierda. Se recomienda aumentar las funciones de este navegador cambiando el nivel de usuario en *Windows > Menu level* a *Expert* o *Super User*. También convendría asegurarse que se está programando para un EV3 en *Robot > Platform Type*.

Para poder usar el simulador, tenemos que cambiar el objetivo de la compilación de robot real a virtual en *Robot > Compiler Target* y señalar *Virtual Worlds*. Ahora el simulador se abrirá cada vez que se compilen los programas en *Robot > Compile and Download Program*. Por último, para seleccionar que tipo de mundo virtual quieres que se abra entre los descargados se accede a *Windows > Select Virtual World to use* y, como se ha dicho anteriormente, se va a centrar en *RVW Level Builder.*

Por último, cabe destacar que, una vez compilado el programa, en *Robot > Debugger Windows* puedes abrir distintas ventanas que pueden mostrar cosas de ayuda, por ejemplo, ventanas de mensajes, el contenido de variables o lo que mostraría por pantalla el EV3.

A continuación, se muestra el resultado de la simulación por RVW de las aplicaciones propuestas.

4.2.1 Control cinemático de trayectoria.

El archivo del programa utilizado para la programación de este control se puede encontrar en la dirección *Programas > RobotC.* Además, se puede ver un video con el resultado de esta simulación en la carpeta *Videos* con el nombre de “RVW Control Trayectoria Circular”.

El programa es bastante similar al empleado para el robot real ya que ambos están hechos desde RobotC y la finalidad es la misma. La principal diferencia es que el simulador no deja interactuar con los botones, anulando el fichero “gen_trayect.h” (ANEXO I). Tampoco funciona el comando “datalogOpen ()”, pero es sustituido por el *Datalog* o el *Debug stream* que tiene interno RobotC si opera en modo en virtual, los cuales se pueden acceder desde *Robot > Debugger Windows* y, seguido, se copia los resultados a un fichero interno para poder abrirlo con MATLAB y generar las gráficas.

Con los resultados obtenidos se han realizado las siguientes gráficas:

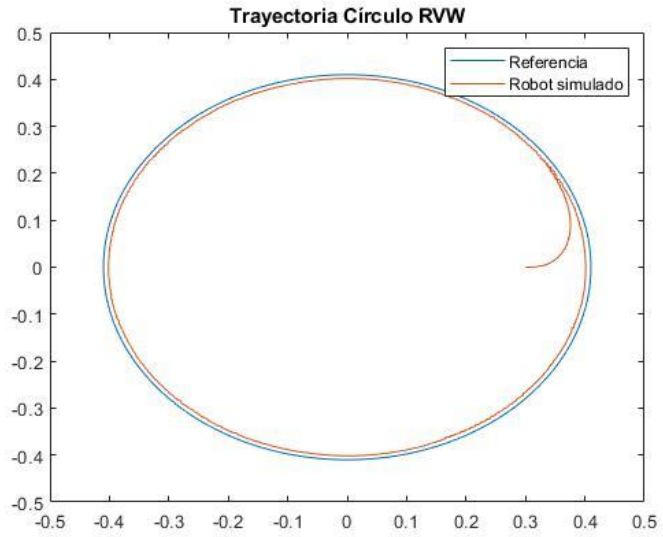


Figura 22. Gráfica resultado de la trayectoria circular en RVW.

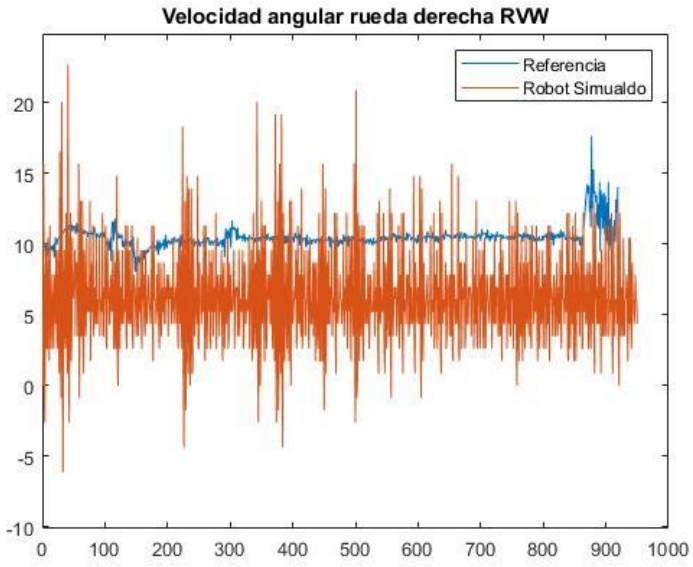


Figura 23. Gráfica de velocidad angular de rueda derecha en RVW

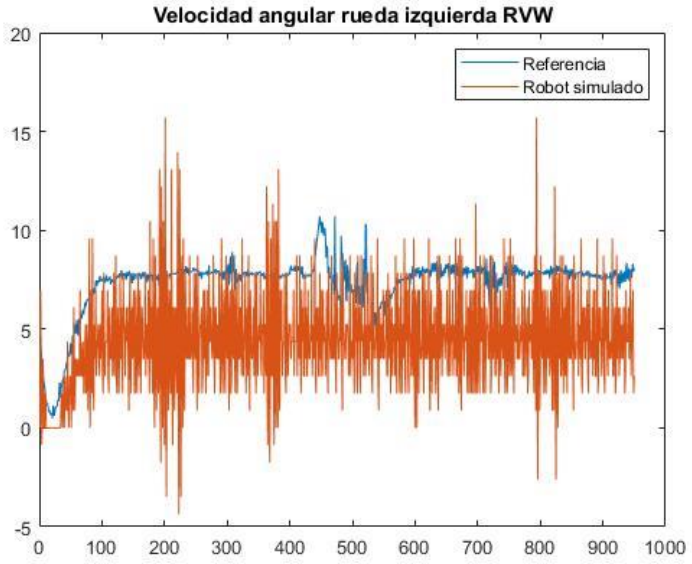


Figura 24. Gráfica de velocidad angular rueda izquierda en RVW.

Si se desea mirar las gráficas de velocidades angulares experimentales del robot real, pueden verse en el ANEXO I.

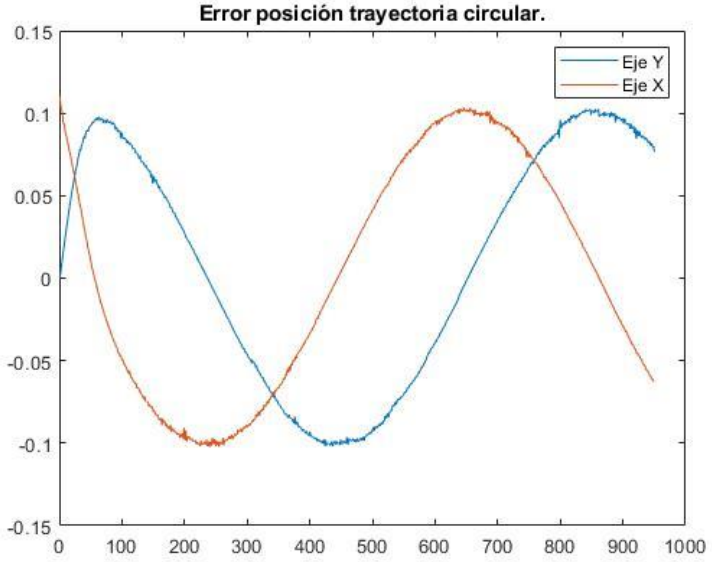


Figura 25. Gráfica del error de posición de los distintos ejes en RVW.

Se observa que las gráficas de posición son prácticamente idénticas que las obtenidas por el robot real en el laboratorio, lo que tiene sentido pues se usa prácticamente el mismo algoritmo de programación y la misma plataforma de programación. La trayectoria se lleva a cabo perfectamente, pudiéndose ver el error en los distintos ejes respecto a la referencia en la figura 25.

En la gráfica de la figura 26 se puede observar la función del error cuadrático que generaba el robot simulado, dando como resultado un Error Cuadrático Medio = 0.1004.

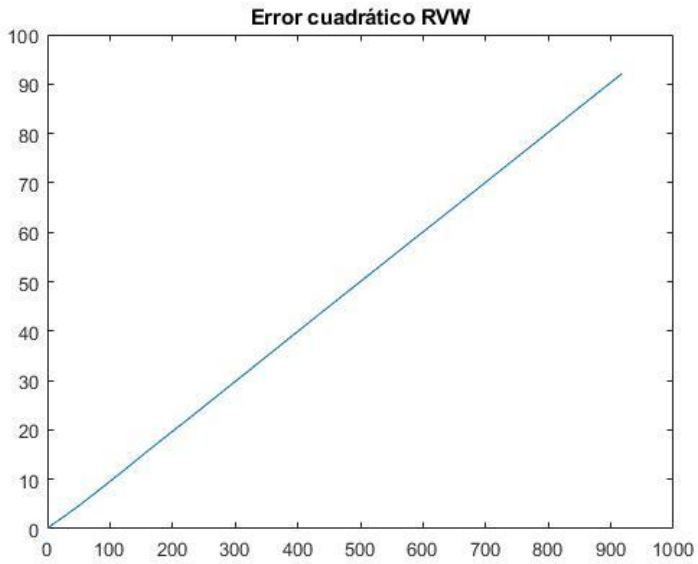


Figura 26. Gráfica del Error Cuadrático de la trayectoria en RVW.

4.2.2 Programa "Robot Limpiador".

Una vez creado el escenario, se ha desarrollado el programa de la aplicación de Robot Limpiador. El algoritmo del programa puede encontrarse en la dirección *Programas > RobotC > Robot Limpiador*. Para la ejecución del programa se puede cambiar la posición de las pelotas o del robot como se desee. También se puede añadir más pelotas.

En la carpeta *Videos* se puede encontrar un video llamado "RVW Robot Limpiador" en el que se ha grabado la simulación del resultado del programa. El robot saca las pelotas y cuando no encuentra ninguna da una vuelta y avanza a otro lugar sin salirse perfectamente.

El resultado es bastante parecido a un comportamiento real y el programa es bastante intuitivo. Se puede deducir que el comportamiento de los sensores y motores del robot es muy similar al del robot real, como era de esperar ya que esta versión del simulador tiene como finalidad únicamente la simulación del robot de Lego, sin dejar opción a otro robot. Sin embargo, más allá de una sencilla aplicación este simulador no puede simular, ya que no permite ni modelar el robot, ni usar más de un robot en la misma aplicación, ni la creación de escenarios excesivamente complejos.

Con lo cual se concluye que para tareas, ejercicios o programas sencillos es un fluido simulador con características, resultados técnicos en él y simulaciones bastante reales. Sin embargo, su utilidad reside en esto programas sencillos viéndose imposible algo complejo.

5. SIMULACIÓN EN COPPELISIM.

5.1 Diseño del modelo EV3 en LeoCAD.

Para la creación del modelo en LeoCAD, simplemente hay que ir arrastrando y colocando piezas según los manuales de instrucciones de montaje que se puede encontrar en la dirección *Modelos > Manual Montaje EV3* dentro de la carpeta compartida, o en la referencia (16). Se recomienda desactivar *Move Snap Enable* en la opción *Movement Snap* en la barra de herramientas (). Desactivar esta opción permite cambiar el desplazamiento de las piezas a uno más libre y menos cuadriculado. De lo contrario muchas piezas no cuadran bien. *Rotation Snap Enables* () conviene desactivarlo también para el montaje de algunas piezas, aunque para otras conviene tenerlo para poder hacer giros exactos de 90º o 180º.

Como ayuda para aligerar y facilitar la creación, se puede usar el buscador de piezas y buscar el nombre de la pieza en Brickset (22) o apoyarse en el video de YouTube de Alberto Martín Domínguez (23). También hay una lista de piezas en la página 63 de la referencia (24). La mayoría de las piezas están en la categoría *Technic*. Al final deberá quedar algo como en la figura 26.

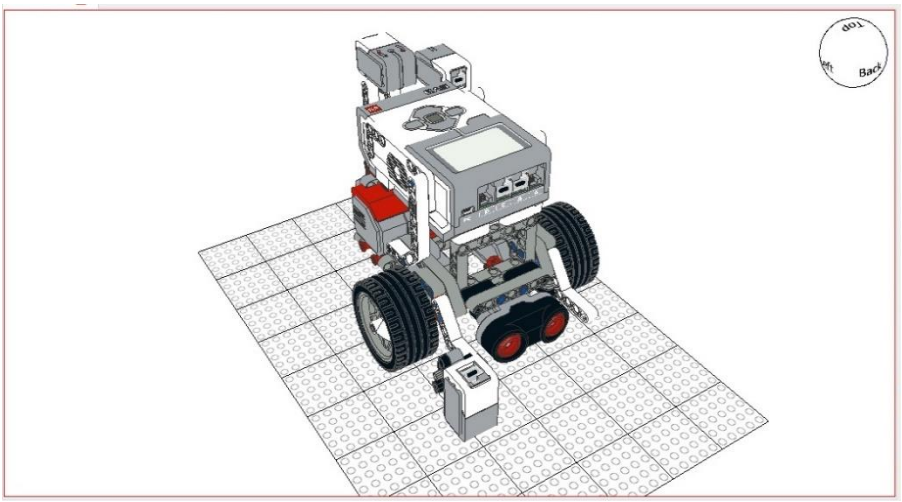


Figura 27. Modelo EV3 en LeoCAD

Una vez terminado, se exporta con *File > Export > Wavefront* en OBJ File (.obj), ya que así se puede importar después a CoppeliaSim. También convendrá guardarlo con "*Save as...*" para posibles modificaciones. En la dirección *Modelos > LeoCAD* de la carpeta compartida hay un modelo del EV3 ya exportado. También se puede encontrar modelos del EV3 sin sensores, para usarlo de partida, en la misma dirección.

5.2 Importación y modelado en CoppeliaSim.

5.2.1 Importación.

Con CoppeliaSim abierto, para importar el modelo creado por LeoCAD o cualquier otro archivo de modelo válido, se debe ir a *File > Import > Mesh*. El programa admite modelos 3D de archivos "*.obj"*, "*.dxf"*, "*.ply"*, "*.stl*" y "*.dae"*. Una vez se le da a importar el modelo de EV3 aparecerá una pestaña en la que le damos a *Import*.

La biblioteca de LeoCAD es un archivo de la biblioteca de LDraw, es decir, usa las piezas de LDraw. Las piezas de LDraw, según el apartado de documentación de su página web (25), se miden en LDU (LDraw Unit). Un LDU es igual a 0.4 mm. Esa es la razón por la que se tiene que cambiar la escala de todo modelo importado diseñado con este programa. Para ello se selecciona manteniendo la tecla Shift todas las formas recién importadas que configuran el robot y se accede al menú *Tools > Scene object properties* de la barra de herramientas. En la pestaña *common*, donde se configuran las propiedades comunes a la escena general, se entra en *Scaling* y se escribe la nueva escala de 0.4 y se le da a *OK*.

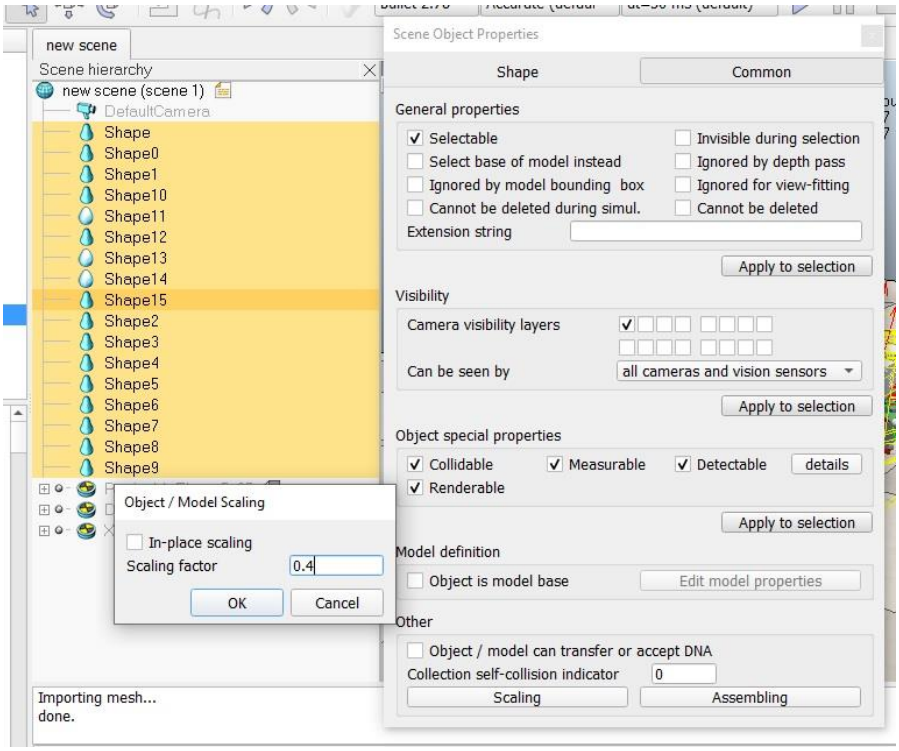


Figura 28. Pestaña "Scene Object Properties" y "Object / Model Scalling"

En el robot que se ha importado también se ha realizado el siguiente cambio puramente por comodidad visual. En el mismo menú *Scene Object Properties*, en *Shape*, se ha activado la opción *Show edges with angle* con 30.0 deg. Así se visualiza mejor los contornos del robot.

El robot debería quedar como el de la figura 29.

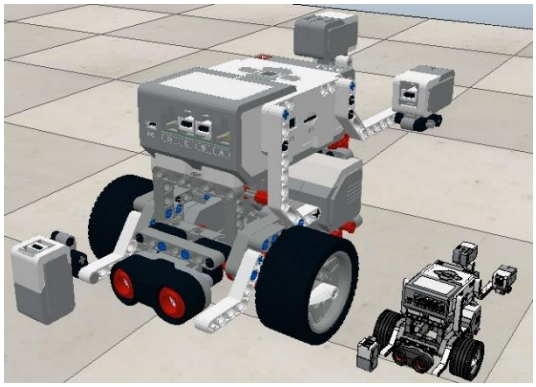


Figura 29. EV3 antes y después de los cambios tras su importación.

5.2.2 Geometrías puras.

Para describir el comportamiento dinámico del robot, se deben de crear unas geometrías puras y más sencillas. Adjudicarles estas propiedades físicas a las geometrías diseñadas externamente a CoppeliaSim puede dar problemas de fluidez y de renderización. La idea es crear un objeto dinámico simple por cada una de las piezas del robot, o al menos, aquellas más exteriores que puedan estar en contacto con otros objetos y/o crear colisiones. Se recomienda por comodidad hacer solo visible la parte del robot que se le quiera crear su forma pura simplemente combinando entre capas visibles y no visibles.

Se selecciona la parte que se le quiere crear la forma pura y vamos al menú *Common* de *Scene Object Properties* desde *Tools* o clickando dos veces en el icono de la pieza en el árbol de la jerarquía de escena. Se activa, por ejemplo, la segunda capa de la primera fila y accediendo *Layers* ,bien desde *Tools* o bien desde el icono la barra de herramientas vertical (), se desactiva la vista de la primera capa de la misma fila (figura 30). Como todas las piezas menos la que se acaba de cambiar tienden a estar solamente en esta capa, solo quedará visible la que se quiere editar en ese momento. El modo de uso de las capas es tan sencillo como eso, cada pieza activa la capa a la que quiere pertenecer y en *Layers* se activan las que se quiere que sean visibles. Las formas complejas tienden a pertenecer a la primera fila de capas y las geometrías puras a ocultarse en la segunda.

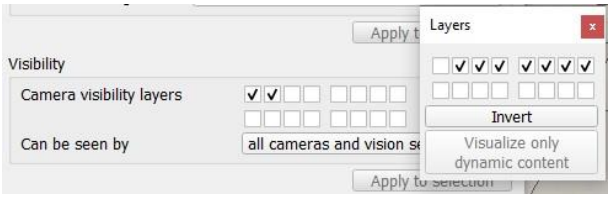


Figura 30. Menú Layers de CoppeliaSim.

Si se selecciona alguna forma de nuestro robot y se accede a la opción *Toggle shape edit mode* del menú de herramientas vertical ( ) se puede observar que ahora la geometría seleccionada está compuesta por una serie de triángulos y que se abre el menú *Shape Edition*. Se va a crear la geometría pura a partir del modo de edición de triángulos seleccionando los triángulos que se quiere que sirvan como referencia para la creación de la nueva geometría. Para la selección de los triángulos se recomienda seleccionar varios dentro de la escena manteniendo la tecla shift (Figura 31). Los triángulos no seleccionados son de color azul y los seleccionados se vuelven amarillos.

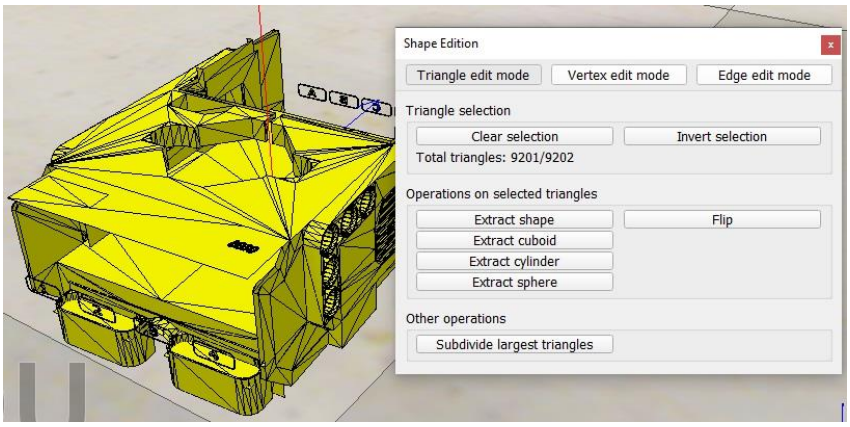


Figura 31. Modo de edición de geometrías puras en CoppeliaSim.

Por último, se extrae la o las geometrías puras que se deseen por cada pieza. La mayoría de las formas simples se extraen como cuboides, pero algunas piezas como las ruedas laterales o la rueda loca se han utilizado cilindros y esferas con su mismo radio. Otras piezas quizá necesiten dos o tres geometrías puras como la pieza *Technic Beam 3x 3.8x 7 Liftarm Bent 45 Double* colocada a los laterales del robot con forma de L. El resultado final deberá ser algo similar al de la figura 32.

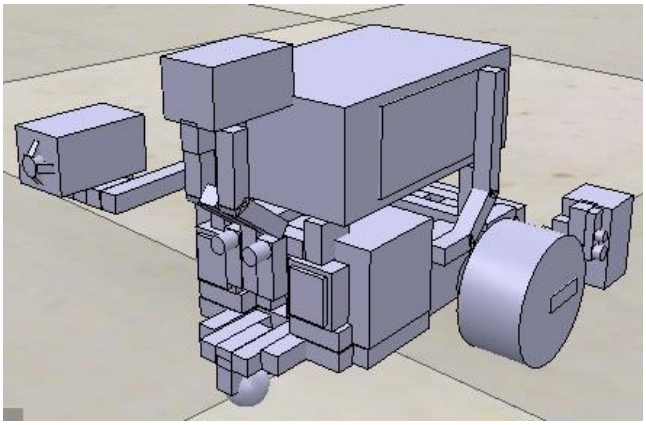


Figura 32. Geometrías puras del robot EV3.

5.2.3 Propiedades del objeto.

Un objeto puede ser dinámicos o estáticos y reactivos o no reactivos. La diferencia entre dinámicos o estáticos es si se ven afectados por fuerzas, como la gravedad, o no. Por otro lado, los reactivos producen una fuerza de reacción sobre otro objeto también reactivo, al contrario de los no reactivos. Con lo cual se pueden dar las siguientes posibilidades en la relación entre dos objetos:

- L**os dos son estáticos independientemente de su reactividad.** Ninguno de los dos se moverá y por lo tanto tampoco influirá al otro.
- **Si uno de los objetos es dinámico y hay un objeto no reactivo.** El objeto dinámico podrá moverse, pero debido a su no reactividad traspasará al otro objeto. Cuidado porque el suelo es un objeto reactivo que también podría ser atravesado.
- **Los dos objetos son reactivos y al menos uno de ellos es dinámico.** Existirá colisión basada en la Tercera Ley de Newton de acción y reacción.

Para modificar estas propiedades se tiene que abrir el menú *Show Dynamic Properties Dialog* dentro de Shape *Object Properties/Shape*. Se tendrán que activar en las geometrías puras creadas para el EV3. Aquí también se puede cambiar propiedades como la masa o la inercia.

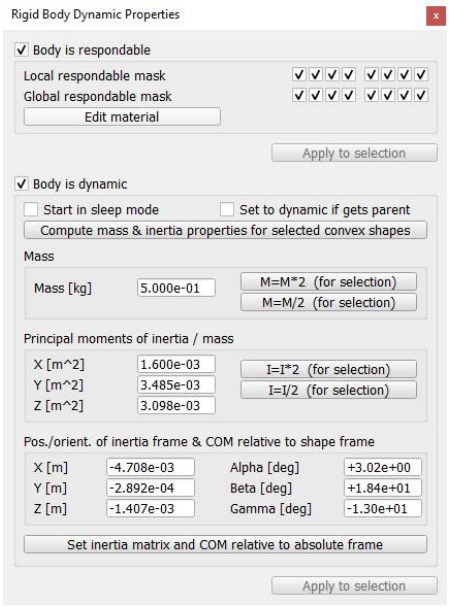


Figura 33. Menú de propiedades dinámicas de CoppeliaSim.

En este menú también se puede observar que con la reactividad también hay una especies de capas. Eso quiere decir que dos objetos reaccionarán entre sí solo si tienen la misma capa activa. De este modo si dos objetos puros pertenecientes al mismo robot chocaran entre sí,

como por ejemplo por algún tipo de apriete, e impidiera el funcionamiento del mismo, podría anularse la reactividad entre ambos.

Otras propiedades que se deben tener en cuenta están en *Object special properties*, la cuales sirven para saber si el objeto es detectable o se tiene en cuenta. Estas propiedades son:

- **Collidable**. Solo si es reactivo. Si está activo el objeto se tendrá en cuenta para la realización de cálculos para posibles colisiones.
- **Measurable**. Si esta activo el objeto es usado para el cálculo de mínima distancia, función del programa.
- **Detectable**. Si está activo puede ser detectado por sensores de proximidad. Si además se abre la ventana *details* que está a la derecha, se puede elegir a que sensores de proximidad se quiere que el objeto sea detectable, algo bastante útil como se comenta más adelante (Figura 34).
- **Renderable**. Si está activo puede ser visible para sensores de visión.

Estas propiedades que se acaban de definir no son de gran importancia para el modelado del robot. Se usarán para objetos u obstáculos en la creación de escenarios.

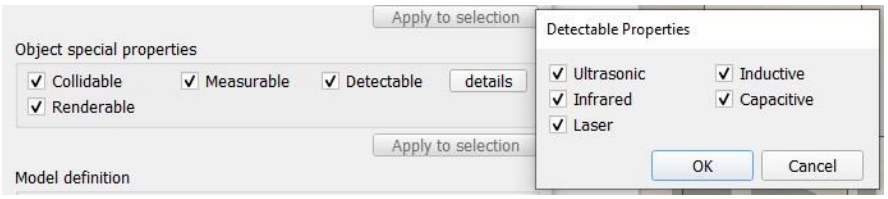


Figura 34. Propiedades especiales de los objetos en CoppeliaSim.

Otras propiedades que se pueden ajustar serían el color o la textura en el menú de propiedades del objeto.

5.2.4 Articulaciones y motores.

Si se comenzara la simulación en este punto el robot se desmontaría. Aún se tiene que formar una jerarquía de piezas para fijar unas con otras o, en caso de que se quiera añadir algún grado de libertar entre piezas, añadirle articulaciones.

Para crear una articulación solo hay que seleccionar una opción dentro de *Add* > *Joint*. Hay tres tipos de articulaciones: de revolución, prismática o esférica. Las dos primeras tienen un grado de libertar, la primera permite un desplazamiento angular con centro en su eje y la segunda un desplazamiento axial a lo largo de su eje. La articulación esférica añade 3 grados de libertad. Para fijar la articulación entre dos partes se tendría que crear una jerarquía entre ellos. La creación de jerarquía se verá en el siguiente apartado.

Las articulaciones pueden ser pasivas, es decir que no se controlan directamente, o dinámicas, simulando así un motor. El modo de las dinámicas se divide en:

- **Par/Fuerza.** Se controlan a través de físicas. Pueden estar actuadas por un motor o tener un movimiento libre. Los valores de referencia se modifican por programación. Tendrá un movimiento libre si el motor esta desactivado. En el caso de usar un motor, puede ajustarse la velocidad de este si no se usa ningún tipo de control. Por otro lado, si activamos un control, este puede ser de bucle cerrado PID o de impedancia (para articulaciones elásticas). Si el par máximo es muy elevado la velocidad referente se alcanza casi al instante, por el contrario, se realizaría un control de fuerza hasta alcanzarla.
- **Cinemática inversa.** Una articulación pasiva pero que su posición puede ser controlada por un módulo de cinemática inversa.
- **Modo híbrido.** Un modo par/fuerza, pero cuya posición de referencia la establece el módulo de cinemática inversa.
- **Dependiente.** Su posición depende de otras articulaciones.

Se debe tener en cuenta que un objeto dinámico siempre conviene que sus articulaciones estén en modo par/fuerza o híbrido, por otro lado, si el objeto es estático, no debe usarse ninguno de los dos.

Ahora se procede a posicionar las articulaciones en las ruedas laterales del EV3. Se añaden dos articulaciones de revolución. El siguiente paso es colocar las articulaciones coaxiales con los ejes de las geometrías puras de las ruedas. Para ello primero se selecciona las articulaciones y seguido, con el shift, las geometrías puras de la rueda. Se abre el menú *Object/Item traslation/position* en la barra de herramientas () y en *position* se selecciona *Apply to selection*. Las articulaciones deberían ahora estar en la misma posición que la geometría pura.

Figura 36. Menú Object/Item Translation/Position

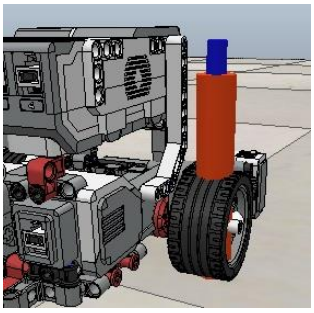


Figura 35. Articulación de revolución ajustada a la posición de la geometría pura de la rueda.

Para terminar de colocarlo ya solo falta girarlo. Para ello se accede al menú *Object/Item Rotation/Orientation* () y se gira las articulaciones -90º respecto al eje Y de la escena (figura 38). El signo negativo se debe a que las articulaciones giran en sentido antihorario.

También se le puede cambiar la longitud o el diámetro de la articulación en *Scene Object Properties*, pero estos cambios serían puramente estéticos. Finalmente, las articulaciones de las ruedas laterales quedan como en la figura 37.

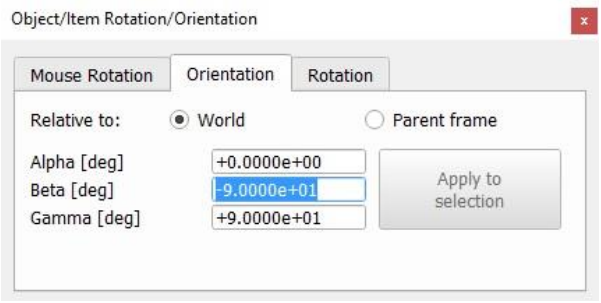


Figura 38. Menú Object/Item Rotation/Orientation.

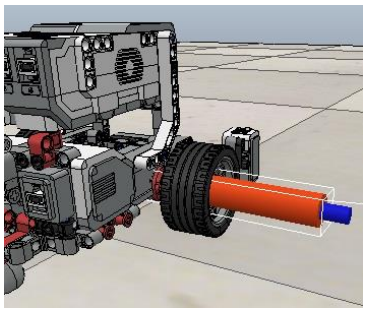


Figura 37. Articulación de revolución coaxial a la rueda.

Ahora faltan las propiedades dinámicas de las articulaciones. Lo primero que se debe tener en cuenta es que, consultando especificaciones técnicas del EV3, los motores funcionan a 160-170 rpm, con un torque de rotación de 20 Ncm y un torque de rotor bloqueado de 40 Ncm con un sensor de rotación que tiene una resolución de 1 grado (24).

Se abre *Scene Object Properties*. Como se quiere ajustar la velocidad por scripts, se pone en modo *Par/Fuerza* y no se requiere ningún tipo de control. En el menú de propiedades dinámicas se habilita el motor y se pone el par máximo tal como en la figura 39. No se activa el control.

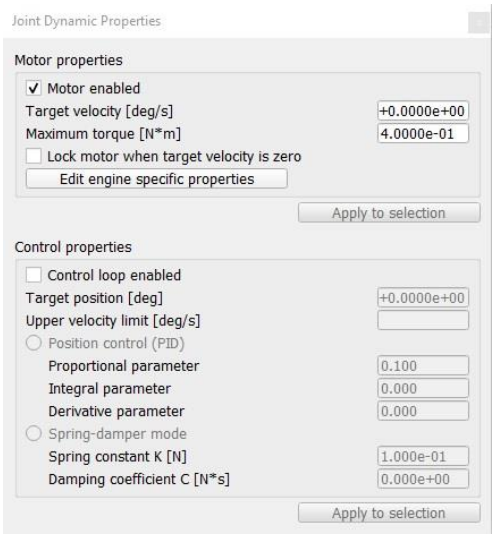


Figura 39. Menú de propiedades dinámicas de las articulaciones.

Para la rueda loca se procede de la misma manera, creando esta vez una articulación esférica, posicionándola concéntrica a la geometría pura de la rueda loca. Esta articulación debe estar

en modo Par/Fuerza pero teniendo el motor inhabilitado, dejándose llevar así por las ruedas laterales.

5.2.5 Sensores.

Si se sitúa en el menú de opciones *add* se puede comprobar que se dispone de 3 tipos de sensores: sensores de proximidad, de visión y de fuerza.

- **Sensor de proximidad**. Mide la mínima distancia desde el robot hasta los objetos presentes en el entorno siempre que estén dentro de su rango. Tiene distintos 6 tipos de volúmenes de detección diferentes. Se puede modificar su tamaño de volumen de detección o sus propiedades de detección desde el menú *Scene object properties*. Como se puede observar en la figura 40, estos 6 tipos de volúmenes son, en orden, con forma de haz, haz aleatorio, pirámide, cilindro, disco o cono.

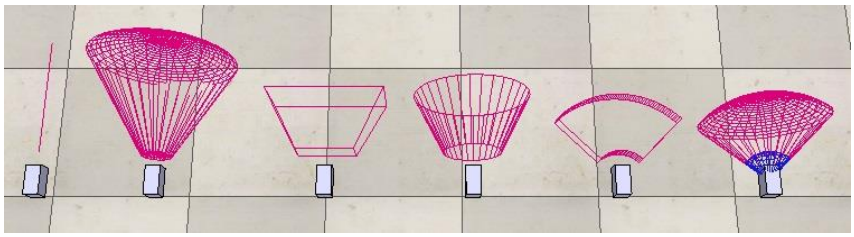


Figura 40. Tipo de voúmenes de sensores de proximidad.

También se puede elegir, dentro del mismo menú, el tipo de sensor que se quiere simular entre ultrasonido, infrarrojo, láser o capacitivo. Esto, junto la posibilidad de marcar al objeto porque tipos de sensores se desea que se pueda detectar, puede acercar bastante los sensores a la realidad. Por ejemplo, un objeto de cristal no es detectable ante sensores de infrarrojos y los sensores capacitivos detectan objetos metálicas o no metálicos.

- **Sensor de visión.** Simula cámaras que pueden captar imágenes de objetos renderizables (*renderables*). Pueden tener dos tipos de volúmenes de detección, rectangular (ortográfico) y trapezoidal (perspectiva), pudiéndose cambiar el tamaño de ambos. CoppeliaSim permite crear ventanas en la escena que muestren lo que ve la cámara. Estos sensores pueden usarse como cámaras de color (en RGB o escala de grises) o como cámaras de profundidad.
- **Sensores de fuerza.** Miden el par y la fuerza en sus tres ejes principales. Trabaja como una unión entre dos objetos que puede llegar a romperse si se sobrepasa determinada fuerza o par, pudiendo definir esa fuerza o par en *Scene object properties*.

En el caso del EV3 que se está usando, lleva un sensor de proximidad ultrasónico en el centro de la parte frontal 40, otro de visión en la esquina frontal derecha apuntando para el suelo 41 y otro de fuerza en la parte trasera 42.

Figura 41. Sensor ultrasónico EV3. (24)

Figura 42. Sensor de color EV3. (24)

Figura 43. Sensor táctil EV3. (24)

El sensor ultrasónico es un sensor que lanza ondas de sonido de frecuencia alta que se reflejan y vuelven al sensor, el cual mide el tiempo que tardan las ondas en hacer esto. Tiene un rango de distancia entre 3cm y 250cm con una precisión de +/-1cm. Se coloca un sensor se proximidad de tipo cono en la misma posición que el sensor ultrasónico modelado apuntando al frente. Es decir, se coloca en el centro de la parte frontal. Después, se definen sus propiedades. Se le define que es tipo ultrasónico y en los parámetros del volumen de detección *Scene object properties* > *Show volumen parameters* se ha reducido el ángulo de detección a 15º y se ha puesto un rango de 250cm.

El sensor de color del EV3 tiene 3 modos de uso: modo color, modo intensidad luz reflejada y modo luz ambiental. Para los programas que se plantearán después interesa sobre todo el segundo. Se crea un sensor de visión tipo trapezoidal y se coloca en la misma posición que el sensor de color del EV3. Por último, se cambian sus propiedades, bajando el ángulo a 45 en *Persp. Angle* y adjustando la distancia mínima en 0.001 y la máxima en 0.003 en *Near / far clipping plane*, ya que el sensor no va a ver nada más allá del suelo. También se puede subir la resolución a 64.

Para terminar de añadir sensores, se crea al de fuerza y se posiciona dentro del sensor táctil del EV3. Los cambios de este simplemente han sido disminuir su tamaño de 0.05 0.01 m.

5.2.6 Jerarquía de los modelos.

Como se ha dicho anteriormente, para que las piezas del robot se queden fijas sin desmontarse hace falta jerarquizarlas y añadirle articulaciones, las cuales también se jerarquizan. Como ya se han creado y situado las articulaciones y los sensores en los anteriores apartados, solo falta dicha relación de jerarquía de las piezas del robot. Este proceso consiste en arrastrar objetos en esta ventana y ponerlos en un nivel jerárquico más bajo, como “hijos” de otros objetos. El objeto padre puede acceder a las funciones de los

objetos hijos. Se recomienda siempre que las geometrías puras sean padres de las no puras, es decir, estén un nivel jerárquico por encima de ellas. De esta forma la geometría no pura es simplemente un objeto estático y estético cuya posición depende de la geometría pura, la cual sí posee las propiedades dinámicas y reactivas. Evidentemente, cuanto más se parezca el tamaño y volumen entre geometrías puras y no puras, más realista serán las colisiones. Las articulaciones, a su vez, deben ser superiores jerárquicamente a las formas puras que se quiere que dependan de ellas e hijas o pertenecientes al modelo general del robot. Para convertir una pieza en hija de otra solo hace falta arrastrarla encima de la otra en la ventana de la escena de jerarquía.

Dicho esto, el proceso propuesto para hacer la relación de jerarquía de un robot móvil es la siguiente:

1. Toda geometría pura que dependa de una articulación se agrupa seleccionándolas, haciendo click derecho y dándole *a Edit > Grouping / Merging > Group selected shapes* y, después, se hace hija de la articulación de la que depende.
2. Las geometrías no puras de las anteriores geometrías puras dependientes de articulaciones se agrupan y se hacen hijas de estas.
3. El resto de las geometrías no puras que no dependen de ninguna articulación se agrupan.
4. El resto de las geometrías puras que no dependen de ninguna articulación también se agrupan creando el objeto base.
5. Por último, la agrupación de geometrías puras independientes de articulaciones trabajará como base del robot y, los sensores, articulaciones y la agrupación de geometrías no puras se convierten en hijos de esta.

Al final, deberá quedar algo como la figura 44.

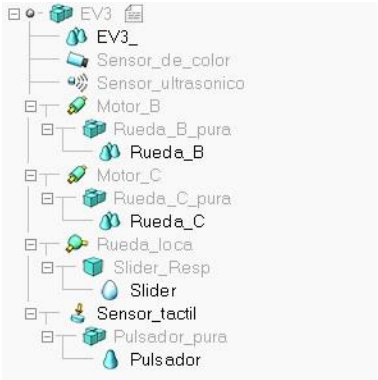


Figura 44. Relación de jerarquía del EV3 en CoppeliaSim.

Los objetos también podrían haber sido combinados en vez de agrupados, es decir, haber usado *Merge selected shapes* en lugar de *Group selected shapes*. CoppeliaSim aclara que la diferencia entre ellos es (26):

- **Merge:** obtiene una forma simple con un solo color y atributos común a todos los objetos que la componían. Dos formas fusionadas no siempre se pueden dividir de nuevo restaurando el estado inicial. Además, el resultado de la combinación es una sola forma, o así lo verá el motor de física. Si fusiona dos esferas, el motor de física dejará de reconocer las esferas y las manejará como una sopa de polígonos. La estabilidad de la simulación será mala y los tiempos de cálculo altos.
- **Group:** obtiene una forma agrupada con diferentes colores y atributos y siempre puede desagrupar para restaurar el estado inicial. El motor de física sí reconocería que aún se trata de dos esferas y optimizaría los cálculos.

Leyendo las dos definiciones está claro que agruparlas es mejor opción para este trabajo, así que esta es la opción escogida. Sin embargo, si el ordenador de trabajo es menos potente y no va bien la opción de agruparlos, se recomienda combinarlos.

5.2.7 Generación del modelo.

Un modelo solo puede existir en archivo ".ttm" en CoppeliaSim (27). Una vez se ha creado las articulaciones y los sensores y hecha una relación de jerarquía del robot, el último paso es guardarlo como modelo, para así poder usarlo cuando se requiera sin pasar otra vez por todo este proceso. La base del árbol jerárquico del objeto que se quiere convertir en modelo es la que debe ser marcado como objeto base del modelo. Dicho esto, se selecciona esta base en la ventana de jerarquía de escena y en la barra de menús se accede a *File > Save model as > CoppeliaSim model…* y se guarda el modelo.

Una vez generado el modelo de robot, se puede hacer que aparezca en el navegador de modelos. Tendrás que copiar el archivo ".ttm" del modelo en directorio de instalación de CoppeliaSim. Seguramente este directorio se sitúa en la unidad (c:) del ordenador y, dentro de esta, en archivos del programa. En el caso del ordenador y la versión de CoppeliaSim que se ha usado para la creación de este trabajo la dirección ha sido *c: > Archivos del programa > CopppeliaRobotics > models > robots > mobile*, situándolo así en la carpeta de robots móviles, como se muestra en la figura 45.

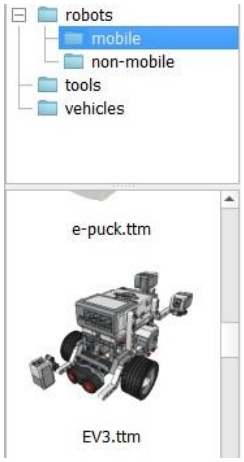


Figura 45. Modelo EV3 dentro del explorador de modelos.

Se ha incorporado el archivo de modelo de CoppeliaSim del EV3 modelado en este trabajo en la dirección *Modelos > CoppeliaSim*.

5.3 Creación de escenarios.

Para el diseño de escenarios, CoppeliaSim prácticamente no tiene límite. Con un modelador 3D externo y la opción de importar se puede obtener casi cualquier obstáculo u objeto que se desee. También se puede crear propios objetos o robots a base de formas puras y articulaciones, y cambiando sus propiedades visuales en el menú *Scene Object Properties*, y uniendo las partes creando su jerarquía. No obstante, si uno no desea complicarse tanto, en el navegador de modelos hay gran variedad de objetos, desde complejos robots hasta simples utensilios de oficina, o incluso terrenos o personas. Algunos de ellos con sus propios scripts hijos ya incrustados, aunque puedes cambiarlo cuando se desee. Además, cada uno de estos objetos pueden tener un control o movimiento independiente, debido a que otra ventaja a la hora de crear escenarios es que CoppeliaSim permite ilimitados scripts secundarios (28) asociados, permitiendo más de un robot por aplicación, cada uno con su propio script, u objetos, obstáculos o robots secundarios con movimiento automatizado propio. Para saber en más profundidad sobre los scripts de CoppeliaSim, se habla de ellos en el apartado 5.4.

Por otro lado, en *add > light*, dentro de la barra de menús, además se puede simular que los objetos emiten luz, propiedad interesante e imposible de simular en RVW que usaremos más tarde cuando se lleve a cabo la simulación del comportamiento de Braitenberg.

También dentro de este mismo menú, en *path*, se puede crear cualquier forma o diseño de líneas, dibujos o losas para el suelo, además de diseñar muros de cualquier tamaño, volumen o forma y en con cualquier disposición. La edición de estos caminos se explica más abajo. Para añadir y diseñar una ruta de camino, simplemente se selecciona añadir *segment type* y después el icono *Toggle path edit mode* del menú de herramientas vertical ( ) y, dentro de

este modo edición, se crean tantos nuevos puntos de camino como se quiera, que luego se pueden mover creando cualquier ruta, ya sea de líneas o muros. El camino simplemente sigue estos puntos por orden. Un ejemplo de esto es la escena creada para la tarea de seguimiento entre paredes, donde se ha diseñado un muro en espiral siguiendo esto, para luego, en el menú que se ve en la figura 47 y se explica más abajo, se le da forma vertical y se genera la forma en *Generate Shape*.

Figura 46. Escena de la aplicación de seguimiento entre paredes en CoppeliaSim

Para poder ver las propiedades de estar rutas de camino y los tipos y formas que tienes mira la creación de la escena de la aplicación de "Robot Limpiador" más abajo.

También se puede cambiar el diseño del suelo. Si se le da al script que te viene adjunto con el modelo del suelo con toda nueva escena, se abre un menú para cambiar el tamaño del suelo. También puedes cambiar el suelo arrastrando uno desde el navegador de modelos o cambiar las propiedades, textura o color desde el menú de propiedades de escena del suelo.

En el caso del "Robot Limpiador" lo que se quiere crear es una circunferencia, así que lo que se selecciona es *Add* > *Path* > *Circle*. Lo siguientes es cambiar las propiedades. Se selecciona y se abre el menú de propiedades *Scene Object Properties*. Se accede al cuadro de diálogo donde se puede editar la ruta, *Show Path Shaping Dialoge*. Se habilita la edición de la ruta con *Path shaping enable* y, luego, en *Section Characteristics* se configura el relieve de la línea en *Type*. Se deja en *Horizontal segment* para que sea plano. Si se deseara un muro solo habría que cambiarlo a *Vertical segment*. En *Scaling Factor* se puede configurar el grosor que, en este caso, se ha dejado en 0.5. Se puede ajustar el color, para hacerlo negro, en *Adjust Color* y luego *Ambient/Diffuse* Component. Se recomienda quitar todas las opciones en *Visual properties* para que no estorben.

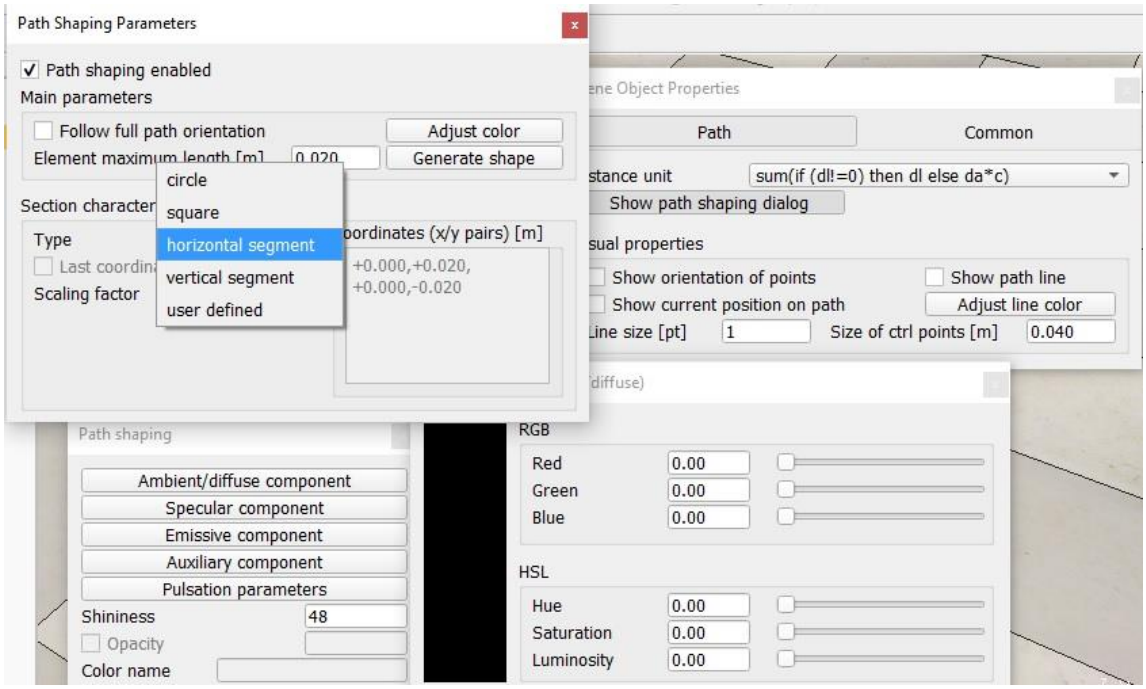


Figura 47. Menús de propiedades de Path en CoppeliaSim.

Ahora con la circunferencia seleccionada, se pincha en el icono de *Toggle path edit mode* y se activa *Path is flat* para asegurarse que el camino es plano y *Keep X UP* para asegurarse que se mantiene la misma distribución de coordenadas (*x* hacia arriba) en todos los puntos de control. *Path is closed* cierra el segmento, en este caso activado ya que hablamos de un círculo.

Para terminar con la circunferencia quedaría modificar el radio. Con la circunferencia seleccionada, en la ventada de la escena, arriba, se puede leer que pone "total length" (longitud total), es decir, el perímetro de la circunferencia. La circunferencia se crea con 1.527m de perímetro. La fórmula del perímetro es

$$P = 2\pi r$$

Entonces, si queremos un círculo de r = 1m, el perímetro necesario será P = 6.28319, luego la escala que hay que aplicar es 6.28319/1.572 = 3.9969, que es aproximadamente 4 veces más grande. Se abre *Scene Object Properties* > *Common* > *Scaling* y aplicamos 4.

Por último, para añadir las bolas bastaría con *Add* > *Primitive Shape* > *Spheres*. Se seleccionan y se abre el menú de propiedades y, en *Common* > *Object special properties* se activan todas las opciones menos *Renderable* (detectable para un sensor de color). Si se quiere saber más sobre las propiedades de los objetos, se debe acudir a el apartado 5.2.3. En *Shape* > *Adjust color* se les pone un color rojo únicamente por gusto para que se parezca al escenario de RVW. En Shape > Show dynamic properties deberán estar activadas las opciones *Body is respondable* (reactivo) y Body is dynamic (dinámico).

Con esto ya se tiene creada la escena de “Robot Limpiador” a falta de añadir el EV3 o arrastrarlo desde el navegador de modelos. La escena ha debido quedar parecida a la de la figura 48. También se puede encontrar un archivo con esta escena creada en la dirección Escenas > CoppeliaSim con el nombre de “Robot Limpiador”.

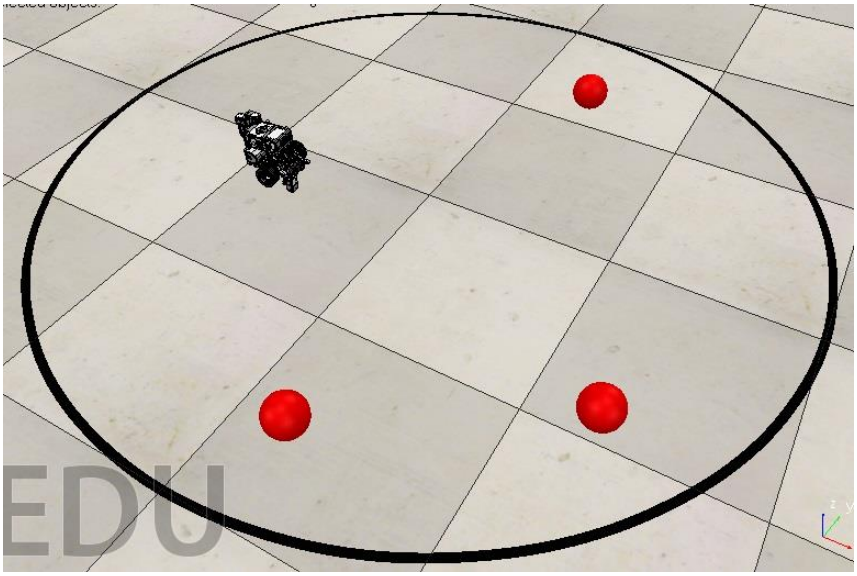


Figura 48. Escena de Robot Limpiador en CoppeliaSim.

Para el caso del Robot de Braitenberg, simplemente se ha añadido una forma pura cúbica de 6x0.05x0.3m y se ha copiado y pegado tres veces más para luego disponer estas 4 geometrías en forma de cuadrado creando una habitación donde se desarrollará esta aplicación. Seguido, se han añadido 6 luces en forma de lámparas en *add* > *light* > *spotlight*, y jugando con el tamaño y difusión de la luz en el menú *Scene Object Properties*. Es importante deshabilitar las luces predeterminadas de la escena, que se pueden encontrar como *DefaultLights* en la ventana de jerarquía de escena, desactivando la opción *Light is enable* en su menú de propiedades. El resultado debería quedar como en la figura 49.

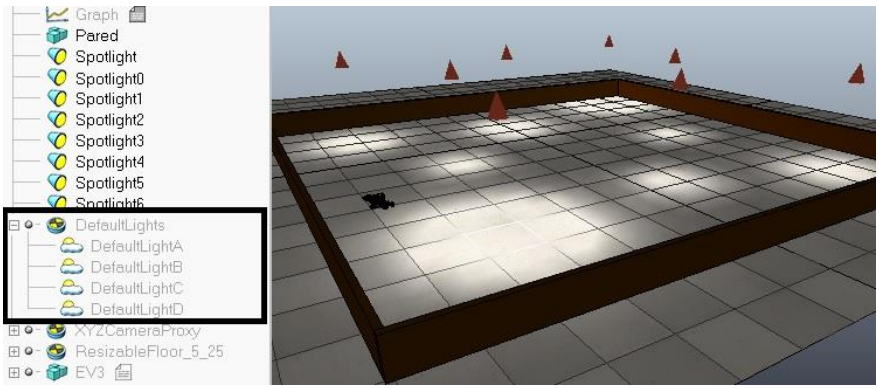


Figura 49. Escenario de la aplicación de "Robot de Braitenberg".

5.4 Scripts internos.

CoppeliaSim es un simulador altamente personalizable: casi cada paso de una simulación es definido por el usuario. Esta flexibilidad está permitida a través de un intérprete de script integrado. El lenguaje de comandos interno de este simulador es Lua. Lua es un lenguaje interpretado, es decir, no requiere del paso adicional de la compilación, el cual convierte el código que escribes a lenguaje de máquina, antes de ser ejecutado. Un lenguaje interpretado, por otro lado, es convertido a lenguaje de máquina a medida que es ejecutado (29). La declaración de variables puede ser mediante asignación, es decir, no hace falta declararlas al principio, se pueden crear solas a mitad del programa cuando le asignes un valor. Al ser un lenguaje interpretado también es innecesario el uso de punto y coma al final de cada instrucción.
Algunas de las palabras clave más importantes para hacer bucles o condiciones en este lenguaje son:

PALABRAS CLAVE				
and	break	do	else	else if
end	false	for	function	if
in	local	nil	not	or
repeat	return	then	true	until
while				

Por otro lado, los siguientes operadores denotan otros elementos, por ejemplo, aritméticos o relacionales.

Strings						
+	-	*	/	%	^	#
==	~=	<=	>=	<	>	=
(	)	{	}	[	]	
;	:	,	.	..	…	

La mayoría de estos strings son lógicos e intuitivos, ya que son iguales a la mayoría de stings de otras plataformas de programación. El string ".." se usa para concatenar frases o variables en las funciones que muestran textos por la ventana de mensajes de CoppeliaSim (sim.addStatusBarMessage()), como puedes ver en el ANEXO II.

Las estructuras de control de más importantes, como if, while, repeat o for, tienen el significado habitual. Su sintaxis es:

- **while** condición **do** instrucciones **end**
- **repeat** instrucciones **until** condición
- **for** variable=exp1, exp2, exp3 **do** instrucciones **end**
- **if** condición **then** instrucciones (**elseif** condición **then** instrucciones) (**else** instrucciones) **end**

Para el bucle for, las instrucciones se dan desde que la variable tiene el valor “exp1” hasta que sobrepasa “exp2” con un paso de “exp3”

En cuanto comandos matemáticos, Lua tiene la mayoría de los comandos iguales al resto de programadores, los cuales hay que llamarlos de su respectiva librería con el comando “math.”, por ejemplo “math.cos” sería la función de coseno.

Para saber más sobre palabras clave y como crear bucles o condiciones, para conocer mejor los comandos de los operadores aritméticos, relacionales o lógicos, conocer funciones matemáticas, o cualquier otra cosa relacionada con el lenguaje en Lua, se puede mirar el manual de Lua de la referencia (30).
Por otro lado, las funciones de Lua están basadas en las funciones de C.Además, CoppeliaSim extiende los comandos de Lua y agrega comandos específicos de CoppeliaSim que pueden ser reconocidos por sus prefijos “*sim.*” (31). Algunas de las funciones más importantes a la hora de programar leyendo sensores y suministrando acciones de control para el EV3 pueden ser:

- **sim.getObjectHandle:** obtiene el manejador de un objeto. Es importante para poder acceder a cualquier sensor o motor y usarlo en la programación, al igual que pasaba con los scripts internos de CoppeliaSim.
- **sim.readProximitySensor:** lee el sensor de proximidad. Le entra el nombre del manejador del sensor de proximidad que se desea leer y devuelve el estado de la detección (detectado o no), la distancia, un vector con las coordenadas relativas de la posición del punto detectado ye el manejador del objeto detectado.
- **sim.getVisionSensorImage:** lee el sensor de color. Le entra el nombre del manejador del sensor de color que se desea leer y la posición y tamaño dela imagen y una matriz con cada una de la escala de cada píxel de la imagen. Si se quiere que el valor devuelto esté en escala de grises, además hay que añadirle a los parámetros de entrada la función “sim.handleflag_greyscale”. Para obtener solo un valor se podría simplemente coger el primer valor.
- **sim.setJointTargetVelocity:** ajusta la velocidad de la articulación o motor, dando como parámetro de entrada a la función el manejador del motor que se desea ajustar y la velocidad. Angular o lineal dependen de la articulación.
- **sim.setJointTargetPosition:** ajusta la posición de la articulación o motor, dando como parámetro de entrada a la función el manejador del motor que se desea ajustar y la posición. Angular o linear dependen de la articulación.
- **sim.getJointTargetVelocity:** obtiene la velocidad de una articulación o motor. A la función le entra el manejador del objeto del cual se desea obtener la velocidad y la velocidad. Angular o lineal dependen de la articulación.
- **sim.getJointTargetPosition:** obtiene la posición de la articulación o motor. Le entra el manejador de la articulación y devuelve la posición. Angular o lineal dependen de la articulación.

- **sim.readForceSensor:** lee el sensor de fuerza. Entra el manejador del sensor de fuerza y devuelve el estado del sensor, seguido de dos vectores de tamaño tres que representan la fuerza y el torque.

Están son básicamente las funciones más importante necesarias para usar los sensores y los motores del EV3. En algún programa también se ha usado "sim.getObjectPosition" para obtener la posición del robot respecto a la escena. Otra función interesante sería "sim.getObjectVelocity", sin embargo, estas últimas funciones no han sido nombradas como importantes ya que este simulador se está comparando con la programación de RobotC y están no existen en dicha plataforma. Muchas de las funciones dichas han sido usadas en los programas del trabajo, y como todas las funciones usadas, se pueden encontrar definidas en el ANEXO II, con la explicación de su código y forma de uso. Tambiém hay listas de las funciones de Lua que son de gran utilidad. Se puede encontrar estas listas ordenadas alfabéticamente (32) o por categorías (33) donde se compara cada comando de Lua con su respectivo equivalente de C++, ya que la programación de Lua está basada en C++.

Para obtener la posición también se pueden usar gráficas que incorpora CoppeliaSim en *Add > graph*, como se puede ver en el apartado 5.4.3, donde se ha creado una la cual se puede mirar como ejemplo. El periodo de muestreo se encuentra en la barra de herramientas horizontal (Simulation time step) junto con ajustes dinámicos de simulación. Por defecto este periodo es de 50ms, y se recomienda no tocar ninguno de estos ajustes si no se posee grandes conocimientos de programación de este simulador, pues puede ralentizar la simulación.

En cuanto a sus scripts, son incrustados en escenas o modelos, formando parte de la escena y guardándose y cargándose junto con el resto de la escena o del objeto asociado. Pueden ser creados tantos como se desee y ejecutados a la vez en la misma simulación. Hay 2 tipos principales de scripts incrustados:

- **Simulation scripts:** son scripts que se ejecutan solo durante la simulación y que se utilizan para personalizar una simulación o un modelo de simulación. A su vez, los scripts de simulación se pueden dividir en:
    - **Main script.** Cada escena tiene un script principal que maneja toda la funcionalidad y se encarga de llamar a los guiones secundarios. Sin el script principal, no se puede ejecutar una simulación. Este script ya viene definido y asociado al nombre de la escena cuando creas esta. Se recomienda no tocarlo.
    - C**hild scripts.** (34) Cada objeto de escena se puede asociar con un script secundario que manejará una parte específica de una simulación. Una particularidad de ellos es que también pueden ejecutar subprocesos. Los modelos o robots suelen ser controlados por este tipo de scripts. Pueden ser de dos formas:

- **Non-threaded scripts.** No se ejecutan en hilos separados del propio proceso de simulación. El scripts principal irá ejecutando una serie de pasos y en un momento de la simulación llamará a este script, y cuando el script termine devolverá el control al script principal. Es decir, están de paso en la secuencia principal de la simulación. Es la forma más habitual.
- **Threaded scripts.** Son lanzados en un hilo de ejecución aparte**.** Estos comandos secundarias tienen varias debilidades en comparación con las secuencias de comandos secundarias sin subprocesos si no se programan adecuadamente: requieren más recursos y pueden perder algo de tiempo de procesamiento. Sin embargo, son los que se deben usar para secuencias de bucle infinito que no devolverían el control al script principal, como es el caso del Robot Limpiador, o para objetos que se mueven totalmente independientes a la escena o una parte específica de la simulación como, por ejemplo, para el manejo una puerta corredera automática.

- **Customization scripts**: son scripts que también se pueden ejecutar mientras no se ejecuta la simulación, y que se utilizan para personalizar una escena de simulación o el simulador en sí. Por ejemplo, el suelo por defecto que viene con las escenas lleva este tipo de scripts.

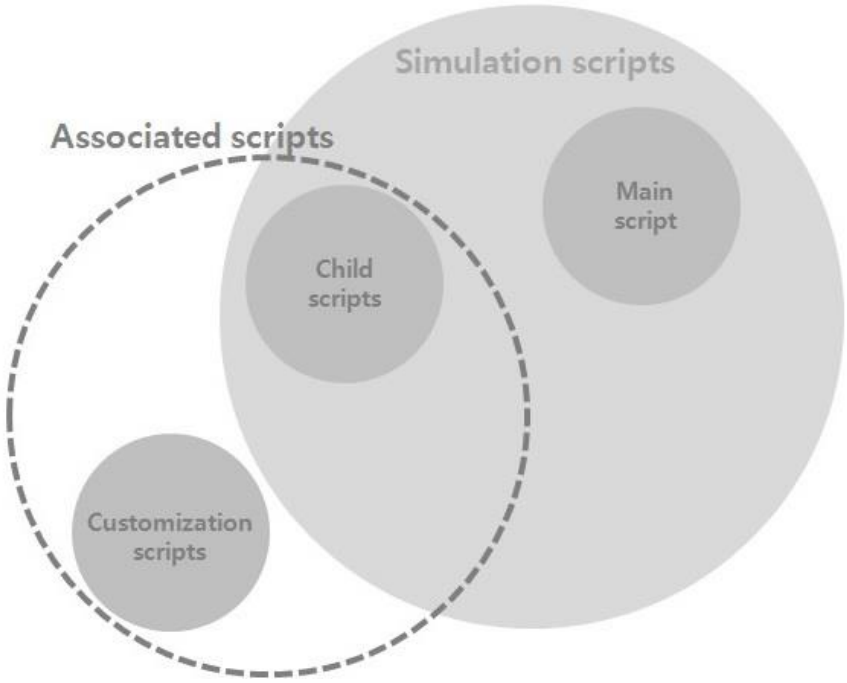


Figura 50. Esquema de tipos de scripts embebidos cogido de la página de CoppeliaSim.

Se puede encontrar estos diferentes tipos de script asociados nombrados para añadirlos a algún objeto en *Add* dentro de la barra de menús, o bien, desde el menú de comandos en *Tools > Scripts*.

Cuando se crea un script asociado non-threaded y se abre, se observa que el script lleva por defecto 4 determinadas funciones. Estas funciones dividen el script en las siguientes partes:

- **sysCall_init.** La función de inicialización. Suele usarse para llamar a todos los manejadores al principio de la función. Cada sensor, objeto o articulación tiene su manejador, que tiene que ser llamado por unas funciones que obtienen el nombre de estos manejadores para, después, poder usarlo en las funciones siguientes permitiendo su control. Sin su manejador un actuador no puede ser controlado. Es la única función del script que no es opcional.
- **sysCall_actuation.** En esta función se espera que se muevan los diferentes actuadores o articulaciones.
- **sysCall_sensing.** Esta función está reservada para que los sensores lean.
- **sysCall_cleanup.** Función de finalización, utilizada para limpiar todo lo que hayas hecho que se quiera limpiar, aunque no es muy habitual.

Si por el contrario se crea un threaded script, estas funciones se dividen en:

- **sysCall_threadmain.** Esta función no es opcional. Se ejecutará cuando se inicie el subproceso, hasta poco antes de que finalice el subproceso. Por lo general, se pondría algo de código de inicialización, así como el bucle principal en esta parte.
- **sysCall_cleanup.** Igual que en los non-threaded scripts. Esta parte se ejecutará una vez justo antes de que finalice una simulación o antes de que finalice el subproceso.

5.4.1 Robot Limpiador.

Se ha querido poner a también la aplicación de “Robot Limpiador” para este lenguaje interno. El escenario con el script embebido que programa esta aplicación puedes encontrarlo en la dirección *Escenas > CoppeliaSim > Robot Limpiador Lua*.

Se puede ver en el video “CoppeliaSim Robot Limpiador LUA” de la carpeta *Videos* con el resultado, donde se aprecia que el robot cumple perfectamente con la aplicación. Sin embargo, si se comparan las simulaciones de ambos entornos, se ve que la respuesta, aunque exitosa, no es exactamente la misma. Se deduce que la razón es debida a dos factores. El primero es que en RVW el robot viene ya configurado mientras que en el CoppeliaSim lo hemos configurado manualmente, teniendo así más inexactitud en sus especificaciones técnicas. La segunda razón es la forma del robot. Mientras que en CoppeliaSim no le hemos

puesto ningún brazo al robot, pues no hacía falta, en RVW tiene una especia de brazo delantero que deja pillada la bola sin posibilidad de quitarlo.

Se puede observar que se usa un script de hijo hilado pues, al ser el programa un bucle infinito, si se usara un script hijo no-hilado la simulación daría problemas, como se podría comprobar si se intentara.

Todas las funciones usadas de LUA para la programación de esta aplicación pueden encontrarse definidas y explicadas en el ANEXO II, facilitando la programación y ayudando a entenderlas.

También es importante nombrar que la velocidad angular en este entorno está en grados entre segundo, para tenerla en cuenta para posibles cambios de unidades.

5.4.2. Robot de Braitenberg.

Una de las aplicaciones que se planteaban resolver es el Robot de Braitenberg. Como ya se ha dicho, esta aplicación que iba a ser base del presente TFG antes del cambio debido a la pandemia se pone a prueba solo en CoppeliaSim, ya que no es posible en el simulador RVW.

Al cambiar el enfoque del trabajo esta aplicación dejo de ser el centro de trabajo, asique no se va a indagar demasiado en ella y se va a resolver de forma sencilla. Al robot se le han puesto 3 sensores en cada lado que medirán la luz del suelo dando más velocidad a la rueda del lado por donde más luz haya, siendo así capaz de esquivar esta. Todo esto se explica en el apartado 3.3. Los sensores están distribuidos en forma de triángulo, es decir, dos al frente y los siguientes cada vez más atrás y laterales. Se ponderan y se les da más importancia a los sensores delanteros, para así esquivar más rápido la luz que se acerca frontalmente y darle menos importancia a la que este a los lados del robot, pues influye menos. Como el motor del robot tiene una velocidad máxima de 170 rpm y el simulador se mide en grados por segundo, la velocidad angular máxima de las articulaciones es de 28,4 g/s. Si se comprueba el sensor de color en este escenario se puede ver que como máximo el valor recogido ronda entre 0,9 y 1. Así, se obtiene la velocidad interpolando, buscando una velocidad máxima de 28.4 para 9.5. El valor mínimo de la velocidad no se quiere que llegue a cero para así que el robot no pare de moverse (aunque en el Robot de Braitenberg, éste debería pararse al encontrar la sombra). Por otro lado, se ha creado una pared para encerrar el robot y que no salga del escenario, así que se a implementado también más velocidad en la respectiva rueda si el sensor ultrasónico delantero detecta la pared por algún lado para así esquivarla.

En la dirección *Escenas* > *CoppeliaSim* se puede encontrar el escenario de la aplicación creado que ya incorpora el script hijo con el algoritmo. También se puede ver el resultado en el video “Robot de Braitenberg” de la carpeta *Videos*, donde se observa como evita las zonas con luz y acelera en estas para salir de ellas.

Para ver la trayectoria que ha hecho el robot evitando las luces, se ha creado un gráfico en CoppeliaSim en *Add > graph*. Luego en el menú de propiedades se añade las variables que se quieren registrar en *Add new data stream to record*, donde se selecciona las posiciones *x* e *y* del robot EV3, como se puede ver en la figura 51. Luego se añaden estas variables a la gráfica 2D en el menú *XY graphs/3D curves > Edit XY graphs*, y dentro de este menú en *Add new curve*.

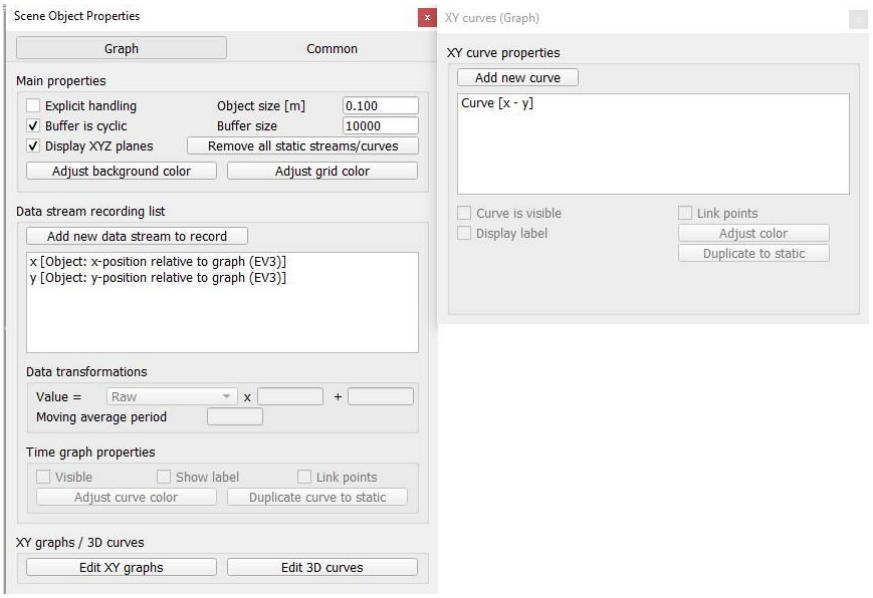


Figura 51. Menú de propiedades de un gráfico en CoppeliaSim.

Una vez configurada la gráfica y se haya ejecutado la simulación durante un tiempo, el resultado obtenido es el de la figura 52. Se puede ver como las formas huevas de la gráfica cuadran con las zonas iluminadas de la escena, y que el robot ha rodeado estas con éxito.

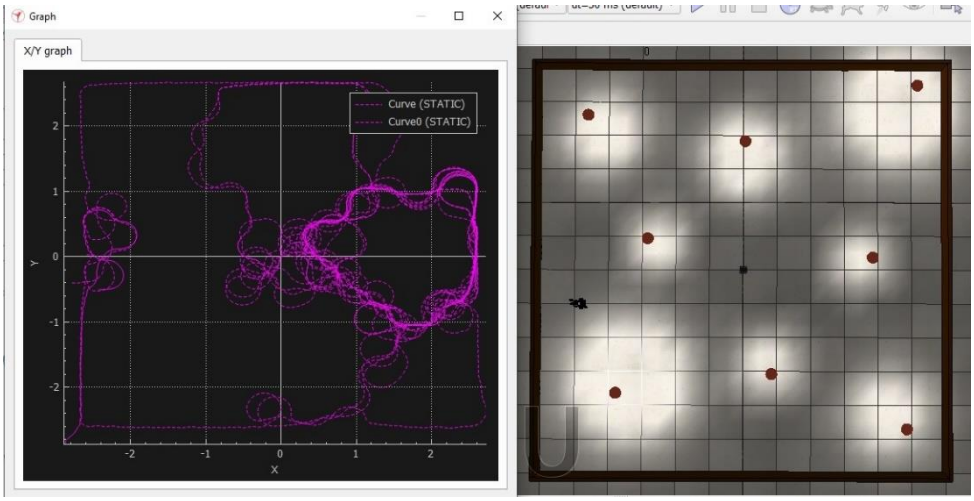


Figura 52. Gráfica de la posición del robot resultado de la aplicación Robot de Braitenberg en CoppeliaSim.

5.4.3 Seguimiento entre paredes.

Se programa un script hijo en Lua basándose en el razonamiento y funciones explicadas en el apartado 3.4. Una vez obtenidos los resultados, se puede apreciar como el robot responde con éxito al seguimiento entre paredes, incluso aunque estas hayan sido creadas con *segment path* manualmente resultado unas paredes bastante irregulares. Para la comprobación de los resultados se puede ver el video "Seguimiento entre paredes" de la carpeta de videos. También puedes encontrar el escenario con el script de la programación incluido en la dirección *Escenas > CoppeliaSim > Seguimiento de paredes*.

También, para mostrar su resultado, se ha creado una gráfica siguiendo el procedimiento del anterior apartado de Braitenberg mostrando la trayectoria del robot. Los resultados, donde se ve como sale perfectamente de una irregular espiral creada, se muestran a continuación.

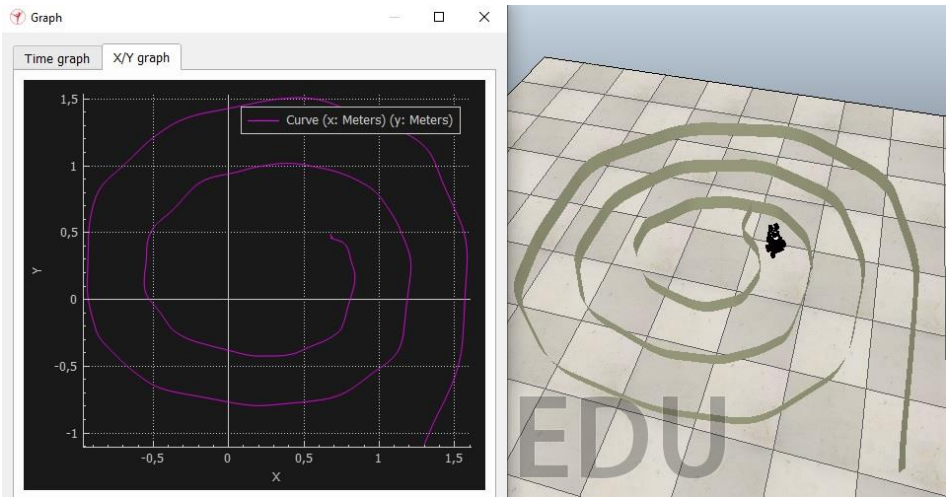


Figura 53. Gráfica de la posición del robot resultado del seguimiento entre paredes en CoppeliaSim.

5.5 API Remota.

A parte de su script interna o regular API Lua, CoppeliaSim tiene otras formas de programación. Se admiten seis enfoques diferentes de programación o codificación, los seis son compatibles entre sí, es decir, se pueden usar al mismo tiempo. Son las siguientes formas:

- **Scripts embebidos.** El lenguaje interno de CoppeliaSim, Lua, ya ha sido explicado en el anterior apartado. Esta flexible secuencia de comandos es el enfoque de programación más fácil y utilizado.
- **Add-on.** Se carga automáticamente al inicio del programa y permite que la funcionalidad de CoppeliaSim se extienda mediante funciones o funciones escritas por el usuario. Tienen como finalidad la personalización de CoppeliaSim en sí.
- **Plugin:** Un *plugin* es una biblioteca compartida que se carga en CoppeliaSim. Permite que la funcionalidad de CoppeliaSim se extienda mediante funciones escritas por el usuario, como con *Add-ons*.
- **API remota.** Este método permite que una aplicación se conecte a CoppeliaSim de una manera muy fácil, utilizando comandos remotos de la API**.** Consiste en una biblioteca de instrucciones de un programa para poder ser usadas en otro programa. Existen una gran cantidad de APIs las cuales permiten conectarse a CoppeliaSim de forma sencilla (35).
- **Nudo ROS.** Este método permite que una aplicación externa se conecte a CoppeliaSim a través de ROS, el sistema operativo del robot.
- **Nodo BlueZero.** Este método permite que una aplicación externa se conecte a CoppeliaSim a través de BlueZero y actúa como cliente/ servidor. Este método es el más complejo.

Este trabajo se va a centrar, concretamente, en la forma de API remota (aparte de en los scripts embebidos en el apartado anterior). Se ha elegido esta forma de programación por que la finalidad del TFG es encontrar una manera, lo más versátil y sencilla posible, de resolver trabajos de programación de robótica en casa, y entre las posibles formas de conexión con un programador externo, hacerlo por API es la forma más fácil. Es compatible con los siguientes lenguajes: C / C ++, Java, Python, Matlab, Octave y Lua. Aquí se ha usado una API remota para conectarnos con Matlab para resolver las tareas.

5.5.1. Conexión CoppeliaSim-MATLAB a través de API remota.

A continuación, se va a explicar cómo trabajar con CoppeliaSim a través de MATLAB en un ambiente de cooperación. Como ya se ha dicho, se ha elegido este tipo de conexión porque es bastante sencilla de hacer. Solo habría que seguir los siguientes pasos:

- Como primer paso, se crea una carpeta que contendrá todos los elementos que se van a necesitar o, en vez de esto, se meten todos estos elementos dentro de la carpeta con la que se vaya a querer trabajar en MATLAB. Entonces, se tiene que ir buscar la carpeta de CoppeliaSim en los archivos de programa de tu ordenador. Una vez aquí, entramos en *programming* > *remoteApiBindings* > *matlab*, se copia todos los archivos de esa carpeta y se pegan en la carpeta en la que se va a trabajar.
- Ahora, lo que se debe hacer es ir a *programming* > *remoteApiBindings* > *lib* y, en el caso de Windows, entrar a la carpeta *Windows* y copiar el archivo que hay dentro también de donde se va a trabajar. Evidentemente, si el sistema operativo donde se está trabajando fuera otro, se entra a su respectiva carpeta.
- Una vez hecho esto, se abre CoppeliaSim con el escenario que se quisiera controlar desde MATLAB y, en cualquier objeto de la escena, se agrega un child script. Se va ahora a la carpeta donde se ha copiado todos los archivos y se abre uno archivo llamado "*simpleTest*" de forma que sea posible la visualización de su texto. En él se debería poder leer algo parecido a "simRemoteApi.start(19999*)*", texto que ahora se debe copiar y pegar en la parte de inicialización del script que se acaba de crear. Esto permite generar un API remoto dentro de la simulación partiendo, en este caso, por el puerto 19999.
- Dentro de MATLAB, hay que navegar y posicionarse en esta carpeta donde están los archivos y se va a trabajar. Ahora debería de poder visualizarse estos archivos en la ventana de *Current Folder.* Se selecciona y se abre "*simpleTest*". Si se inicializa la simulación en CoppeliaSim y, seguido, en MATLAB con el archivo "*simpleTest"*, se puede ver que la conexión ya está realizada. Este programa lo que hace es escribirte en *Command Window* en MATLAB la posición del ratón siempre que se esté moviendo dentro de la ventana de escena de CoppeliaSim. Es un simple programa de prueba para verificar que se hace la conexión.
- Lo que realmente ha creado la conexión son las siguientes instrucciones al principio de este archivo, los cuales deberán de copiarse en cualquier programa con el que se desee este control remoto.

```
vrep=remApi('remoteApi');
vrep.simxFinish(-1); % just in case, close all opened connections
clientID=vrep.simxStart('127.0.0.1',19999,true,true,5000,5);
if (clientID>-1)
disp('conectado');
…
end
```

(Se ha cambiado donde antes, en el fichero "*simpleTest*", ponía "*sim*" por "*vrep*" por comodidad, ya que la mayoría de las funciones ya empiezan por *sim-* y este nuevo código, como se verá más adelante, se tiene que poner delante de la mayoría de las funciones también. De esta forma no quedaba tan repetitivo el comienzo de las funciones y se evitaba confusiones.)

- Por último, aunque opcional, se han añadido a los programas las instrucciones

```
vrep.simxAddStatusbarMessage(clientID,'comunicacion con MATLAB
iniciada',vrep.simx_opmode_blocking);
disp('Comunicacion con CoppeliaSim iniciada');
```

para comprobar la conexión en ambos programas. En las ventana de comandos de ambos debería salir un mensaje verificando la conexión.

Todos los archivos usados para esta conexión se pueden encontrar también en la carpeta de la dirección *Programas* > *Coppelia-MATLAB* > *Conexión*. Los archivos de escenarios y programas utilizados de la carpeta compartida ya tienen estos pasos hechos.

La programación en MATLAB para esta conexión es parecida la de MATLAB en general. El lenguaje, sus palabras clave, operadores aritméticos, operadores relacionales, bucles, etc. son los del lenguaje de MATLAB.

La principal diferencia son las funciones usadas de la biblioteca de la API remota, las cuales influyen directamente en la simulación en CoppeliaSim. Se puede encontrar una lista de las funciones de esta API remota en la referencia (35), con cada una de sus explicaciones y modos de empleo. A la hora de usar estas funciones remotas, es importante poner antes de cada función el código que se ha creado con "[]=remApi('remoteApi')" para indicar que la función pertenece a una API remota (en el caso del trabajo es "vrep.", como se puede ver en el quinto punto de las instrucciones para crear la conexión entre CoppeliaSim y MATLAB o en los ejemplos del ANEXO II).

Algunas de las funciones más importantes para la programación del EV3, leyendo sus sensores y determinando las acciones de control de motores pueden ser, como en los script internos de CoppeliaSim, las siguientes:

- **simxGetObjectHandle:** obtiene el manejador de un objeto. Es importante para poder acceder a cualquier sensor o motor y usarlo en la programación, al igual que pasaba con los scripts internos de CoppeliaSim.
- **simxReadProximitySensor:** lee el sensor de proximidad. Le entra el nombre del manejador del sensor de proximidad que se desea leer y devuelve el estado de la detección (detectado o no) y un vector con las coordenadas relativas de la posición del punto detectado. No devuelve el valor de la distancia, pero podría sacarse fácilmente con el módulo de la suma de las tres coordenadas de posición de los distintos ejes.

- **simxGetVisionSensorImage2:** lee el sensor de color. Le entra el nombre del manejador del sensor de color que se desea leer y el tipo de lectura (escala de grises o escala RGB) y te devuelve un vector con dos números valores de su resolución y una matriz con cada una de la escala de cada píxel de la imagen. Para obtener solo un valor podría hacerse la media de todos los valores o simplemente coger el primer valor.
- **simxSetJointTargetVelocity:** ajusta la velocidad de la articulación o motor, dando como parámetro de entrada a la función el manejador del motor que se desea ajustar y la velocidad.
- **simxSetJointTargetPosition:** ajusta la posición de la articulación o motor, dando como parámetro de entrada a la función el manejador del motor que se desea ajustar y la posición.
- **simxGetObjectVelocity:** obtiene la velocidad de un objeto. A la función le entra el manejador del objeto del cual se desea obtener la velocidad y devuelve 2 vectores de tamaño 3, con las velocidades lineales y angulares en cada uno de los tres ejes. (en la API remota de MATLAB no existe función para obtener la velocidad directamente de un motor o articulación).
- **simxGetJointPosition:** obtiene la posición de la articulación o motor. Le entra el manejador de la articulación y devuelve la posición de la misma. Devuelve grados de rotación si es una articulación de revolución y devuelve la cantidad de traslación si es prismática.
- **simxReadForceSensor:** lee el sensor de fuerza. Entra el manejador del sensor de fuerza y devuelve el estado del sensor, seguido de dos vectores de tamaño tres que representan la fuerza y el torque.

Además, en todas las funciones hay una serie de parámetros que se repiten:

- **number clientID.** Parámetro de entrada. Variable del número de cliente obtenida al realizar la conexión CoppeliaSim-MATLAB. En el trabajo tiene siempre tiene el nombre de clientID y se usa en todas las funciones de la API remota.
- **number returnCode.** Parámetro de salida. Variable de un número de retorno que da la función. Puedes ver lo que indica el número de retorno en la referencia (36). En el trabajo tiene siempre tiene el nombre de returnCode y se usa en todas las funciones de la API remota.
- **number operationMode.** Parámetro de entrada. La función del modo de operación. Puedes encontrar estas funciones en la referencia (36). Como el resto de funciones debe ir acompañada del prefijo “vrep.”.

Para encontrar la explicación de las funciones, la mayoría de estas están en el ANEXO II, donde están todas las funciones usadas en los programas del trabajo y donde se define el comando de cada una de estas funciones, junto con sus parámetros de entrada y parámetros de salida. También pueden buscarse en la lista de la referencia (35).

El resto de lenguaje es igual al lenguaje de MATLAB.

Para su simulación, una vez hecha la conexión simplemente se ejecuta la simulación de CoppeliaSim y, seguido, se le da a ejecutar la programación de MATLAB. Una vez ejecutado, si no hay ningún comando o instrucción para terminar la simulación, cuando acabe las instrucciones de la programación de MATLAB la ejecución se terminará, mientras que en CoppeliaSim sigue ejecutando las ultimas instrucciones hasta que la simulación se pare manualmente.

La instrucción para indicar que la ejecución ya ha finalizado y establecer la conexión puede ser como la siguiente:

```
vrep.simxAddStatusbarMessage(clientID,'comunicacion con MATLAB
finalizada, vrep.simx_opmode_blocking);
vrep.simxfinish(-1);
disp(comunicación con CoppeliaSim finalizada);
vrep.delete();
clear all; clc;
```

5.5.2 Control Cinemático de trayectorias.

Para el control cinemático se ha aprovechado de los esquemas ya generados de Simulink que pueden verse en el ANEXO I. Se ejecuta estos esquemas de Simulink y se ajustan con la conexión de API remota las acciones de control a las velocidades de las ruedas del robot, generando la trayectoria, como se puede comprobar en el programa situado en *Programas* > *Matlab-CoppeliaSim* > *Control_trayectoria_circular*. El resultado se puede ver en el video “Control Trayectorias Circular CoppeliaSim-MATLAB”. Puedes encontrar las definiciones de las funciones usadas de la API remota de MATLAB en el ANEXO II.

Aunque la circunferencia se ejecutaba bastante bien, como se podrá comprobar en la gráfica del robot, la gráfica de la trayectoria del robot salía desplazada respecto la de la trayectoria de referencia. Esto es debido a que en Simulink y RobotC toma el centro de la circunferencia generada como origen de coordenadas, mientras que en CoppeliaSim el origen de referencia de las coordenadas debe de ser un objeto relativo a partir del cual se compara. En el caso del trabajo respecto al origen o centro de la simulación, el cual coincide con el del suelo. Por ello se ha ido moviendo el robot poco a poco hasta cuadrar, en la medida de lo posible, ambas circunferencias: la de referencia y la del robot simulado. El resultado se puede ver en la figura 54. El escenario con el robot posicionado para ajustar en la medida de lo posible ambas circunferencias se puede encontrar en la dirección *Escenas* > *CoppeliaSim* > *Control Trayectoria Circular CoppeliaSim-Matlab.*

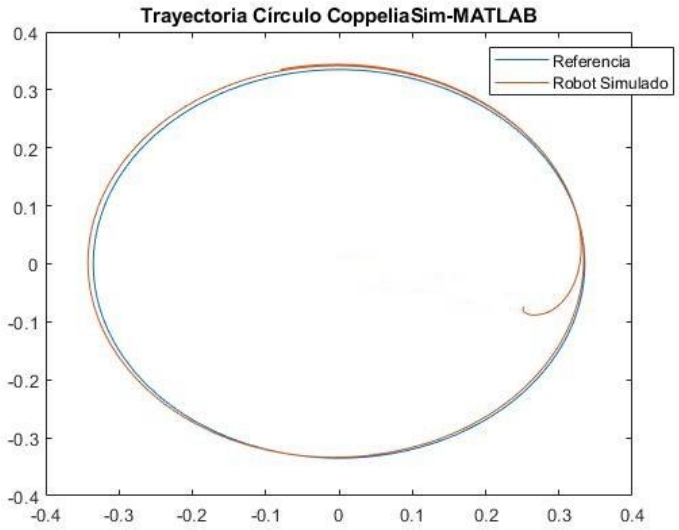


Figura 54. Gráfica resultado de la trayectoria circular en CoppeliaSim con la API remota de MATLAB.

Se puede apreciar una línea recta en la gráfica. Esto no significa que el robot haya avanzado linealmente durante un tiempo al principio. Es debido a que la primera coordenada registrada es 0,0 y la segunda es automáticamente el verdadero inicio del robot.

La ventaja de CoppeliaSim es que existe funciones para obtener la verdadera posición del robot (simxGetObjectPosition), es decir, se obtienen datos reales directamente de la trayectoria que está haciendo el robot durante la simulación y no posiciones resultados donde debería estar teóricamente como las obtenidas en RVW.

Aunque las gráficas de posiciones de referencia y del robot no son exactamente coaxiales por el motivo dicho anteriormente, y por ello los errores de posición no son determinantes, en la siguiente figura 55 se pueden ver estos errores para, al menos, ver sus dimensiones.

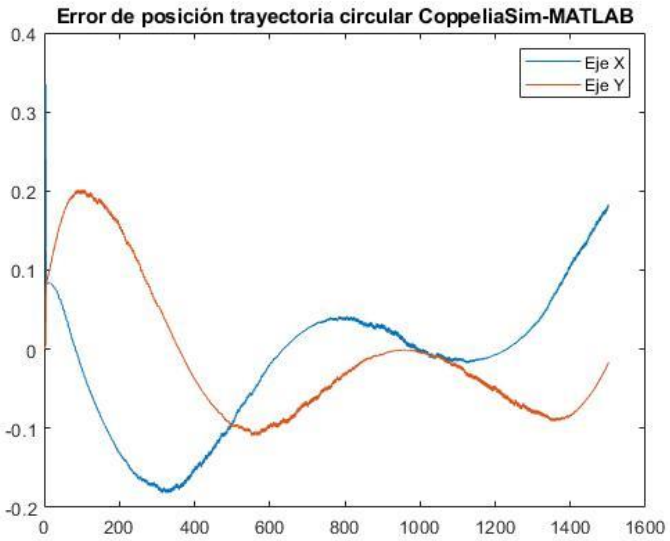


Figura 55.Gráfica del error de posición en los distintos ejes de la trayectoria en CoppeliaSim con API remota de MATLAB.

También se puede observar la gráfica del Error Cuadrático de la trayectoria en la figura 56, dando como resultado un Error Cuadrático Medio = 0.1065. El error es bastante bajo teniendo en cuenta que ni siquiera se ha sido capaz de centrar o ajustar al 100% ambas gráficas.

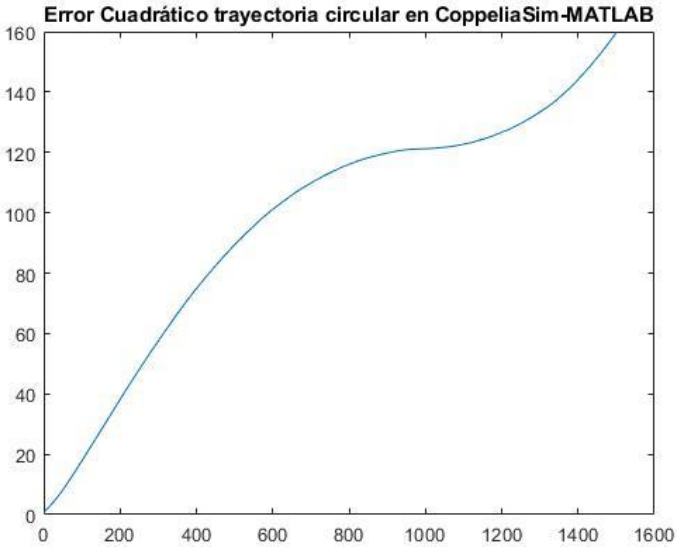
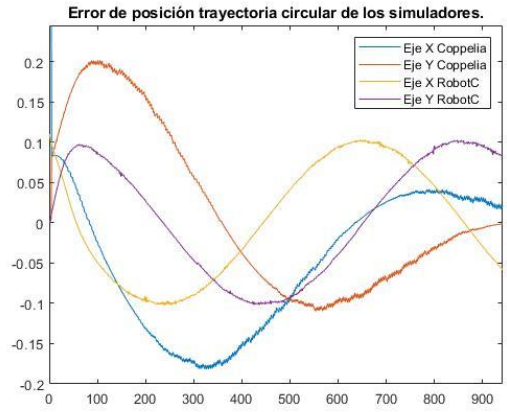


Figura 56. Gráfica del Error Cuadrático de la trayectoria en CoppeliaSim con API remota de MATLAB.

Como se puede observar en las gráficas el número de iteraciones empleadas para el cálculo del control de las trayectorias en Simulink es de 1500 iteraciones.

Si se compara entonces los resultados obtenidos en RobotC y CoppeliaSim queda:



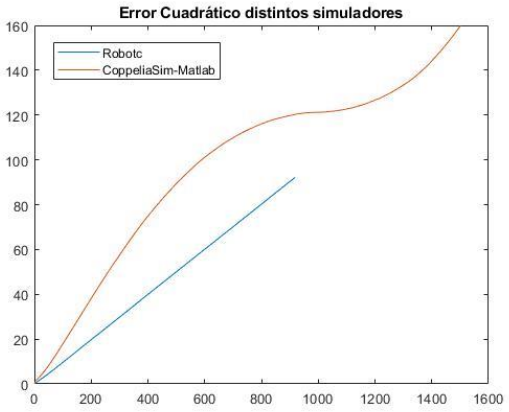


Figura 57. Gráfica de error de posición de la trayectoria en los distintos ejes de los distintos simuladores.
Figura 58. Gráfica de Error Cuadrático de la trayectoria en los distintos simuladores.

Aun teniendo en cuenta la imprecisión de la concentricidad en CoppeliaSim, se puede apreciar que no hay gran diferencia entre los errores de ambas posiciones, teniendo un error cuadrático medio de 0.1004 y 0.1065 en CoppeliaSim y RobotC respectivamente.

5.5.3 Robot Limpiador

Una vez creado el escenario como se explica en el apartado 5.3, se pone en práctica la aplicación de "Robot Limpiador". En el video "Robot Limpiador Coppelia-Matlab" de la carpeta compartida se puede ver como el robot realiza esta aplicación sin ningún tipo de problema. Como ya ha pasado con el Robot Limpiador programado en Lua, si se comparan las simulaciones de ambos simuladores, se ve que la respuesta, aunque tiene un positivo resultado, no es exactamente igual. Las razones son las mismas. La primera es que en RVW el robot viene ya configurado mientras que en el CoppeliaSim lo hemos configurado manualmente, teniendo así mas inexactitud en sus especificaciones técnicas. La segunda razón es la forma del robot. Mientras que en CoppeliaSim no le hemos puesto ningún brazo al robot, pues no hacía falta, en RVW tiene una especia de brazo delantero que deja pillada la bola sin posibilidad de quitarlo.

El archivo de programa usado se puede encontrar en la dirección *Programas > Coppelia-MATLAB > Control_trayectoria_circular* y el escenario en la dirección *Escenas > CoppeliaSim > Robot Limpiador*. Cualquier duda respecto a las funciones usadas para la programación se puede recurrir al ANEXO II, donde se explica cada una de ellas y su forma de uso.

CONCLUSIÓN.

Se ha desarrollado el control del robot móvil en los 2 entornos y el modelado de este en el único entorno que lo permitía. Se ha creado los escenarios y se han ido resolviendo las tareas.

Para la programación y simulación del EV3, robot usado en el laboratorio de la Universitat Politècnica de València, en Robot Virtual Worlds, simplemente se necesitaría crear el escenario de una manera muy simple si fuera necesario y programar el algoritmo en RobotC, una plataforma de programación bastante intuitiva que además se estudia en esa misma universidad. Sería pues, un procedimiento muy sencillo.

Sin embargo, no lo es tanto con CoppeliaSim. En este simulador, se ha tenido que diseñar de cero el EV3 desde un modelador externo e importarlo, para luego tener que terminar de modelarlo dentro de CoppeliaSim configurando y añadiéndole los sensores, creando sus formas puras, definiendo las propiedades dinámicas y jerarquizando las partes del robot. A continuación, se crea el escenario, se hace la conexión de la API remota entre CoppeliaSim y MATLAB pudiendo ya programar con comandos un tanto más complicados que los de RobotC. Aunque todos estos pasos no tendrían que hacerse en RVW, donde ya te viene un EV3 determinado, la ventaja que tiene CoppeliaSim sobre éste es que puedes modelar cualquier tipo de robot o cambiar la configuración de estos. En RVW se tiene solo 3 tipos de configuraciones del EV3 y no se puede cambiar la posición de ningún sensor, limitando la funcionalidad del mismo. Por otro lado, CoppeliaSim tiene más variedad de posibilidades a la hora de crear el escenario o importar obstáculos y la posibilidad de programar distintos objetos y usar varios robots a la vez. Sin embargo, la polivalencia a la hora de crear la escena y las múltiples propiedades que puede tener cada objeto, articulación o motor conlleva también más dificultad a la hora de comprender el funcionamiento de este simulador. La programación también es bastante menos intuitiva, por lo que se ha tenido que ir buscando la definición de cada función usada en las biblioteca de funciones de API remota de MATLAB que tienen en internet.

Después, una vez resueltas las tareas para ambos simuladores se puede comprobar, atendiendo a los resultados, que ambos simuladores responden y funcionan perfectamente e, incluso, cumplen objetivos con errores bastante bajos. La aplicación de “Robot Limpiador” es programada con éxito en los dos simuladores y viendo los videos se puede observar que en ambos se obtiene similar respuesta. RVW tiene una respuesta más precisa y parecida la que hizo el robot real, debido a que este simulador está echo específicamente para estos tipos de robots de modo que los robots que se simulan ya vienen determinados con las características reales de los mismos, a diferencia de CoppeliaSim donde el robot se ha modelado personalmente, siendo así menos exacto. Sin embargo, esta diferencia no es significativa, con lo cual se quedan igual de validos ambos simuladores en cuanto a resultados de este tipo de aplicaciones.

Respecto al control de trayectoria, también puede verse en los resultados de los distintos simuladores que ambos la siguen a la perfección. Como se ha comentado en su respectivo apartado, en CoppeliaSim además se ha tenido que ajustar las circunferencias manualmente

para hacerlas coaxiales en la medida de los posible, por eso los errores de posición obtenidos son menos precisos. Por otro lado, el robot recorre con éxito una circunferencia con los radios propuestos, como se puede ver en las gráficas y videos mencionados en sus respectivos apartados. Cabe destacar que en CoppeliaSim existen funciones con las que se han podido obtener la posición real del robot durante la simulación, mientras que en RobotC no existe tal función y la posición obtenida ha sido mediante los cálculos del control cinemático de configuración diferencial. En las figuras 57 y 58 se puede comparar los errores entre ambos simuladores.

Entonces, si comparando resultados ambos simuladores obtienen respuestas similares e igual de exitosas, ¿cuándo se recomendaría usar uno u otro? Como ya se puede deducir mirando en la página anterior, dónde se ha resumido la programación y simulación de ambos entornos, y como se ha comentado en varias ocasiones, la principal diferencia entre ellos es la complejidad y polivalencia. La diferencia se puede notar tan solo viendo el tamaño de los apartados de los distintos simuladores.

Con todo esto, se puede llegar a la sencilla conclusión de cuando se recomienda usar uno u otro: depende de la complejidad de la tarea. Siempre que la tarea o ejercicio que se busque sea sencilla y entre en los límites de programación de RVW y RobotC se recomienda usar esta por su comodidad, facilidad, rapidez y las precisas características del EV3 que viene ya determinado. Cualquier trabajo o estudio que requiera más complejidad se recomienda usar CoppeliaSim, donde su utilidad es prácticamente ilimitada, pudiendo crear cualquier escena, tarea y siendo capaz de obtener cualquier dato.

Cabe señalar que al comienzo del TFG se probaron otras plataformas de programación y simulación del EV3 como *Virtual Robotics Toolkit* y *Lego Mindstorms EV3 home edition* y otros modeladores de LEGO como *LEGO digital designer*, pero estos fueron rápidamente descartados pues no servían ni cumplían con los objetivos buscados.

En cuanto a modeladores externos, se recomienda usar el empleado en este trabajo, LeoCAD, siempre que se quiera modelar un modelo de LEGO, pudiéndose emplear otros modeladores como AutoDesk Inventor.

Por último, queda nombrar que una de las finalidades de este trabajo es que se pueda partir de él como guía para la simulación de futuros trabajos, apoyándose y ayudándose de él para facilitar el entendimiento y uso de estos distintos simuladores, dando así una oportunidad de resolver los trabajos a distancias.

ANEXO I.

Desarrollo de los resultados experimentales del Control Cinemático de trayectorias.

Simulink.

La figura 59 muestra el esquema general de Simulink para el control del círculo. Se puede ver que las entradas son las posiciones y velocidades de referencia en *x* y en *y*, dadas por la trayectoria que queremos seguir. Las salidas del sistema serán las posiciones *x*, *y* y el ángulo theta del robot real Lego.

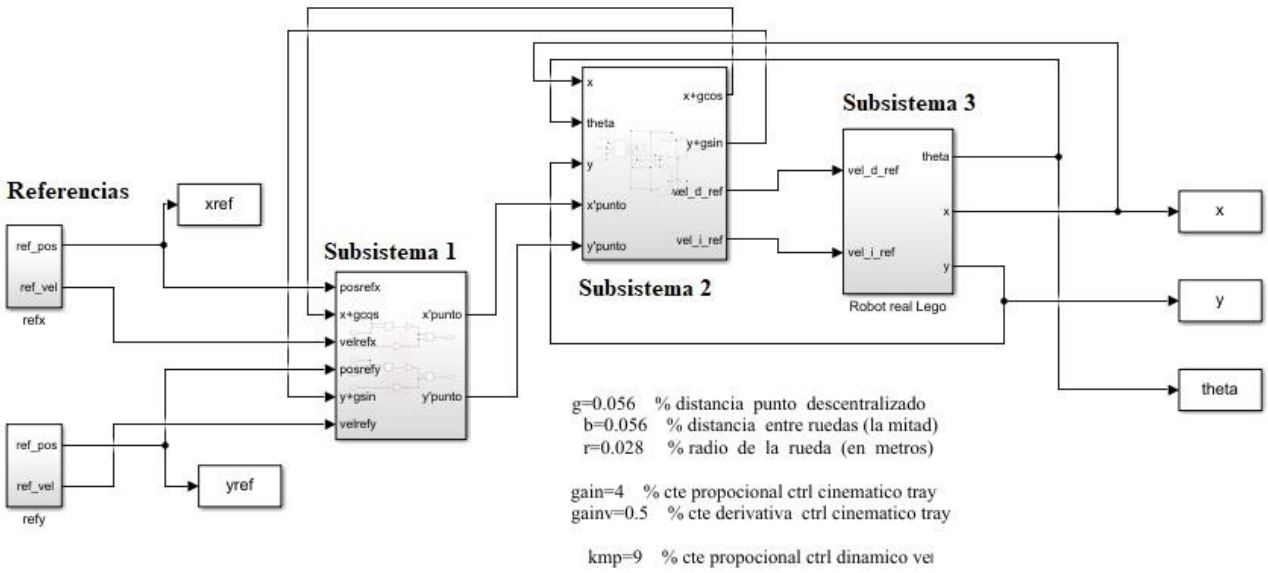


Figura 59. Esquema general del control de la trayectoria circular en Simulink.

Las referencias se crean a partir de la ecuación de la trayectoria que generaría un círculo. En esta gráfica que generará Simulink el radio es ligeramente inferior al real (r = 335mm).

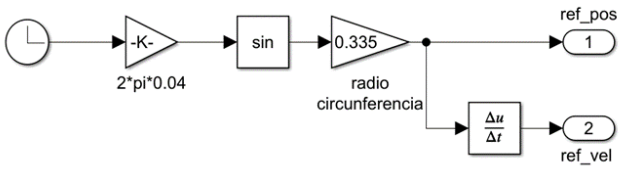


Figura 60. Esquema Simulink de la referencia en Y de la trayectoria del círculo.

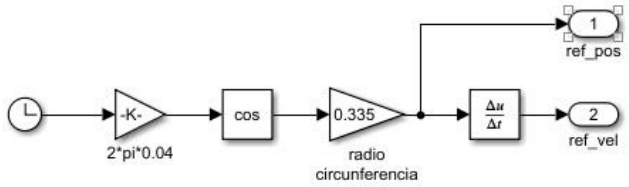


Figura 61. Esquema Simulink de la referencia X de la trayectoria del círculo.

En cuanto a los subsistemas, cada subsistema presenta una parte correspondiente a al esquema de control de posición del punto descentralizado.

En el subsistema 1 sucede el control cinemático. De hecho, este esquema sigue las mismas operaciones que la función de control cinemático antes definida. Se pasa de las posición y velocidad de referencia, junto con la posición del robot, a la velocidad del punto descentralizado. El esquema es el siguiente:

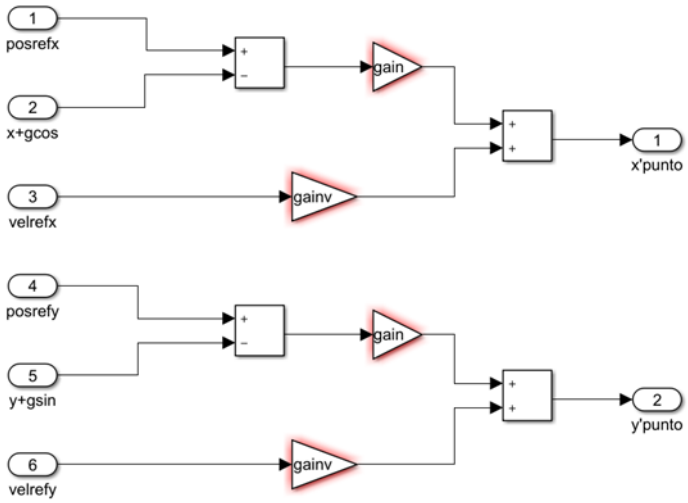


Figura 62. Esquema del control cinemático en Simulink

El subsistema 2 equivale al control de cinemática inversa. Se parte del ángulo theta y la velocidad del punto descentralizado y se obtiene como salida las velocidades de referencia de la rueda izquierda y derecha. También se aprovecha el ángulo para hacer una operación intermedia donde se obtienen las variables y+gsin(θ) y x+gcos(θ) empleadas en el anterior control cinemático.

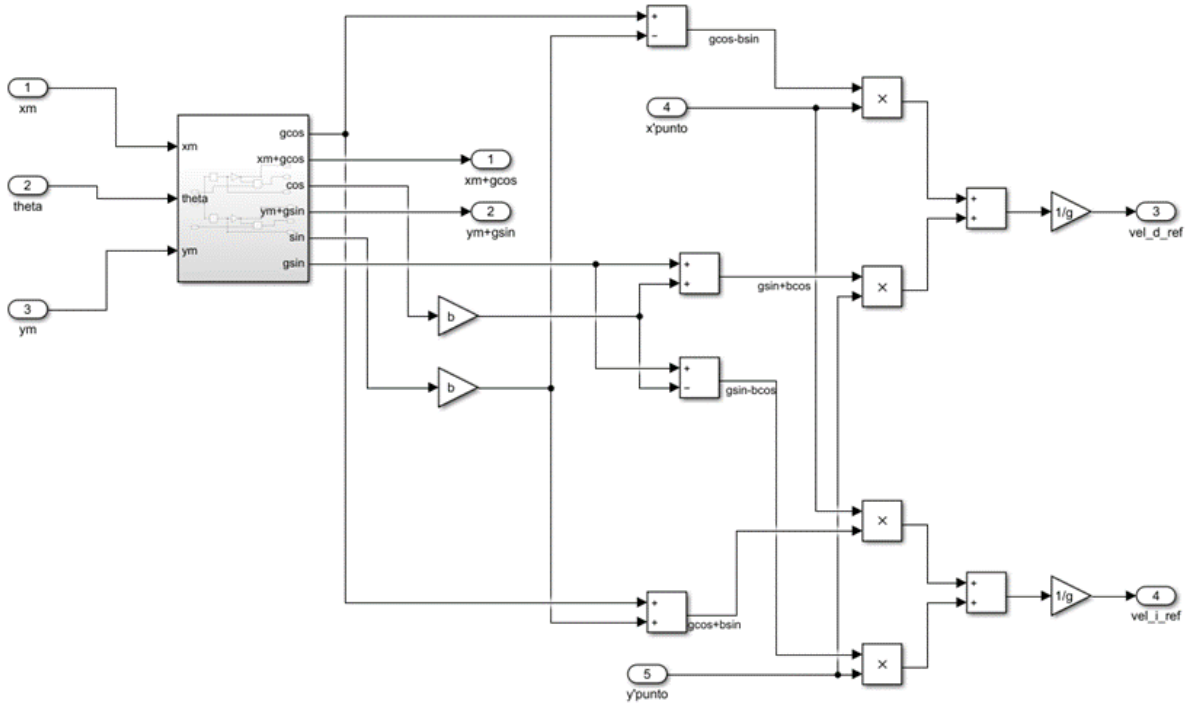


Figura 63. Esquema Cinemática Inversa en Simulink

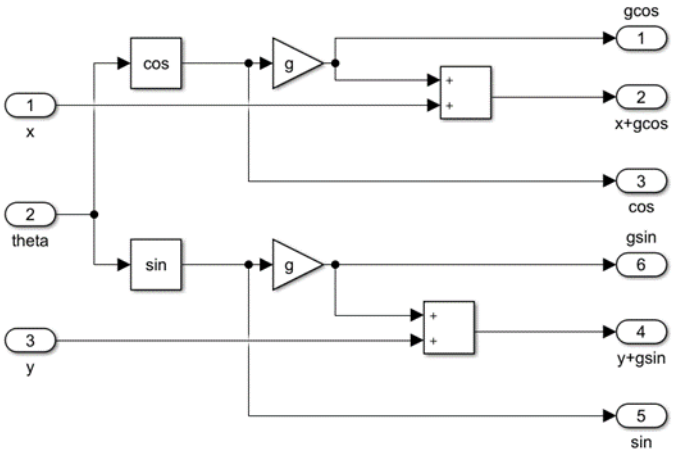


Figura 64. Subsistema dentro del Control de Cinemática Inversa en Simulink

El subsistema 3 está formado por el control dinámico del robot y por la cinemática directa del robot. Tiene como parámetros de entrada las velocidades angulares de referencia obtenidas de dividir la velocidad lineal entre el radio y se obtiene tras el control dinámico las velocidades angulares de las ruedas. Después, vuelve a multiplicar la velocidad angular por el radio y obtiene la velocidad lineal de las ruedas, que es, a su vez, el parámetro de entrada de la cinemática directa para obtener la posición y orientación del robot como se indica en el esquema.

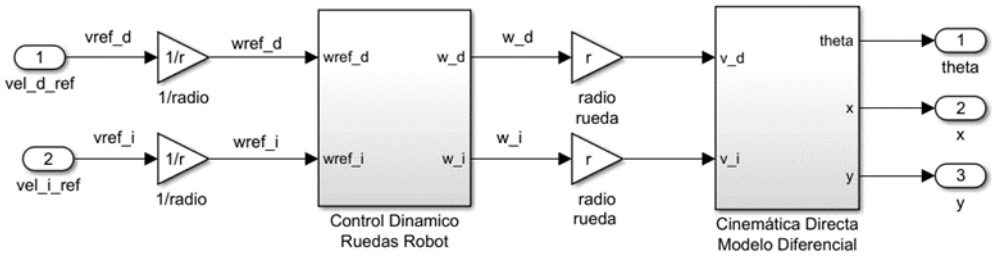


Figura 65. Subsistema 3del control cinemático de Simulink

Para el control dinámico se propone el siguiente control que sigue esquema del control dinámico añadiéndole la función de transferencia del motor.

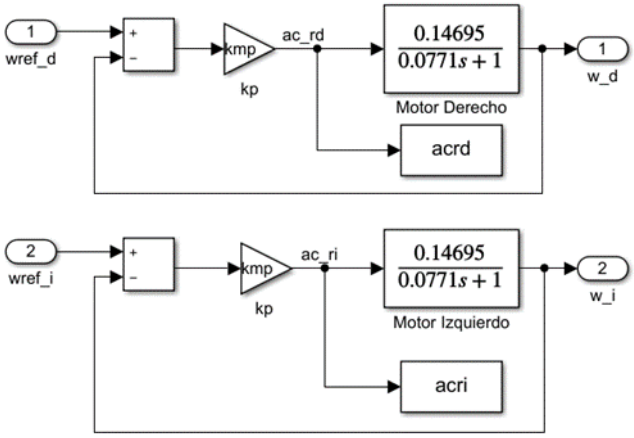


Figura 66. Esquema del Control Dinámico en Simulink

Mientras que el esquema de control directo sigue la ecuación de cinemática directa del robot diferencial antes definida.

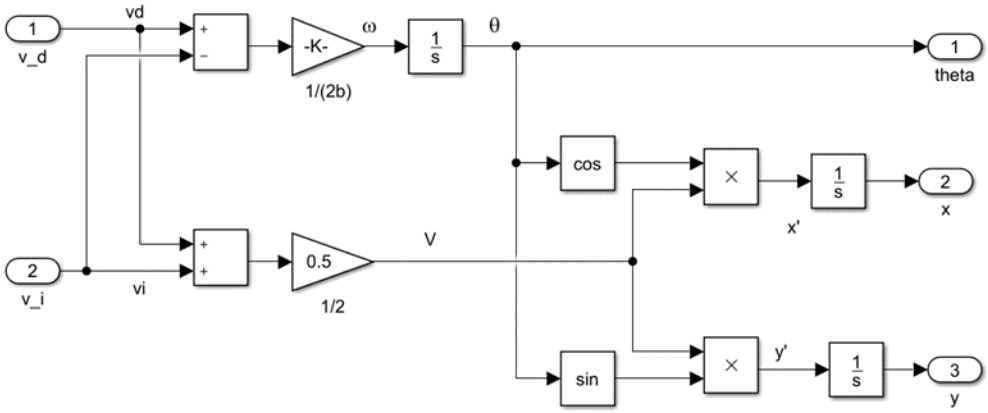


Figura 67. Esquema del Control Cinemático directo del EV3 en Simulink.

Cerrando con este el control del robot. Las variables no definidas en el esquema, como las ganancias, se definen y se ejecutan en el script de Matlab antes de ejecutar Simulink.

gain=4
gainv=0.5
kmp=9
b=0.056
r=0.028
g=0.056

Solo falta obtener la gráfica desde Matlab con la función *plot (x, y)*, seguido de *hold on* y *plot (xref, yref)*. El resultado es el siguiente:

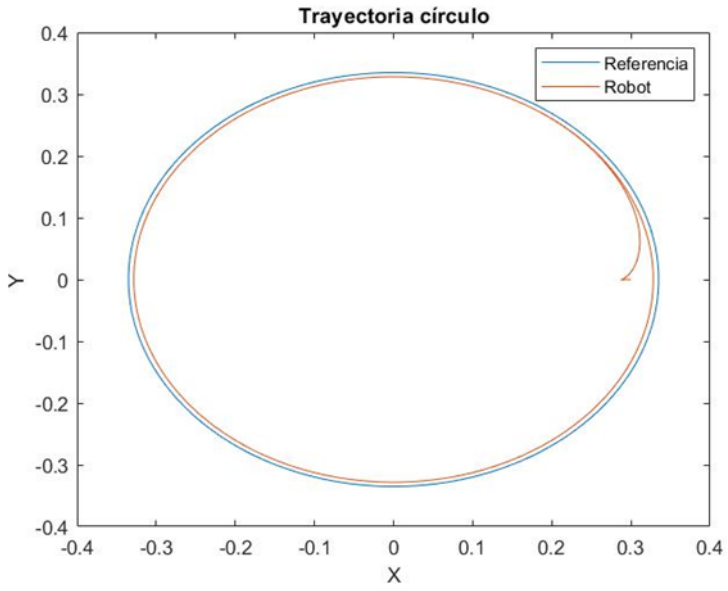
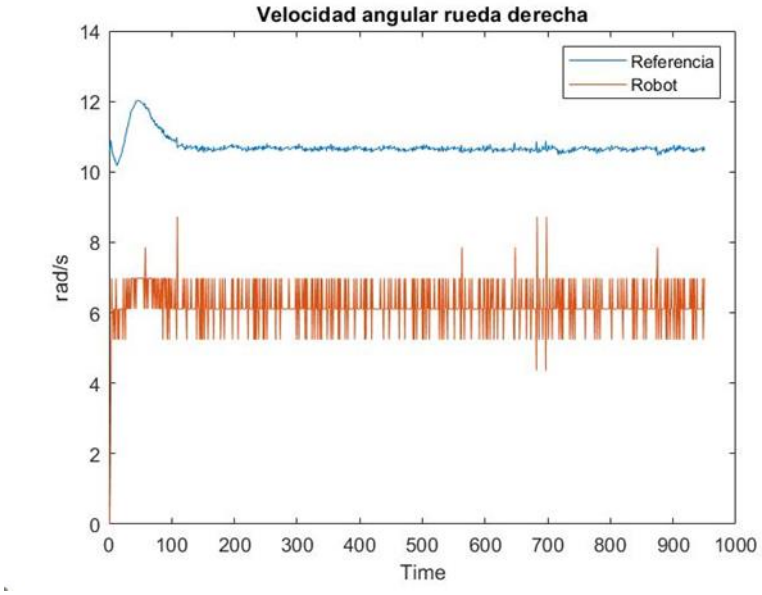
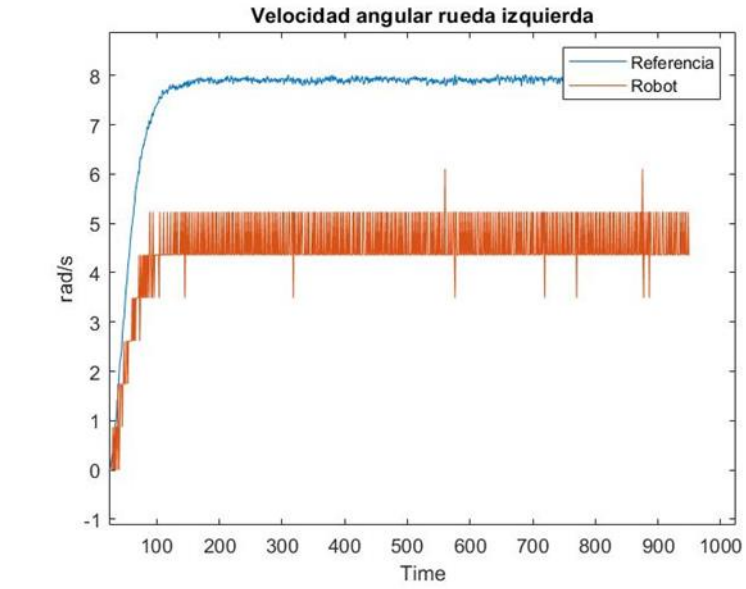
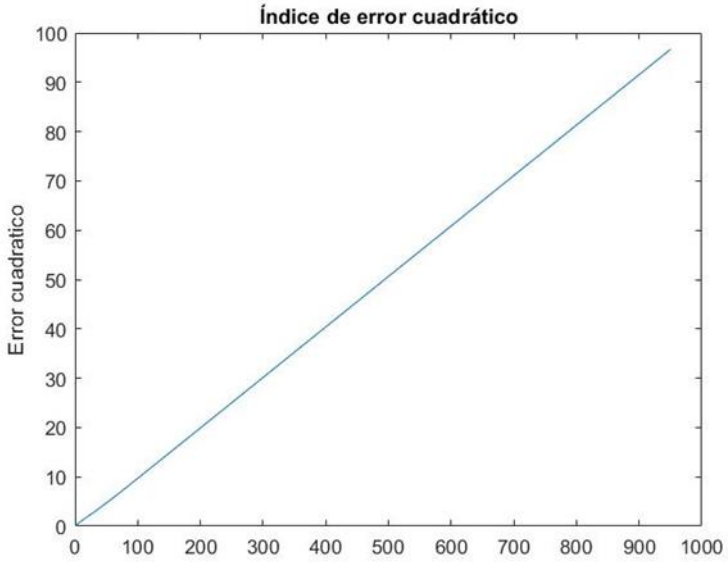


Figura 68. Gráfica resultado de la trayectoria circular en Simulink


Velocidad angular rueda derecha
Referencia
Robot
rad/s
Time
0
2
4
6
8
10
12
14
0
100
200
300
400
500
600
700
800
900
1000


Velocidad angular rueda izquierda
Referencia
Robot
rad/s
Time
-1
0
1
2
3
4
5
6
7
8
100
200
300
400
500
600
700
800
900
1000

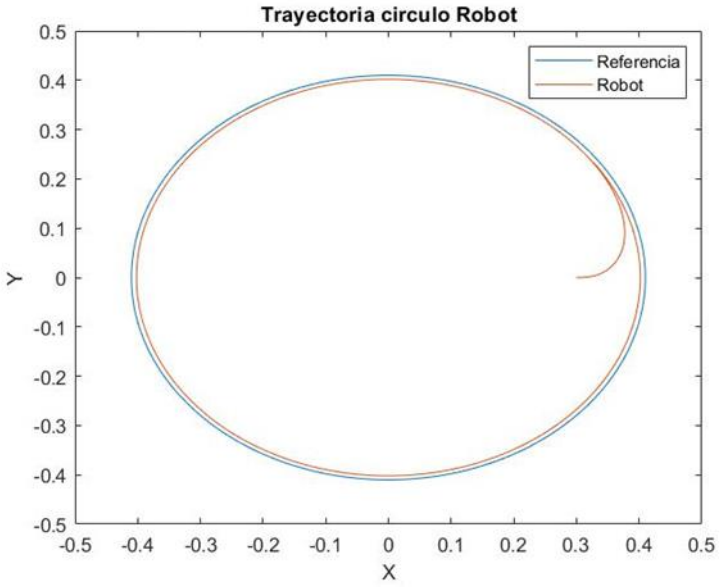


Programación real con RobotC.

Para la programación del EV3 en RobotC se han generado los ficheros “gen_trayect.h” y “pdescen_EV3.c” implementados a partir de los esquemas de Simulink que pueden encontrarse en la dirección *Laboratorio > RobotC.* Deberían estar guardados en la misma carpeta debido a que al ejecutar “pdescen_EV3.c”, en una de las instrucciones llama a “gen_trayect.h”.

El archivo “gen_trayect.h” es una función que genera la trayectoria que debe seguir el robot, dando la opción de elegir cual quieres que sea (circular o cuadrada) en la pantalla del mismo robot. Por otro lado, el archivo “pdescen_EV3.c” genera el control cinemático por punto descentralizado.

Una vez compilado y descargado el programa al robot EV3, este hizo la trayectoria generando un fichero con cada una de sus posiciones a tiempo real. Generando la gráfica de la trayectoria en Matlab gracias a este fichero se obtiene como resultado:


Trayectoria circulo Robot
Referencia
Robot
Y
X
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.5
-0.4
-0.3
-0.2
-0.1
0
0.1
0.2
0.3
0.4
0.5

ANEXO II.

DEFINICIÓN DE LAS FUNCIONES USADAS.

Funciones usadas de la API regular LUA.

sim.addStatusBarMessage

- Función:

sim.addStatusbarMessage(string message)

Añade un mensaje en la ventana de estado. El mensaje va entre comillas simples altas y se puede concatenar con variables poniendo dos puntos entre ellas como se muestra en el ejemolo.

- Parámetros de entrada:
  - **message**: mensaje que se quiere que aparezca entre apostrofes.
- Ejemplo en programa:

```
sim.addStatusbarMessage('Dist.:'..distance..'Luz:'..grey..'result:'..
result);
```

sim.getObjectHandle

- Función:

number handle=sim,getObjectHandle(string objectName)

Esta función se tiene que usar siempre al principio del programa para obtener los manejadores de los objetos que se van a controlar o llamar en las siguientes funciones.

- Parámetros de entrada:
  - **objectName**: nombre que aparece en la jerarquía de escena del objeto del cual se quiere obtener el manejador. Entre apostrofes (‘’).
- Parámetros de salida:
  - **handle**. Variable con el número del manejador del objeto. Se recomienda un nombre intuitivo que recuerde al objeto como en el ejemplo.
- Ejemplo en programa:

```
motor_izqdo=sim.getObjectHandle('Motor_B');
```

simxGetObjectPosition

- Función:

[number returnCode, array position] = simxGetObjectPosition (number clientID, number objectHandle, number relativeToObjectHandle, number operationMode)

Obtiene la posición de un objeto.

- Parámetros de entrada:
  - **ClientID**.
  - **objectHandle**. Nombre del manejador del objeto.
  - **relativeToObjectHandle**. Indica la referencia a partir de la cual queremos la posición. Usa -1 para obtener la posición absoluta, sim_handle_parent para recuperar la posición relativa al elemento primario o el nombre del manejador de un objeto que se quiera usar como referencia.
  - **operationMode**. Se recomienda usar *vrep.simx_opmode_streaming* para la primera llamada a la función y *vrep.simx_opmode_buffer* para las siguientes. Esta es la razón por la que en los programas de API remota de MATLAB del trabajo se ha llamado a este tipo de funciones al menos una vez antes del bucle principal.
- Parámetros de salida:
  - **returnCode**.
  - **position**. Matriz de 3 valores representando la posición respecto a los 3 ejes (x y z).
- Ejemplo en programa:

```
[returnCode,position]=vrep.simxGetObjectPosition(clientID,robot,dummy,
vrep.simx_opmode_streaming);
```

Sim.getVisionSensorImage

- Función:

table/string imageBuffer = sim.getVisionSensorImage (number sensorHandle, number posX, number posY, number sizeX, number sizeY, number returnType)

Lee el sensor de color.

- Parámetros de entrada:
  - **sensorHandle**. Nombre del manejador del sensor de color que se desea leer. Puede combinarse con *sim.handleflag_greyscale* si desea recuperar el equivalente en **escala de grises**, como se muestra en el ejemplo.
  - **posX / posY.** Posición de la porción de imagen que se recupera. Es 0 por defecto si no se escribe nada.

- **sizeX / sizeY.** Tamaño de la porción de imagen que se recupera. Es 0 por defecto si no se escribe nada, lo que significa que se recuperaría la imagen completa.
- **returnType.** El tipo de valor devuelto. 0 devuelve una table llena con valores de RGB con un rango entre 0-1. 1 devuelve una table con valores RGB con un rango entre 0-255.

- Parámetros de salida:
  - **imageBuffer**. Una matriz con los valores RGB de la imagen de tamaño sizeX*sizeY*3. En el caso de una recuperación de imágenes en escala de grises, la tabla de imágenes contendrá valores grises. Para acceder al valor de un solo bit de la imagen, como conviene en el caso del programa del TFG, podría hacerse como en el ejemplo.
- Ejemplo en programa:

```
image=sim.getVisionSensorImage(sensorEV3_Color+sim.handleflag_greyscale);
grey=image[1];
```

sim.readProximitySensor

- Función:

number result, number distance, table_3 detectedPoint, number detectedObjectHandle, table_3 detectedSurfaceNormalVector=sim.readProximitySensor(number sensorHandle).

Lee el sensor de proximidad.

- Parametros de entrada:
  - **sensorHandle**. Nombre del manejador del sensor de proximidad que se desea leer.
- Parámetros de salida:
  - **result.** Estado de detección. 0 si no se detecta nada o 1 si ha detectado algún obstáculo.
  - **distance.** Distancia al punto detectado.
  - **detectedPoint.** Tabla de 3 números que indican las coordenadas relativas del punto detectado.
  - **DetectObjectHandle.** Manejador del objeto que se detecta.
  - **DetectadoSurfaceNormalVector.** Vector normalizado de la superficie detectada. Relativo al marco de referencia del sensor.
- Ejemplo en programa:

```
result,distance,detectedPoint,detectedObjectHandle,detectedSurfaceNormalVector=si
m.readProximitySensor(sensorEV3_ultrasonic);
    if result~=1 then
    distance=2.5;
    end
```

sim.setJointTargetVelocity

- Función:

sim.setJointTargetVelocity(number objectHandle, number targetVelocity)

Establece la velocidad de una articulación. Que sea angular (deg/s) o lineal (m/s) depende del tipo de articulación.

- Parámetros de entrada:
  - **objectHandle**. Nombre del manejador de la articulación.
  - **targetVelocity**. Velocidad que se desea establecer a la articulación.
- Ejemplo en programa:

```
sim.setJointTargetVelocity(motor_izqdo,10);
```

os.clock() (sleep)

En Lua no existen funciones de espera como serían *sleep* (C++) o *pause* (MATLAB). Por ello recurrimos a una serie de comandos con la finalidad de simular estas funciones. Se crea y se iguala una variable a la función *os.clock()*. Esta función obtiene el tiempo que se lleva de simulación el programa. Luego se mantiene en espera con un bucle *while* durante el tiempo que se defina, en segundos. A continuación, se ve un ejemplo usado en los programas.

```
local x2 = os.clock()
while ((os.clock()-x2) < 2) do end.
```

Funciones usadas de la API remota de MATLAB.

Para ayudar a entender las funciones usadas en este trabajo, aquí se explican su comando. Todas las funciones **deberían ir seguidas del prefijo que indica la conexión API remota**, en el caso de este trabajo "vrep.". Si se quisiera saber más sobre estas funciones remotas hay que acudir a la referencia (35).

Además, en todas las funciones hay una serie de parámetros que se repiten:

- **number clientID**. Parámetro de entrada. Variable del número de cliente obtenida al realizar la conexión CoppeliaSim-MATLAB. En el trabajo tiene siempre tiene el nombre de **clientID** y **se usa en todas las funciones** de la API remota.
- **number returnCode.** Parámetro de salida. Variable de un número de retorno que da la función. Puedes ver lo que indica el número de retorno en la referencia (36). En el trabajo tiene siempre tiene el nombre de **returnCode** y **se usa en todas las funciones** de la API remota.
- **number operationMode**. Parámetro de entrada. La función del modo de operación. Puedes encontrar estas funciones en la referencia (36). Como el resto de funciones debe ir acompañada del prefijo "vrep.-".

Las funciones usadas en los programas del trabajo ordenadas por orden alfabético son:

simxAddStatusBarMessage

- Función:

[number returnCode]=vrep.simxAddStatusbarMessage(number clientID, string message, number operationMode)

Añade un mensaje en la ventana de estado.

- Parámetros de entrada:
  - number **clientID**.
  - string **message**: mensaje que se quiere que aparezca entre apostrofes.
  - number **operationMode**. Para este comando se recomienda *vrep.simx_opmode_blocking*.
- Parámetros de salida:
  - number **returnCode**.
- Ejemplo en programa:

```
[returnCode]=vrep.simxAddStatusbarMessage(clientID,'comunicacion con MATLAB iniciada',vrep.simx_opmode_blocking);
```

simxGetObjectHandle

- Función:

[number returnCode,number handle]=vrep.simxGetObjectHandle(number clientID, string objectName, number operationMode)

Esta función se tiene que usar siempre al principio del programa para obtener los manejadores de los objetos que se van a controlar o llamar en las siguientes funciones.

- Parámetros de entrada:
  - **clientID**.
  - **objectName**: nombre que aparece en la jerarquía de escena del objeto del cual se quiere obtener el manejador. Entre apostrofes (‘’).
  - **operationMode**. Para este comando se recomienda *vrep.simx_opmode_blocking*.
- Parámetros de salida:
  - **returnCode**.
  - **handle**. Variable con el número del manejador del objeto. Se recomienda un nombre intuitivo que recuerde al objeto como en el ejemplo.
- Ejemplo en programa:

```
[returnCode,motor_izqdo]=vrep.simxGetObjectHandle(clientID,'Motor_B',
vrep.simx_opmode_blocking);
```

simxGetObjectPosition

- Función:

[number returnCode, array position] = simxGetObjectPosition (number clientID, number objectHandle, number relativeToObjectHandle, number operationMode)

Obtiene la posición de un objeto.

- Parámetros de entrada:
  - **ClientID**.
  - **objectHandle**. Nombre del manejador del objeto.
  - **relativeToObjectHandle**. Indica la referencia a partir de la cual queremos la posición. Usa -1 para obtener la posición absoluta, sim_handle_parent para recuperar la posición relativa al elemento primario o el nombre del manejador de un objeto que se quiera usar como referencia.
  - **operationMode**. Se recomienda usar *vrep.simx_opmode_streaming* para la primera llamada a la función y *vrep.simx_opmode_buffer* para las siguientes. Esta es la razón por la que en los programas de API remota de MATLAB del trabajo se ha llamado a este tipo de funciones al menos una vez antes del bucle principal.

- Parámetros de salida:
  - **returnCode**.
  - **position**. Matriz de 3 valores representando la posición respecto a los 3 ejes (x y z).
- Ejemplo en programa:

```
[returnCode,position]=vrep.simxGetObjectPosition(clientID,robot,dummy,
vrep.simx_opmode_streaming);
```

simxGetVisionSensorImage2

- Función:

[number returnCode, array resolution, matrix image] = vrep.simxGetVisionSensorImage2(number clientID, number sensorHandle, number options, number operationMode)

Lee el sensor de color.

- Parámetros de entrada:
  - **clientID**
  - **sensorHandle**. Nombre del manejador del sensor de color que se desea leer.
  - **options**. Si está activado el bit 0, es decir, si ponemos un 1, en cada pixel de la imagen lee un byte en escala de grises. De lo contrario lee 3 bytes en rgb.
  - **operationMode**. Se recomienda usar *vrep.simx_opmode_streaming* para la primera llamada a la función y *vrep.simx_opmode_buffer* para las siguientes. Esta es la razón por la que en los programas de API remota de MATLAB del trabajo se ha llamado a este tipo de funciones al menos una vez antes del bucle principal.
- Parámetros de salida:
  - **returnCode**.
  - **resolution**. Una matriz de dos valores numéricos que representan la resolución de la imagen.
  - **image**. Una matriz con los datos de la imagen. Los valores están en el rango de 0-255. Por ello para simular la lectura del sensor del EV3 se hace una media de todos los datos como se verá en el ejemplo.
- Ejemplo en programa:

```
[returnCode,resolution,image]=vrep.simxGetVisionSensorImage2(clientID,
sensorEV3_Color, 1, vrep.simx_opmode_buffer);

M=mean(image);
GREY=mean(M);
```

simxReadProximitySensor

- Función:

[number returnCode, boolean detectionState, array detectedPoint, number detectedObjectHandle, array detectedSurfaceNormalVector]=vrep.simxReadProximitySensor (number clientID, number sensorHandle, number operationMode)

Lee el sensor de proximidad.

- Parámetros de entrada:
  - **ClientID**.
  - **sensorHandle**. Nombre del manejador del sensor de proximidad que se desea leer.
  - **operationMode**. Se recomienda usar *vrep.simx_opmode_streaming* para la primera llamada a la función y *vrep.simx_opmode_buffer* para las siguientes. Esta es la razón por la que en los programas de API remota de MATLAB del trabajo se ha llamado a este tipo de funciones al menos una vez antes del bucle principal.
- Parámetros de salida:
  - **ReturnCode**.
  - **detectionState**. Nos dice si el sensor está detectando algo. Si tu variable de detectionState es falsa (0) significa que no está detectando nada.
  - **detectedPoint**. Las coordenadas del punto detectado en relación con el sensor. Es un vector de tres números con la distancia en los distintos ejes (x y z). Además, si el sensor no detecta ningún obstáculo la distancia que se lee es 0. El comando que se usa en el programa para detectar la distancia e imponer la máxima distancia de 2.5m al sensor cuando no esté detectando nada se muestra en el ejemplo.
  - **detectedObjectHandle**. Variable con el número del manejador del objeto detectado.
  - **detectedSurfaceNormalVector**. Vector normalizado de la superficie detectada. Relativo al marco de referencia del sensor.
- Ejemplo en programa:

```
[returnCode, detectado, detectedPoint, detectedObjectHandle,
detectedSurfaceNormalVector] = vrep.simxReadProximitySensor(clientID,
sensorEV3_ultrasonic, vrep.simx_opmode_buffer);
distancia=sqrt(detectedPoint(1)*detectedPoint(1)+detectedPoint(2)*detectedPoint(2
));
if distancia==0
   distancia=2.5;
end
disp(distancia);
```

simxSetJointTargetVelocity

- Función:

[number returnCode] = vrep.simxSetJointTargetVelocity (number clientID, number jointHandle, number targetVelocity, number operationMode)

Establece la velocidad de una articulación. Que sea angular (deg/s) o lineal (m/s) depende del tipo de articulación.

- Parámetros de entrada:
  - **ClientID**.
  - **jointHandle**. Nombre del manejador de la articulación.
  - **targetVelocity**. Velocidad que se desea establecer a la articulación.
  - **operationMode**. Se recomienda usar *vrep.simx_opmode_oneshot.*
- Parámetros de salida:
  - **returnCode**.
- Ejemplo en programa:

```
[returnCode]=vrep.simxSetJointTargetVelocity(clientID,motor_izqdo,10,
vrep.simx_opmode_oneshot);
```

REFERENCIAS.

(1) Milenio.com https://www.milenio.com/opinion/varios-autores/universidad-politecnica-de-tulancingo/la-importancia-de-la-robotica
(2) Revistaderobots.com https://revistaderobots.com/robots-y-robotica/que-es-la-robotica/
(3) Wikipedia. Robótica. https://es.wikipedia.org/wiki/Robótica
(4) Artículo Industria 4.0 https://industria40.me/blog/robots-en-la-industria-para-2020/
(5) Wikipedia. Robótica educativa. https://es.wikipedia.org/wiki/Robótica_educativa#Evolución
(6) Rae. Robot. https://dle.rae.es/robot
(7) Wikipedia. Robótica móvil. https://es.wikipedia.org/wiki/Robot_móvil
(8) ROBOTC 4.X for Mindstorms. http://www.robotc.net/download/lego/
(9) Robotc.net. http://www.robotc.net/
(10) Robot Virtual Worlds. http://www.robotvirtualworlds.com/download/?_ga=2.81386647.1071578782.1594232206-764946863.1586012991
(11) CoppeliaSim. https://www.coppeliarobotics.com/
(12) MATLAB. https://www.mathworks.com/products/matlab.html
(13) LeoCAD. https://www.leocad.org/
(14) LDraw. https://www.ldraw.org/
(15) Ladrillo EV3 LEGO https://www.lego.com/es-es/product/ev3-intelligent-brick-45500
(16) Instrucciones construcción robot. https://education.lego.com/en-us/support/mindstorms-ev3/building-instructions
(17) Lego.com. Pack EV3 con LEGO technic. https://www.lego.com/es-es/product/lego-mindstorms-ev3-31313
(18) Diseño de dispositivos para rehabilitación y órtesis. Capítulo de robot paralelos. Mary Vergara Paredes, Miguel Díaz Rodríguez, Francklin Rivas Echeverría y Magda Restrepo Moná. 2017.
(19) Electric Bricks, Braitenberg. http://blog.electricbricks.com/2010/04/vehiculos-de-braitenberg-en-robotc/
(20) Scielo. Braitenberg. http://www.scielo.org.bo/scielo.php?script=sci_arttext&pid=S2071-081X2016000200009
(21) Import object http://www.robotvirtualworlds.com/model-importer/
(22) Birckset https://brickset.com/parts
(23) YouTube Alberto https://www.youtube.com/watch?v=t81g9hbKG5A&t=48s
(24) Guia de uso de Lego Mindstorms EV3 https://www.lego.com/cdn/cs/set/assets/blt98e28d1c377e9a45/User_Guide_LEGO_MINDSTORMS_EV3_11_All_ES.pdf
(25) LDraw Documentación https://www.ldraw.org/article/218.html

(26) Forum CoppeliaSim. https://forum.coppeliarobotics.com/viewtopic.php?t=2545
(27) CoppeliaSim Guía Models. https://www.coppeliarobotics.com/helpFiles/en/models.htm
(28) Coppelia robotics. Scripts. https://www.coppeliarobotics.com/helpFiles/en/scripts.htm
(29) https://blog.makeitreal.camp/lenguajes-compilados-e-interpretados/
(30) Manual Lua. https://www.lua.org/manual/5.1/es/manual.html
(31) Coppelia robotics. API regular. https://www.coppeliarobotics.com/helpFiles/en/apiOverview.htm
(32) Coppelia robotics. Funciones Lua ordenadas alfabéticamente https://www.coppeliarobotics.com/helpFiles/en/apiFunctionListAlphabetical.htm
(33) Coppelia robotics. Funciones Lua ordenadas por categoría https://www.coppeliarobotics.com/helpFiles/en/apiFunctionListCategory.htm
(34) Coppelia robotics. Child scripts https://www.coppeliarobotics.com/helpFiles/en/childScripts.htm
(35) Coppelia robotics. Biblioteca API remote MATLAB. https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsMatlab.htm
(36) Coppelia robotics. Constantes API remota https://www.coppeliarobotics.com/helpFiles/en/remoteApiConstants.htm#functionErrorCodes
(37) Alberto Martín Domínguez. 2016. Modelado y Simulación de un robot LEGO Mindstorms EV3 mediante V-REP y Matlab.
(38) Pablo Parodi Félix. Análisis de VREP como herramienta de simulación para Aerostack.
(39) Leopoldo Armesto. 2020. Sensores de proximidad.

TRABAJO FIN DE GRADO EN INGENIERÍA DE LAS TECNOLOGÍAS INDUSTRIALES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

PRESUPUESTO.

ESTUDIO DE ENTORNOS VIRTUALES PARA ROBOTS MÓVILES. DESARROLLO DE APLICACIONES DE CONTROL DE ROBOTS MÓVILES CON CONFIGURACIÓN DIFERENCIAL.

Índice del presupuesto.

Índice de tablas.

1. Necesidad del presupuesto.

En este documento se estudia el presupuesto del proyecto "Estudio de entornos virtuales para robots móviles. Desarrollo de aplicaciones de control de robots móviles con configuración diferencial". El trabajo, a pesar de ser un trabajo respecto a simuladores y la mayoría de lo desarrollado ha sido virtualmente, debe incluir un presupuesto como cualquier otro proyecto de ingeniería. Además, el objetivo principal del trabajo era dar la oportunidad de seguir futuros trabajos de robótica, sobre todo en el ámbito educativo, por ello es importante también tener en cuenta el precio de esto.

Para su realización ha sido conveniente mirar las "Recomendaciones en la Elaboración de Presupuestos en Actividades de I+D+I" de la última revisión actual, la del 2018, de la UPV de acuerdo con el art. 83 de la LOU

2. Contenido del presupuesto.

El contenido del presupuesto ha sido divido en tres apartados: mano de obra, maquinaria y materiales fungibles.

2.1 Mano de obra.

En primer lugar, se ha hecho una estimación de las horas empleadas por el graduado en ingeniería en Tecnologías Industriales en la UPV en cada una de las tareas del proyecto.

Actividad	Tiempo (h)
Montaje del Robot LEGO Mindstorms EV3.	3
Estudio de las funciones y desarrollo del control cinemático de trayectorias para Simulink en el laboratorio. Obtención de resultados y creación de gráficas.	8
Estudio de RobotC y programación del EV3 con RobotC en el laboratorio. Obtención de resultados y creación de gráficas.	10
Instalación de RobotC, Robot Virtual World y estudio de los distintos tipos de escenarios y modos diferentes de simulación en este entorno.	10
Estudio de escenarios en Challange Pack for EV3 y creación de escenarios en RVW Level Builder y EV3 de las distintas aplicaciones.	15
Programación en RobotC de las distintas aplicaciones en RVW.	15

Actividades	Tiempo (h)
Obtención de resultados, creación de gráficas y grabación de la simulación.	8
Estudio y búsqueda entre varios posibles simuladores como segunda opción. Entre ellos LEGO MINDSTORMS Home Edition, Virtual Robotics Toolkit y CoppeliaSim.	10
Estudio y búsqueda de diseñadores 3D de LEGO. Entre ellos LEGO Digital Designer y LeoCAD.	8
Diseño y exportación del EV3 en LeoCAD	20
Estudio e investigación sobre el modelado, la jerarquía y las propiedades de los objetos, sensores, y articulaciones en CoppeliaSim.	50
Importación y modelado del EV3 en CoppeliaSim.	35
Estudio e investigación sobre la creación de escenarios, la programación interna, y las conexiones remotas con otros programadores externos en CoppeliaSim.	30
Desarrollo y verificación de la conexión entre CoppeliaSim y MATLAB a través de la API remota.	8
Creación de los distintos escenarios de las aplicaciones en CoppeliaSim.	20
Programación interna en LUA de aplicaciones en CoppeliaSim.	15
Programación externa con la API remota de MATLAB para la simulación de distintas aplicaciones en CoppeliaSim.	15
Obtención de resultados, creación de gráficas y grabación de las simulaciones.	8
Estudio y comparación de los resultados entre ambos simuladores.	8
Búsqueda bibliográfica.	3
Redacción de informes, de la carpeta, memoria y presupuesto.	40
Reuniones en videollamadas.	3
TOTAL	**342**

Tabla1. Estimación de horas del Ingeniero en Tecnologías Industriales.

También se han estimado además unas 35h empleadas por un ingeniero Catedrático de la UPV, incluyendo tutorías, revisión de documentos, etc.

Basándonos en las recomendaciones del presupuesto de la UPV para poner los precios, se ha creado el siguiente cuadro de precios de mano de obra.

Concepto	**Coste (€/h)**	**Rendimiento (h)**	**Importe (€)**
Ingeniero Junior.	31.1	342	10.636,2
Ingeniero Senior.	51.4	35	1.799
		TOTAL	**12.435,2€**

Tabla 2. Cuadro de precios de mano de obra.

2.2 Maquinaria y licencias software.

Ahora vamos a centrarnos en la maquinaria usada para el proyecto y en las licencias de software necesarias para la realización del mismo. La amortización de los equipos se calculará de la siguiente forma:

$$Coste = \frac{coste\ del\ concepto * tiempo\ de\ uso\ (meses)}{12 * tiempo\ de\ amortización\ (años)}$$

Se han tomado los periodos de amortización siguientes:

- 8 años para equipos y útiles.
- 5 años para medios informáticos.
- 1 año para licencias ya que es lo que dura las mismas.

El cuadro de precios de maquinaria queda:

Concepto	**Tiempo de uso (meses)**	**Periodo de amortización (años)**	**Coste del concepto (€)**	**Coste total (€)**
Ordenador portátil	5	5	499	41,58
Pack Robot Lego EV3	2	8	399.99	8,33
Wi-fi	4	5	25	1,67
RobotC y RVW	4	1	78.53	26,18
Matlab y Simulink	4	5	500	33,33
			TOTAL	**111,09€**

Tabla 3. Cuadro de precios de maquinaria y software.

2.3 Material fungible.

El material fungible utilizado en el proyecto ha sido el siguiente:

Concepto	**Unidades**	**Coste (€)**	**Coste total (€)**
Folios	49	0.01	0.49
Impresión	1	14.7	14.7
Encuadernación	2	7	14
Boligrafos	1	0.5	0.5
		TOTAL	**29.69€**

Tabla 4. Cuadro de precios material fungible.

3. Resumen del presupuesto.

A partir de los resultados anteriores se ha calculado el presupuesto de ejecución material (PEM), tomándose un 13% de gastos generales y un 6% de beneficio industrial. Seguido, se obtiene el presupuesto de ejecución de contrata (PEC) y, por último, se calcula el presupuesto base de licitación (PBL) tras aplicarle el 21% de IVA.

Concepto	Coste total
Mano de obra	12.435,2€
Material Inventariable	111,09€
Material fungible	29.69€
PEM	**12.575,98€**

Gastos generales 13%	1.634,88€
Beneficio industrial 6%	754.55€
PEC	**14.965,41€**

IVA 21%	3.142,74€
PBL	**18.108,15€**

El coste del proyecto finalmente asciende a un total de 18.108,15€.