MÁSTER EN AUTOMÁTICA E INFORMÁTICA INDUSTRIAL

UPV

MODELADO DEL ROBOT RB-1 EN V-REP

ALUMNO: JESÚS MACIÁ ROMERO
TUTOR: RANKO ZOTOVIC STANISIC

INDICE

1. Introducción

Durante los últimos veinte años ha existido una gran expansión en el campo de la robótica de servicio, orientada principalmente a la realización de tareas de tipo físico en sectores tan diversos como la agricultura, la construcción, la minería, la sanidad o el sector aeroespacial. Este campo de la robótica, tiene su mayor grado de desarrollo en centros de investigación, tanto públicos como privados, cuyo principal objetivo es conseguir la implantación de la robótica de servicio en el día a día de las personas.

Existen diferentes características que diferencian a los robots de servicio de los robots industriales. La principal sería que, a diferencia de la mayoría de los robots destinados a la producción industrial, los robots de servicio suelen tener una mayor gama de operaciones a realizar, lo que se traduce en: un sistema sensorial más potente, mayor flujo de datos a analizar, algoritmos más complejos, etc. Por otro lado, gran parte de la robótica de servicio está orientada al transporte de objetos, lo que requiere de un sistema de locomoción, que puede implementarse o bien mediante ruedas o mediante extremidades, dependiendo de la aplicación y del terreno en el que se vaya a desarrollar. Por último, cabría mencionar que gran parte de los robots de servicios están destinados a actividades que implican la interacción con humanos, por lo que los sistemas de control y de seguridad a implantar son mucho más exigentes que los de los robots industriales que trabajan en entornos acotados.

Dentro de la robótica de servicios se encuentran los robots personales, entre los que se pueden diferenciar los robots de vigilancia, los robots domésticos o los robots sanitarios. Esta clase de robots tiene en común que operan en entornos cerrados como pueden ser casas, oficinas u hospitales. Por ello, es necesario dotar a estos robots de una autonomía que les permita desplazarse, interaccionar con humanos y cumplir sus objetivos correctamente. Sin embargo, aunque estos robots operen en entornos parecidos, no será igual la capacidad de carga ni las dimensiones de un robot destinado a la clasificación de objetos en un almacén que las de uno cuya función sea el transporte de material sanitario; o el sistema sensorial que requiere un robot aspiradora que el de un robot que se ocupe de dar asistencia a una persona con movilidad reducida.

Llegados a este punto, se puede comprender la indispensabilidad de un software de

simulación, común para cualquier clase de robot en el que, una vez establecido su modelo, podamos averiguar cómo será su comportamiento ante diferentes situaciones; pudiendo desarrollar algoritmos, sistemas de control y de seguridad, dinámicas de las articulaciones y de los sistemas de locomoción, con la certeza de que una vez se implementen en el robot real, estos funcionarán correctamente.

En este proyecto se estudiará el robot personal *RB-1* de la compañía *ROBOTNIK,* de forma que se pueda establecer un modelo en el software de simulación *V-Rep*, que permita el análisis de la respuesta del robot ante el entorno a través de diferentes algoritmos.

2. Objetivos

Este proyecto tiene como objetivo principal la construcción de un modelo fidedigno del robot RB-1 en el software V-Rep. Para ello será fundamental el estudio de los diferentes componentes, tanto mecánicos como electrónicos y computacionales con los que cuente el robot.

Ilustración 1: Robot RB-1 de la compañía Robotnik

Este estudio incluye la realización de un modelado del sistema locomotriz del robot, el análisis de los sensores comerciales y su diseño dentro de la simulación, así como de su sistema de visión. Por último, también será necesario modelar las articulaciones (tanto del torso como del brazo robótico) para poder incluir su comportamiento correctamente en la simulación.

Una vez conformado correctamente el modelo en V-Rep, se implementarán diferentes algoritmos para comprobar su validez.

Para relacionar la simulación con el robot real, se utilizará el framework ROS implementado en Unix, de forma que el modelo del robot quede definido en formato urdf.

Posteriormente se realizará un modelo cinemático y dinámico de la estructura del robot, según su sistema de locomoción, que tendrá como objetivo determinar en qué situaciones aparecerían los mayores costes energéticos del movimiento del robot, o cómo corregir problemas derivados del deslizamiento o la inestabilidad.

En resumen, los objetivos de este proyecto de forma secuencial serán:

- Establecimiento del entorno software en una máquina virtual.
- Análisis del robot RB-1 y de sus componentes.
- Modelado del RB-1 en V-Rep
- Diseño de diferentes algoritmos
- Modelado dinámico y cinemático del robot

3. Entorno Software

3.1 Descripción del framework ROS y de sus características principales

Como producto de la investigación académica e industrial orientada a la solución de problemas relativos a la flexibilidad y portabilidad, generados en la implementación de sistemas robóticos complejos (varios robots trabajando a la vez, alto grado de

sensorización, gran volumen de datos), se desarrollaron los MRS (Multi-Robot Systems), que consisten en plataformas en las se integran todos los elementos presentes en un sistema robótico para la realización de una tarea concreta. ROS (Robot Operative System) fue uno de los primeros MRS, cuya razón de ser fue la de buscar la mayor flexibilidad posible y tratar de convertirse en un sistema universal para el diseño e implementación de aplicaciones robóticas.

La plataforma ofrece una serie de bibliotecas, drivers y herramientas de forma que se facilite y simplifique la comunicación entre los diferentes dispositivos y sistemas distribuidos, apostando por la modularidad, aportando robustez y en definitiva haciendo el desarrollo de aplicaciones robóticas más sencillo e intuitivo.

Actualmente, ROS cuenta con una gran aceptación tanto en el mundo empresarial como en la investigación académica, habiéndose convertido en un estándar en el desarrollo de sistemas robóticos.

Pero ¿qué elementos principales ofrece esta plataforma y cómo interactúan en pleno funcionamiento?

- Nodos: Los nodos son un concepto base sobre el que se sustenta el funcionamiento de ROS, cada nodo puede interpretarse como un proceso diferente, el cual tiene un comportamiento principal, por ejemplo, un nodo será el encargado de obtener la distancia mediante un sensor láser, otro se encargará de controlar los motores de las ruedas, otro realizará la planificación de la ruta del robot, etc.

- Topic: Un topic es una etiqueta con la que poder identificar el contenido de los mensajes y el canal que utilizan los nodos para comunicarse unos con otros. Cada nodo puede estar suscrito o bien publicar en un topic determinado, por lo que los nodos interactúan a través de los topics. Para que un topic exista, debe existir al menos un nodo publicando en ese topic.

- Mensajes: Información emitida por un nodo y recibida por uno o varios nodos, dependiendo del topic en el que se publique el mensaje. Los mensajes pueden incluir estructuras arbitrariamente anidadas o matrices, al igual que en C.

- Servicio: Un único nodo es el prestador del servicio, mientras que el resto de nodos

pueden solicitarlo mediante una arquitectura cliente-servidor, en el que el nodo solicitante emite un mensaje de solicitud y espera una respuesta. La cantidad y tipo de parámetros que requiere el servicio y los que envía como respuesta están indicados en el tipo de servicio.

- Paquete: Contiene todo el código fuente de los nodos y diferentes recursos necesarios para el funcionamiento de la plataforma, como las librerías usadas, las cabeceras, el tipo de lenguaje utilizado o la forma de compilación entre otros.

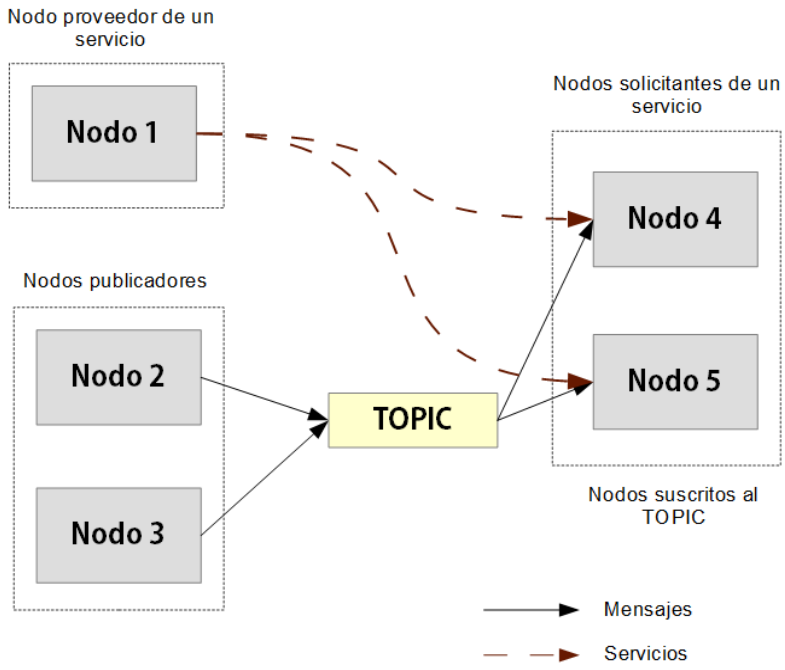


Ilustración 2: Diagrama de funcionamiento de la plataforma ROS

3.1.1. Instalación de ROS en una máquina virtual

En este apartado se describe paso a paso cómo se ha de realizar la instalación de la plataforma ROS en una máquina virtual (Oracle VM Virtual Box) con el sistema Ubuntu 16.04, de 64 bits. ROS también puede ser instalado en MAC OS X o en Windows (utilizando Microsoft Visual C++).

Es importante que antes de proceder con la instalación, se confirme que la sesión que se vaya a utilizar tenga al menos 4 Gb de RAM y 20 Gb de espacio en el disco.

Una vez se tenga la sesión de Ubuntu instalada y funcionando correctamente, se deberá elegir la distribución de ROS más conveniente para instalar en la máquina, ya que la mayoría de distribuciones de ROS (Indigo, Jade, Kinetic, Lunar, etc) solamente son soportadas por una versión de Ubuntu concreta, dependiendo del año de lanzamiento de la distribución. Para Ubuntu 16.04, la distribución más recomendable es Kinetic, así que a partir de este punto se deberán seguir exclusivamente los pasos para la instalación de esta distribución.

Lo primero que se deberá hacer es abrir un terminal e introducir la siguiente secuencia de comandos:

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

- *$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116*

- *$ sudo apt-get update*

- *$ sudo apt-get install ros-kinetic-desktop-full*

Este último comando provocará que el sistema comience la descarga e instalación de todos los paquetes incluidos en ROS Kinetic.

Si durante la instalación se obtiene algún error, será necesario introducir el siguiente comando para solucionarlo:

- *$ sudo apt-get install ros-kinetic-desktop-full –fix-missing*

Una vez termine la instalación, se deberá comprobar si el sistema está actualizado, para ello se habrán de introducir los comandos:

- *$ sudo rosdep init*

- *$ rosdep update*

Por último, será recomendable installar la funcionalidad “rosinstall” que facilita la descarga e instalación de muchas extensiones de ROS.

- *$ sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool build-essential*

Habiendo llegado a este punto, se debería de tener la plataforma ROS instalada correctamente. Una forma de comprobarlo será ejecutar el comando *$ roscore,* mediante el cual se accede al motor de ejecución.

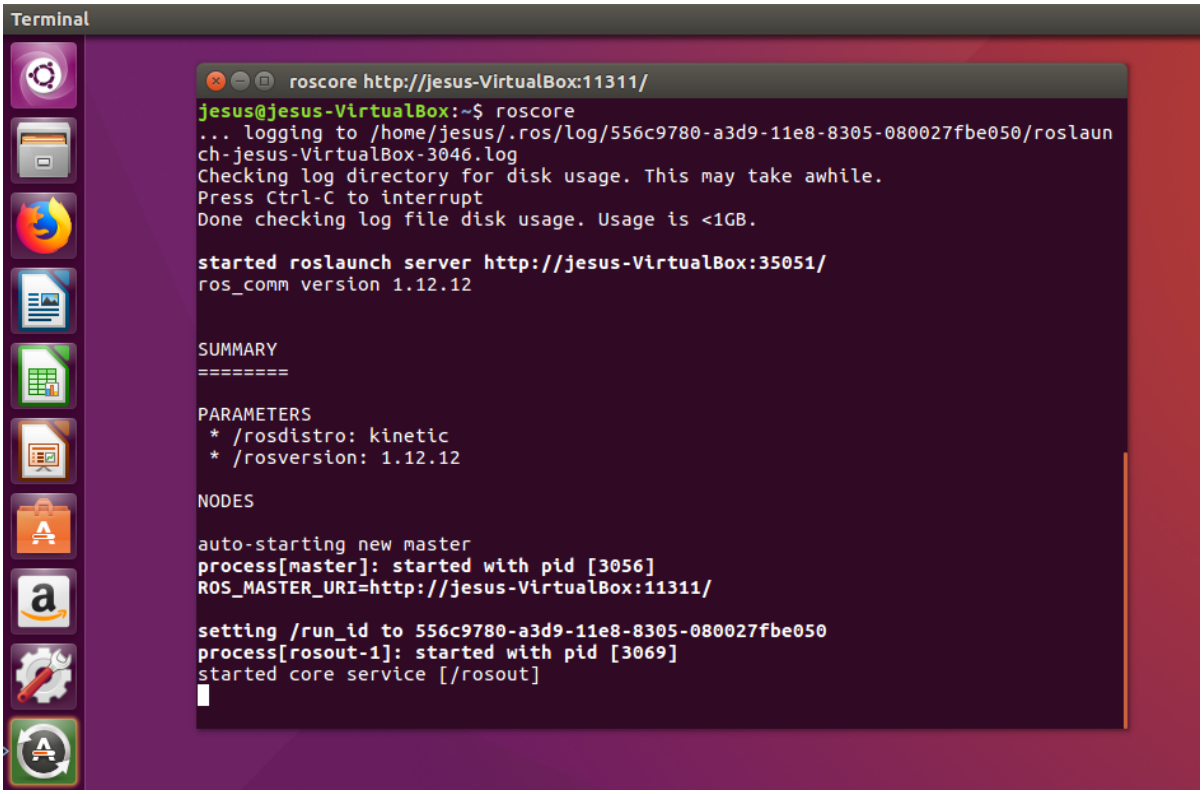


Ilustración 3: Resultado de la ejecución del comando "roscore"

En la imagen se pueden observar los nodos básicos que se inician con el motor de ejecución, entre los que se encuentran el nodo Master y el nodo “rosout”. El nodo Master proporciona un registro de nombres a través del cual, los nodos se pueden comunicar entre sí e intercambiar información y servicios. El nodo “rosout” es la salida de ROS, equivalente a stdout/stderr en UNIX.

Otra forma de verificar la instalación de ROS será ejecutando el comando:

$ printenv | grep ROS, que muestra en qué directorios están establecidas las diferentes variables de entorno como ROS_ROOT o ROS_PACKAGE_PATH.

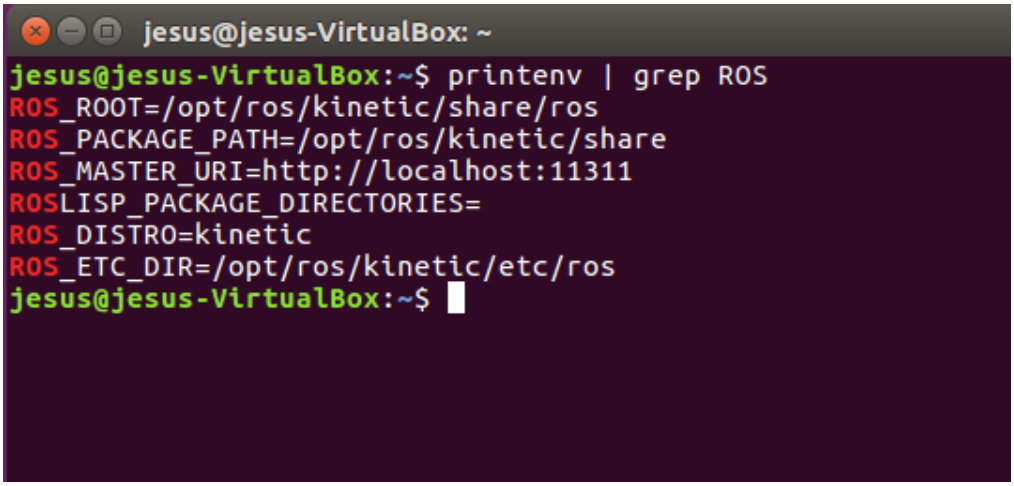


Ilustración 4: Variables de entorno de ROS y su ubicación

Una vez realizadas estas comprobaciones, se puede asegurar que ROS está correctamente instalado.

3.2 Descripción del software de simulación V-Rep

V-Rep es un software de simulación para aplicaciones robóticas, desarrollado por la compañía Coppelia Robotics, y que puede ejecutarse en Windows, Mac OS o Linux. Entre sus principales ventajas respecto a otros softwares de simulación, se encuentran: la gratuidad de su licencia educacional, la posibilidad de programar algoritmos en diferentes lenguajes como Lua (lenguaje principal), C++, Java, Python o MatLab entre otros, o la posibilidad de integrarse dentro de la plataforma ROS y la utilización del formato urdf.

Estas ventajas son compartidas por Gazebo, uno de los softwares de simulación más utilizados en robótica. Sin embargo, el principal motivante de este proyecto fue el hecho de que V-Rep cuenta con un motor de renderización 3D interno, más optimizado que el de Gazebo, que está basado en OGRE. Por ello, ante una gran cantidad de elementos, la simulación se ejecutará de forma más fluida en V-Rep que en Gazebo.

Otra ventaja significativa de V-Rep es la gran implicación de Coppelia en la resolución de problemas y su interacción con la comunidad generada en torno a este simulador.

4. Descripción y análisis del robot RB-1 de Robotnik

El RB-1 es un manipulador robótico móvil desarrollado por la empresa Robotnik, cuya primera versión fue lanzada al mercado en 2015. Se compone de una base móvil, comercializada por Robotnik de forma independiente con el nombre de RB-1 BASE, un torso en el que se encuentra el brazo robótico que permite la manipulación de objetos, y por último un pan-tilt en la parte superior del torso donde está situado el sistema de visión artificial.

Altura (min/max) (mm)	1029/1379
Diámetro base (mm)	500
Peso (Kg)	54
Velocidad (m/s)	1,5
Elevación máxima (mm)	350
GDL	6 Brazo, 1-2 Pinza, 2 Pan-tilt, 2 tracción, 1 elevador
Autonomía (h)	7
Potencia motores (W)	2x250
Rango de temperatura (ºC)	0 a 50
Brazo robótico MICO²	
Capacidad de carga (Kg)	2.1 (medio alcance), 1.5 (extendido)
Alcance (mm)	700
GDL	6
Brazo robótico JACO²	
Capacidad de carga (Kg)	2.6 (medio alcance), 1.5 (extendido)
Alcance (mm)	900
GDL	6
Procesador	Intel i7 (4ª generación)

RAM	8 Gb
Capacidad de disco duro	120 Gb
Control	Arquitectura abierta en ROS
Comunicación	WiFi 802.11n
Conectividad	2xUSB, 2xEthernet y 1xHDMI

Tabla 1: Características del RB-1

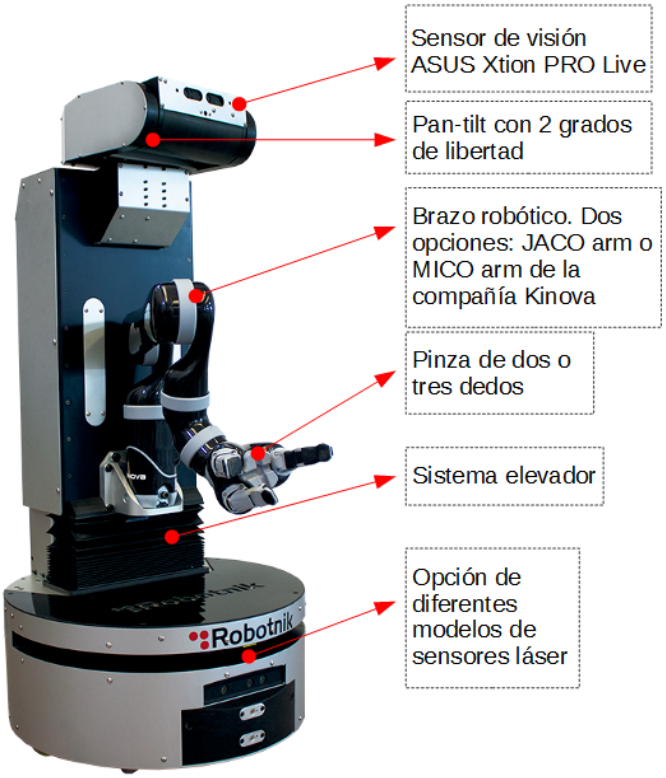


Ilustración 5: Elementos principales del RB-1

4.1. Plataforma móvil

Como se ha comentado anteriormente, Robotnik comercializa la plataforma móvil del RB-1 de forma independiente. Al igual que el RB-1, la plataforma RB-1 BASE está orientada a la investigación y el desarrollo de aplicaciones en interiores, teniendo como funcionalidad principal el transporte de objetos, con una capacidad de carga de 50 Kg.

La plataforma también permite que diferentes sensores RGB (ASUS Xtion, Kinect One, etc), sean montados sobre ella para la detección de obstáculos, sin embargo, esto no será necesario en el RB-1, ya que el sistema de visión artificial montado en el pan-tilt, permite un rango mucho más amplio que el que se podría conseguir con un sensor de visión en la base.

4.1.1. Sensores láser

Como se describe en la hoja de características del RB-1, la utilización de los sensores en el robot depende del modelo que elija el cliente, ya que cada uno de ellos ofrece diferentes posibilidades. La elección dependerá fundamentalmente de la aplicación o conjunto de aplicaciones para las que esté destinado el robot, siendo el precio también un factor importante a valorar.

Robotnik pone a disposición del cliente tres tipos de sensores láser diferentes. Los tres son de la marca Hokuyo y entre ellos varía su rango de detección, su ángulo de escaneo, su peso o su precio.

- UTM 30 LX: este sensor dispone del mayor rango de detección y velocidad de escaneo, pero su principal ventaja sobre el resto es que puede obtener datos de distancia en tiempo real; también es el más caro y el más pesado.

- UST 10 LX: este sensor ofrece unas características parecidas a las del UTM 30, sin embargo su rango de operación nominal es menor, ya que va desde los 60mm hasta los 10m. Es el más ligero, pero al contrario que los otros dos, no cuenta con interfaz USB, y su comunicación se realiza a través de Ethernet.

- URG 04 LX: es el más económico de los tres, por lo que tiene unas prestaciones menores que los dos anteriores. Su ángulo de barrido es de 240º y su resolución de 0.35º, en contraste con los 270º y 0.25º de resolución que ofrecían los dos modelos anteriores. Por otro lado, también cuenta con un rango de distancias menor.

En la siguiente tabla comparativa se pueden apreciar las características comentadas anteriormente.

	Hokuyo utm30 lx	Hokuyo ust 10 lx	Hokuyo urg 04 lx
Rango de detección	100mm a 30m	60mm a 10m	20mm a 5.6m
Velocidad de escaneo	25ms/scan	25ms/scan	100ms/scan
Voltaje	12V	De 10V a 30V	5V
Ángulo de escaneo	270º	270º	240º
Resolución angular	0.25º	0.25º	0.36º
Peso	370g	130g	160g
Precio	4770$	1680$	1080$

Tabla 2: Comparación entre sensores

4.1.2. Sistema de locomoción

El sistema de locomoción de la base del RB-1 se compone de un total de cinco ruedas. Dos de ellas son ruedas motrices y las tres restantes son ruedas pasivas con dos grados de libertad, las cuales permiten el desplazamiento en el eje X y en el eje Y.

El giro del robot se produce a través de la cinemática diferencial entre las dos ruedas motrices, es decir, los motores giran a velocidades diferentes produciendo una velocidad angular, cuyo valor dependerá de la diferencia del par aportado por los motores.

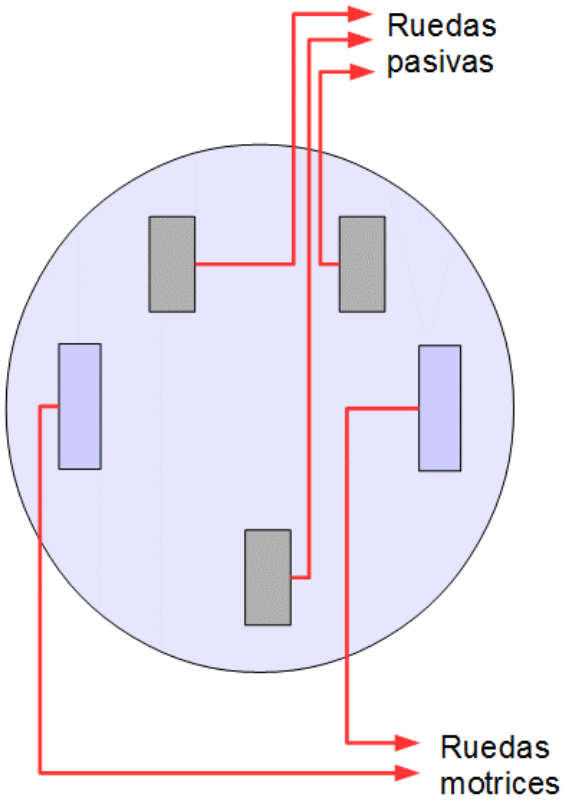


Ilustración 6: Esquema de la distribución de las ruedas en la base del RB-1

Para que las ruedas pasivas puedan adaptarse al movimiento de la base en giro, incluyen unos rodillos dispuestos a 90º del eje definido por la trayectoria que describe la rueda en un movimiento rectilíneo, permitiendo así el desplazamiento en el eje de giro principal. A

estás ruedas se les conoce como ruedas omnidireccionales o ruedas mecanum. No obstante, este tipo de ruedas suele tener los rodillos orientados a 45º de su eje de giro, ya que, si el vehículo cuenta con cuatro ruedas mecanum (dos delanteras y dos traseras), éste puede realizar desplazamientos completamente laterales únicamente combinando las velocidades de sus cuatro ruedas. Sin embargo este no es el caso que se está analizando, ya que las ruedas motrices no son ruedas omnidireccionales y por lo tanto, no

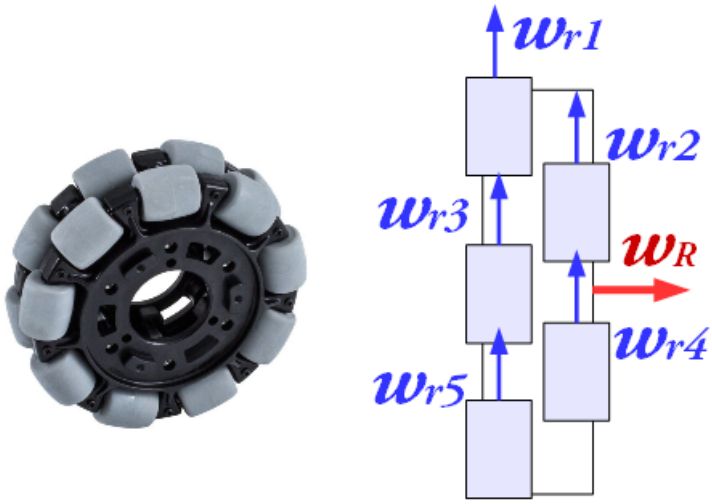


Ilustración 7: Rueda omnidireccional y distribución de las velocidades angulares de los elementos que intervienen en su movimiento

permiten un desplazamiento lateral puro.

Posteriormente, se abordarán con mayor profundidad todas las cuestiones relativas al movimiento del robot, en la parte referente al modelo dinámico y cinemático del mismo.

4.2. Torso, brazo robótico y pan tilt

Con el fin de permitir no solo el transporte, sino también operaciones que incluyen la manipulación de objetos como pueden ser el pick & place o la apertura de puertas, Robotnik desarrolló un torso mecánico móvil, que incorporaba un brazo robótico. El torso

puede desplazarse longitudinalmente en torno a una extensión vertical de la base, aportando así un grado de libertad más al robot. Este movimiento puede ser interpretado en el modelo como una articulación prismática.

4.2.1. Modelos Jaco y Mico de Kinova

El brazo robótico se monta sobre una pequeña plataforma que se encuentra en la parte inferior del torso, y que admite dos modelos diferentes de Kinova; el Jaco o el Mico, cuyas principales diferencias son su alcance y su capacidad de carga, características que han sido reflejadas anteriormente en la Taba 1.

Ambos modelos están principalmente orientados a prestar servicio y ayuda a personas con movilidad reducida, pudiendo montarse en sillas de ruedas y funcionando a través de las baterías que éstas llevan incorporadas.

Sus articulaciones están fabricadas con fibra de carbono y sus actuadores con aluminio, son resistentes al agua y existen, en ambos modelos, versiones de 6 o de 4 grados de libertad, teniendo estos últimos mayor capacidad de carga y menor peso. Tanto el Mico como el Jaco, tienen la posibilidad de incorporar una pinza de dos o de tres dedos,

Ilustración 8: Mico² (izquierda) y Jaco² (derecha) en sus versiones de 6GDL

también fabricada por Kinova.

4.2.2. Pan tilt y sistema de visión artificial

En la parte superior del torso del RB-1, se encuentra implementado el sistema de visión que permite al robot el reconocimiento de objetos y hace posible la manipulación de los mismos, pero también puede ser utilizado como sistema de navegación y localización, mediante el uso de puntos de referencia y algoritmos RGBD Slam. El sistema se compone principalmente de un sensor 3D ORBBEC Astra o bién de un sensor ASUS Xtion PRO

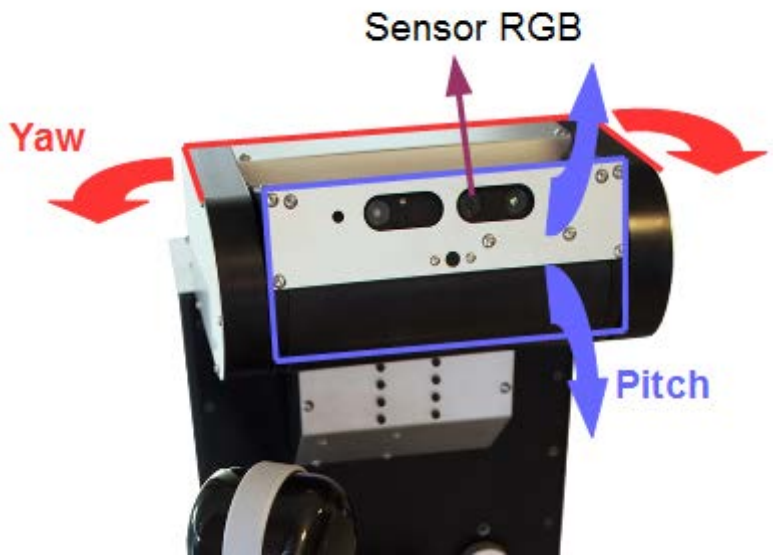


Ilustración 9: Descripción del sistema de visión del RB-1

según sean los requerimientos. La cámara o sensor 3D, se encuentra dispuesto en el interior de una carcasa metálica, la cual forma parte del pan tilt, y permite un movimiento de “pitch” o cabeceo. La segunda parte está unida al torso mediante una articulación que permite un movimiento de “yaw”, o guiñada en castellano.

De esta forma, el campo de visión del robot se amplía de forma notable gracias a los dos grados de libertad que ofrece el pan tilt, y será clave para marcar referencias en los

algoritmos de manipulación de objetos, como se verá posteriormente

5. Modelado del RB-1 en el software de simulación V-Rep

Llegados a este punto, es necesario plantear los elementos que existen de antemano para completar el proyecto.

Como se ha comentado anteriormente, Robotnik utiliza el software de simulación Gazebo, donde tiene introducidos la mayor parte de sus robots y que, vinculado a ROS, permite a ingenieros y técnicos anticipar el comportamiento de los robots antes de implementar los algoritmos en el mundo real.

En este software, Robotnik tiene modelado casi por completo el RB-1 BASE, incluyendo la sensorización láser y el sistema de locomoción. Así que, con el fin de reproducir con la mayor fidelidad posible el robot y aprovechar al máximo el trabajo realizado anteriormente por Robotnik, se van a utilizar en este proyecto los archivos relativos al RB-1 BASE que la compañía tiene colgados en el repositorio:

https://github.com/RobotnikAutomation/rb1_base_common

En el interior de la carpeta “rb1_base_description” se encuentran, por un parte, los archivos STL que se utilizarán para introducir las formas de los diferentes elementos mecánicos en el software de simulación. Por otra parte, se proporcionan los archivos con formato “urdf.xacro”, en los que se halla toda la información relativa a la cinemática, la dinámica y las relaciones de parentesco entre los elementos mecánicos de la base del robot.

Así mismo, en V-Rep se encuentran como modelos predefinidos los brazos robóticos Jaco² y Mico², así como las pinzas de dos y tres dedos de Kinova. Por consiguiente, el objetivo principal de este apartado será aprovechar los archivos que se encuentran en el repositorio de Robotnik para trasladar el modelo de la base a V-Rep, así como utilizar los modelos predefinidos en V-Rep para los dos tipos de brazos que el RB-1 admite. El torso junto con el pan tilt y el sistema de visión, se modelarán desde cero mediante la utilización del software SolidWorks para la creación de las piezas, y de las herramientas que V-Rep

ofrece en aspectos como las relaciones de parentesco entre piezas, sus relaciones dinámicas o todo lo referente a la visión artificial.

5.1. Archivo urdf para el RB-1 BASE

V-Rep cuenta con una función que permite importar archivos con formato urdf, sin embargo, en el repositorio de Robotnik se encuentran archivos urdf.xacro que describen partes concretas del robot. Este apartado tiene como objetivo construir, a través de la plataforma ROS, un archivo urdf en el que se encuentre descrito todo lo relativo al RB-1 BASE y que podamos trasladar a V-Rep.

5.1.1. Archivos urdf y urdf.xacro

Los archivos con extensión urdf (Unified Robot Description Format) son representaciones estándar en lenguaje XML, utilizadas en ROS para definir el modelo de un robot. En estos archivos están descritas todas las cuestiones relativas a la dinámica, la cinemática o la sensorización de los robots. Sin embargo, los archivos xacro se utilizan como un macro para hacer más legibles los archivos urdf, y poder dividir diferentes partes de código XML según correspondan a determinados elementos del robot, ya sean sensores, articulaciones, sistema locomotriz, etc.

En este caso particular, el modelo de la base del RB-1 está definido en archivos urdf.xacro, repartidos entre diferentes carpetas en el directorio: /rb1_base_common/ rb1_base_description/urdf. Cada uno de los archivos describe el comportamiento de un elemento diferente del robot; por ejemplo, en el “rubber_wheel.urdf.xacro” , se encuentran definidas todas las variables que afectan al comportamiento de la rueda motriz, como puede ser su radio, masa, inercia, velocidad máxima, relación de dependencia con otros elementos, etc. Todas estas especificaciones podrán ser modificadas posteriormente en V-Rep, pero es crucial que aparezcan predefinidas con los valores establecidos por Robotnik para la simulación en Gazebo, ya que de esta forma, el modelo se trasladará a

V-Rep fiel a su anterior versión.

```
8           <xacro:property name="M_PI" value="3.1415926535897931" />
9           <!-- Wheels -->
10          <xacro:property name="wheel_radius" value="0.076" />
11          <xacro:property name="wheel_height" value="0.025" />
12          <xacro:property name="wheel_mass" value="1.0" /> <!-- in kg-->
13
14
15          <xacro:macro name="cylinder_inertia" params="m r h">
16                  <inertia        ixx="${m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0"
17                                          iyy="${m*(3*r*r+h*h)/12}" iyz = "0"
18                                          izz="${m*r*r/2}" />
19          </xacro:macro>
20
```

Ilustración 10: Fragmento de código XML, perteneciente a la descripción de diferentes elementos propios de las ruedas motrices

Navegando por los directorios del repositorio puede apreciarse que existen diferentes versiones para la base. Esto se debe a que cada versión responde a unos requerimientos determinados, o lo que es lo mismo, ante unas especificaciones más altas se requerirá una sensorización más precisa que encarecerá el producto.

En el urdf generado, se incluirán los tres modelos de sensores analizados anteriormente para que se encuentren disponibles en V-Rep.

5.1.2. Creación de un archivo urdf a partir de los archivos xacro

El primer paso para generar el archivo urdf será descargar todo el contenido y los directorios que se hallan en el repositorio de Robotnik en Github. Posteriormente, se introducirán en una carpeta de nombre src, la cual será el directorio principal de trabajo en ubuntu.

El siguiente paso será modificar adecuadamente los archivos descargados para que los comandos de ROS tengan efecto.

Es necesario crear dos ficheros; el primero, de nombre “rb1_robot_macros.xacro” será en el que se reflejen todos los objetos “xacro” utilizados en los diferentes ficheros, como son por ejemplo *rubber_wheel*, *omni_wheel*, *sensor_hokuyo_laser_utm30lx,* etc. El otro fichero, de nombre “rb1_base.urdf.xacro” contendrá valores definidos para propiedades diversas relativas a la geometría de las ruedas, o propiedades globales, como el número pi. Ambos ficheros se incluyen en los anexos.

Una vez completada esta tarea, se deberá introducir la siguiente secuencia de comandos ROS en un terminal de ubuntu:

- *$ export ROS_PACKAGE_PATH=~/src:${ROS_PACKAGE_PATH}*

Con este comando se le indicará a ROS qué ruta se quiere añadir en adición a los directorios que ya tenga predefinidos. Para comprobar si la ruta *~/src* se ha añadido correctamente deberemos ejecutar el siguiente comando:

- *$ echo $ROS_PACKAGE_PATH*

Que da como resultado la siguiente línea

- */home/jesus/src:/opt/ros/kinetic/share*

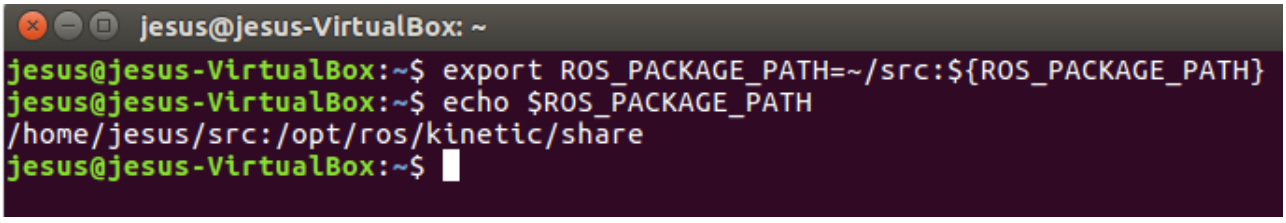


Ilustración 11: Cómo introducir un nuevo directorio de trabajo en ROS

De esta manera, ROS indica que existen dos directorios de trabajo disponibles:

/home/jesus/src (en el que se trabajará), y /opt/ros/kinetic/share, que ya estaba predefinido.

Habiendo llegado a este punto, ya se podrá ejecutar el comando que transforme todos los

archivos xacro en un único archivo urdf:

- *$ rosrun xacro xacro --inorder rb1_base.urdf.xacro > rb1_base.urdf*

Otro comando que realiza exactamente la misma función es:

- *$ rosrun xacro xacro --inorder -o rb1_base.urdf rb1_base.urdf.xacro*

Si se sigue de forma adecuada el procedimiento, ROS generará un fichero “rb1_base.urdf” donde se encuentre agrupado todo el código XML necesario para definir todas las características del robot, y que podrá ser importado a V-Rep.

Para comprobar que el fichero se ha generado correctamente, puede introducirse el siguiente comando:

- *$ check_urdf rb1_base.urdf*

Que resultará en un diagrama donde aparezcan los diferentes elementos y sus relaciones de parentesco.

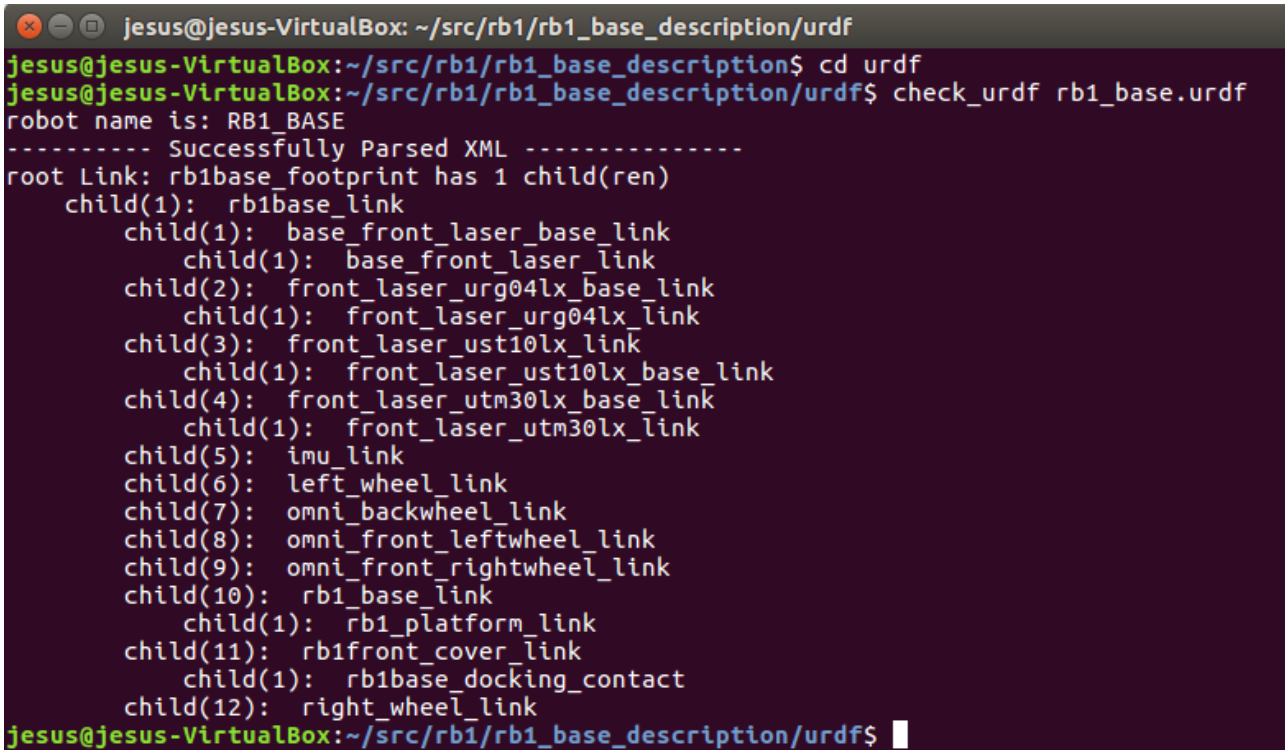


Ilustración 12: Relaciones de parentesco entre los diferentes elementos del fichero "rb1_base.urdf"

Otra forma de comprobar que el fichero se ha generado correctamente es analizar el código XML que contiene. Como se ha comentado anteriormente, en los ficheros urdf aparecen para cada elemento, los valores de sus magnitudes geométricas, inerciales, etc.

En la siguiente ilustración se muestra de qué forma debería aparecer definida una articulación (joint) y un eslabón (link) en el urdf.

```
<joint name="omni_front_rightwheel_joint" type="fixed">
  <parent link="rb1base_link"/>
  <child link="omni_front_rightwheel_link"/>
  <origin rpy="0 0 0" xyz="0.133 -0.148 0.0258"/>
  <axis rpy="0 0 0" xyz="0 1 0"/>
  <limit effort="100" velocity="100"/>
  <joint_properties damping="0.0" friction="0.0"/>
</joint>
<link name="omni_front_rightwheel_link">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://rb1_base_description/meshes/wheels/rb1v2_omniwheel.dae"/>
    </geometry>
  </visual>
  <!-- collision>
    <origin xyz="0 0 0" rpy="${M_PI/2} 0 0" />
      <geometry>
        <cylinder length="${omni_wheel_height}" radius="${omni_wheel_radius}" />
      </geometry>
    </collision -->
  <collision>
    <origin rpy="1.57079632679 0 0" xyz="0 0 0"/>
    <geometry>
      <sphere radius="0.051"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <origin xyz="0 0 0"/>
    <inertia ixx="0.000235075" ixy="0" ixz="0" iyy="0.000235075" iyz="0" izz="0.00039015"/>
    <!-- inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="0.001" / -->
  </inertial>
</link>
```

Ilustración 13: Fragmento del código contenido en el fichero rb1_base.urdf

5.1.3. Enlace entre V-Rep y ROS en la máquina virtual

Una vez generado el archivo urdf del RB-1 BASE, el siguiente objetivo será importarlo a V-Rep. Para que V-Rep esté vinculado con ROS, es necesario descargar e instalar un plugin ubicado en el siguiente repositorio:

https://github.com/CoppeliaRobotics/v_repExtRosInterface

Coppelia ofrece un tutorial completo sobre cómo trabajar con V-Rep estando vinculado a ROS Indigo, a través de un robot llamado “BubbleRob”. El tutorial puede encontrarse en la siguiente dirección:

http://www.coppeliarobotics.com/helpFiles/en/rosTutorialIndigo.htm

5.2. Modelado del RB-1 BASE en V-Rep

El próximo objetivo será modelar correctamente la plataforma o base del RB-1 en V-Rep. Para ello primero habrá que importar el archivo urdf que se ha creado en el anterior apartado, mediante el plugin “URDF import” que ofrece V-Rep en el apartado de plugins.

Una vez importado el archivo, en V-Rep aparecerán todos los elementos de la base con sus magnitudes especificadas y las relaciones de jerarquía correctas.

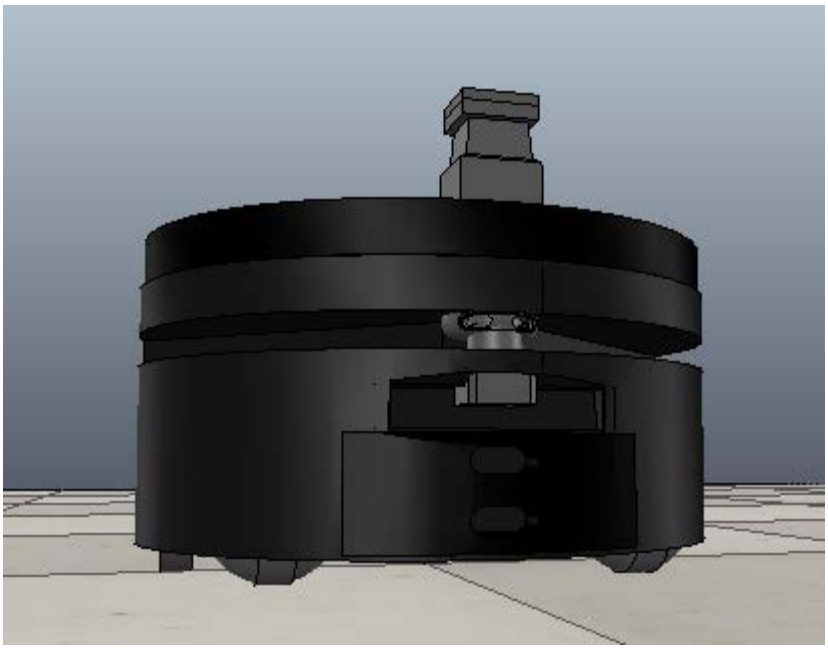


Ilustración 14: RB-1 BASE en el entorno gráfico de V-Rep

Al importar el modelo aparece un mensaje de error indicando que no se han podido importar correctamente algunas características de los sensores propias de Gazebo, por lo que será preciso modelar cada sensor por separado según sus diferentes propiedades. Por otra parte, en el apartado perteneciente al árbol de relaciones jerárquicas entre los diferentes elementos del robot y del entorno, se observa que no existen articulaciones vinculadas a las ruedas omnidireccionales, al contrario que en el caso de las ruedas motrices, que sí que están modeladas correctamente.

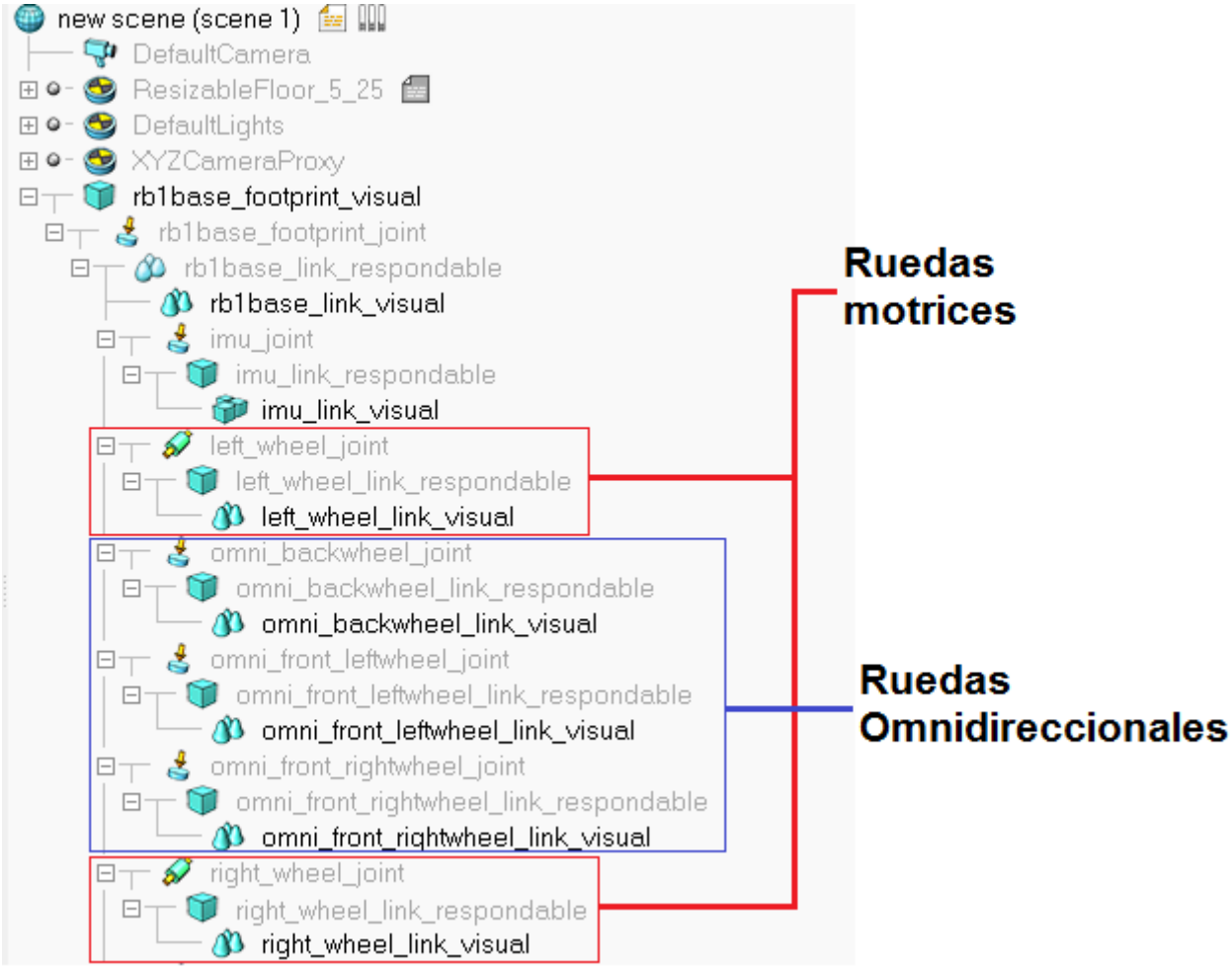


Ilustración 15: Relaciones de jerarquía de las ruedas del RB-1 en la interfaz de V-Rep

Por ello, será necesario adaptar también de forma adecuada el sistema locomotriz en V-Rep, de forma que las articulaciones permitan a las ruedas omnidireccionales un movimiento libre de dos grados de libertad descrito anteriormente.

5.2.1. Modelado de los sensores

Como se ha expuesto anteriormente, el RB-1 permite incorporar en la base diferentes modelos de sensores Hokuyo. Cada uno de los tres modelos tiene características diferentes como son su rango de detección, su velocidad y ángulo de escaneo, o su resolución angular. Obviamente, su precio también difiere entre modelos.

En V-Rep se puede encontrar uno de los modelos, el Hokuyo URG04 LX, en tres versiones diferentes de simulación; la estándar, la rápida, y la que se encuentra vinculada a ROS. Como toda la parte del proyecto relativa al modelado del robot en V-Rep se va a realizar de manera externa a ROS, debido principalmente a la complejidad del manejo de la plataforma y a la poca portabilidad que ello conlleva, se focalizará en las dos primeras versiones.

La primera de las versiones es la que pretende ser más fiel a la realidad, ya que simula el sensor como un único rayo láser que realiza un barrido de 240º en torno a su eje de giro, controlado por una articulación. En su script asociado se estipulan los valores de todos los parámetros que se han citado anteriormente, se recoge la distancia obtenida y se dibujan las líneas concéntricas, correspondientes al barrido del sensor, para aportar una referencia gráfica de su funcionamiento.

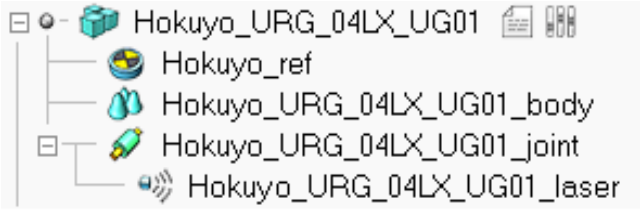


Ilustración 16: Disposición de los diferentes componentes presentes en el modelo del sensor Hokuyo URG04 LX en V-Rep

El empaquetamiento de los datos leídos por el sensor se realiza mediante la función: *sim.packFloatTable(),* y se envía a través de un tubo mediante la función: *sim.tubeWrite()*, ambas funciones Lua. Para leer los datos, se deberá emplear la función *sim.tubeRead().*

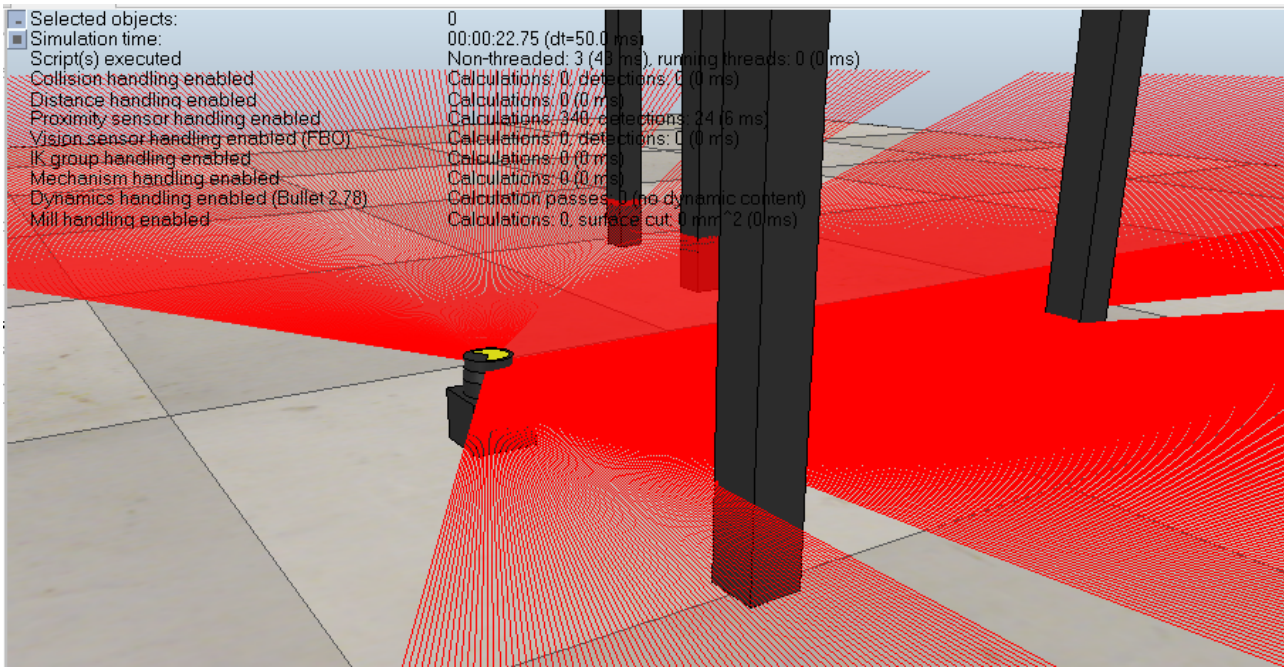


Ilustración 17: Imagen del entorno gráfico de V-Rep tras haber ejecutado una simulación que involucra al Hokuyo URG04 LX

La segunda versión se denomina fastHokuyo, y en lugar de realizar la simulación utilizando un sensor “ray type” de V-Rep (un único rayo láser), utiliza dos sensores de visión dispuestos de forma que cada uno de ellos comprenda un área de 120º, abarcando entre los dos un área de 240º.

Ilustración 18: Disposición de los componentes en la versión "fastHokuyo" de V-Rep

De esta manera, se prescinde de la velocidad de escaneo y la resolución angular, interpretando el sensor como un haz de rayos láser fijos, pudiendo definir únicamente el ángulo de escaneo y el rango mínimo y máximo de detección.

El desarrollo de dos versiones diferentes se debe a que, aunque en la versión estándar el modelo del funcionamiento del sensor es muy parecido a la realidad, éste conlleva un elevado coste computacional en simulación, que es aún mayor si se pretenden adaptar los modelos UST10 LX y UTM30 LX, debido a su alta precisión.

Haciendo uso de la herramienta “timing info”, que aporta información sobre la fluidez de la simulación, y cómo contribuyen los elementos presentes en ella a ralentizarla, puede observarse claramente este problema.

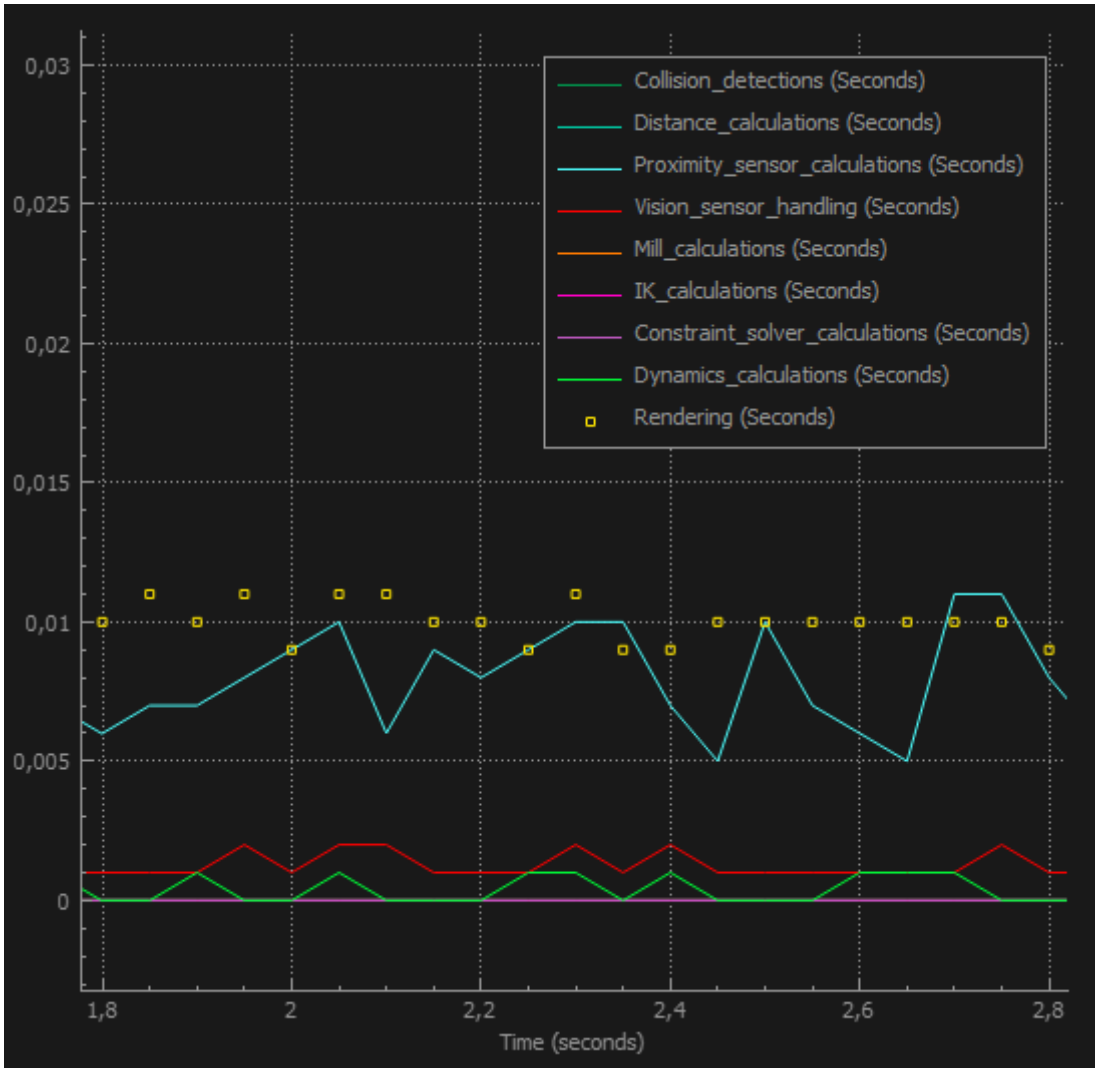


Ilustración 19: Retrasos generados por distintos elementos presentes en la simulación

En la ilustración 19 puede observarse que el retraso producido por la versión estándar del sensor es equivalente al retraso generado por la renderización, mientras que el retraso

debido a la versión rápida es prácticamente despreciable.

Sin embargo, en la siguiente ilustración se muestra el retraso provocado por una versión del sensor Hokuyo superior, con características que aportan mayor precisión (como pueden ser los modelos UST10 LX o UTM30 LX).

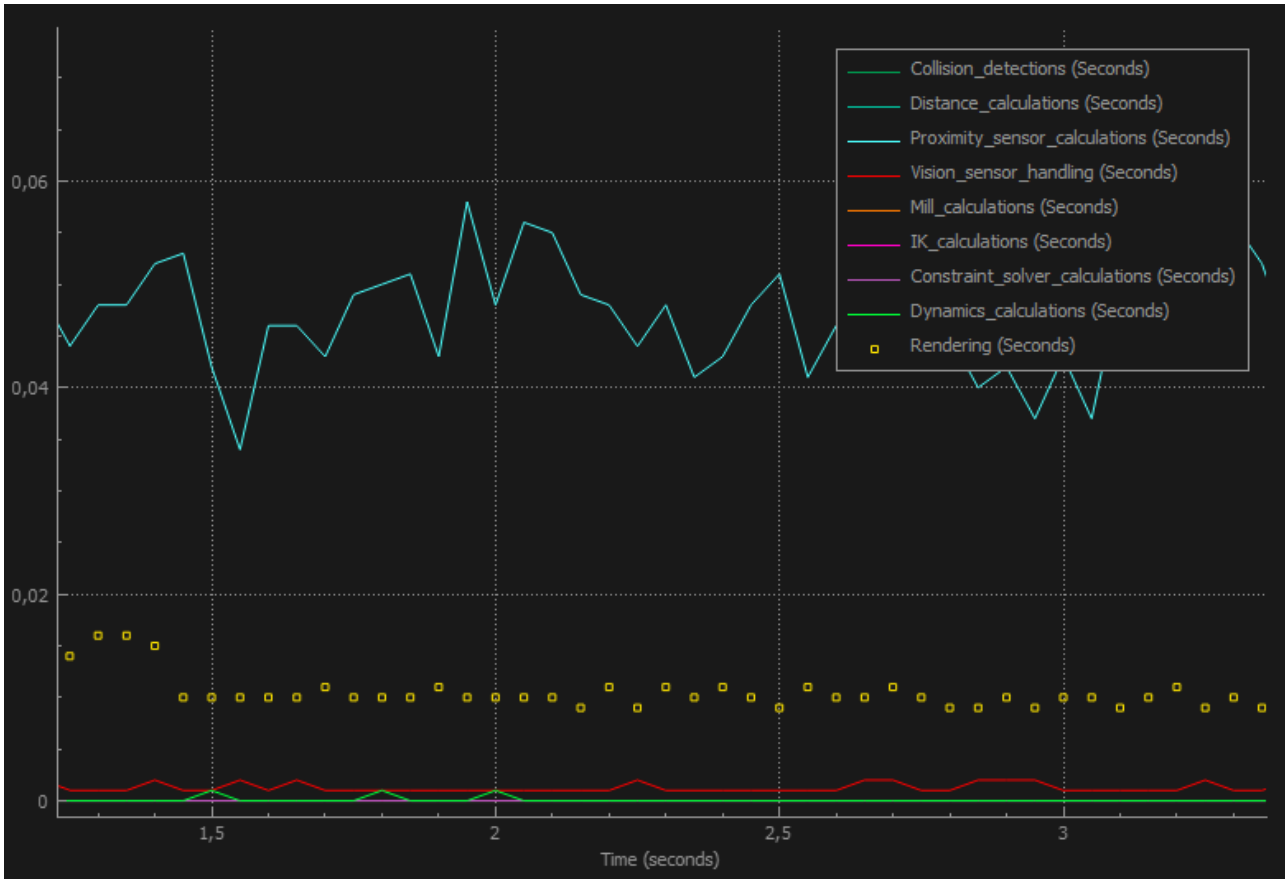


Ilustración 20: Ilustración 19: Retraso generado por una versión superior del sensor Hokuyo (azul)

Cabe señalar que estas gráficas han sido obtenidas en un entorno prácticamente desierto, con la finalidad de analizar únicamente el retraso debido a las especificaciones de resolución angular y velocidad de escaneo. Si la simulación se realiza con el sensor montado sobre la base, con ésta moviéndose a través del entorno mediante un algoritmo de navegación, el retraso generado por los cálculos del sensor aumenta de forma muy notable, llegando a valores de 400ms, haciendo su uso completamente inviable.

Dado que el principal motivante de este proyecto es que la simulación sea fluida (en

adición a que el robot esté correctamente modelado), debe hacerse especial énfasis en el control de este aspecto.

Por tanto, se ha prescindido de los parámetros de resolución angular y velocidad de escaneo, interpretando que tiene mayor importancia la fluidez de la simulación que el modelado de un error, que en la práctica, y para las aplicaciones para las que está destinado el robot, se puede interpretar como inexistente.

Siguiendo el modelo del fastHokuyo, se creará un sensor de características similares pero utilizando dos sensores de proximidad de tipo disco con ángulos de escaneo de 120º (135º en el caso de los UST10 y UTM30), en lugar de los sensores de visión, ya que es mucho más sencillo transmitir la información relativa a la distancia en sensores de proximidad que en sensores de visión.

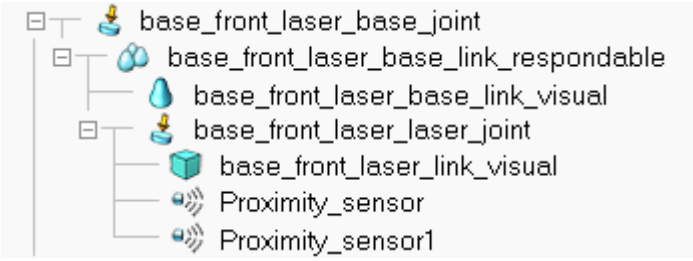


Ilustración 21: Sensor Hokuyo simulado en V-Rep a partir de dos sensores tipo disco

Los datos recogidos por el sensor, se manipularán en el script principal del robot. Estos scripts se encuentran en los anexos a este documento.

A fin de completar este apartado, se incluye en la ilustración 22 un último test donde se puede observar que la contribución del sensor de distancia a los retardos en la simulación es prácticamente nulo; teniendo mucha mayor incidencia el sensor RGB que forma parte del sistema de visión del robot.

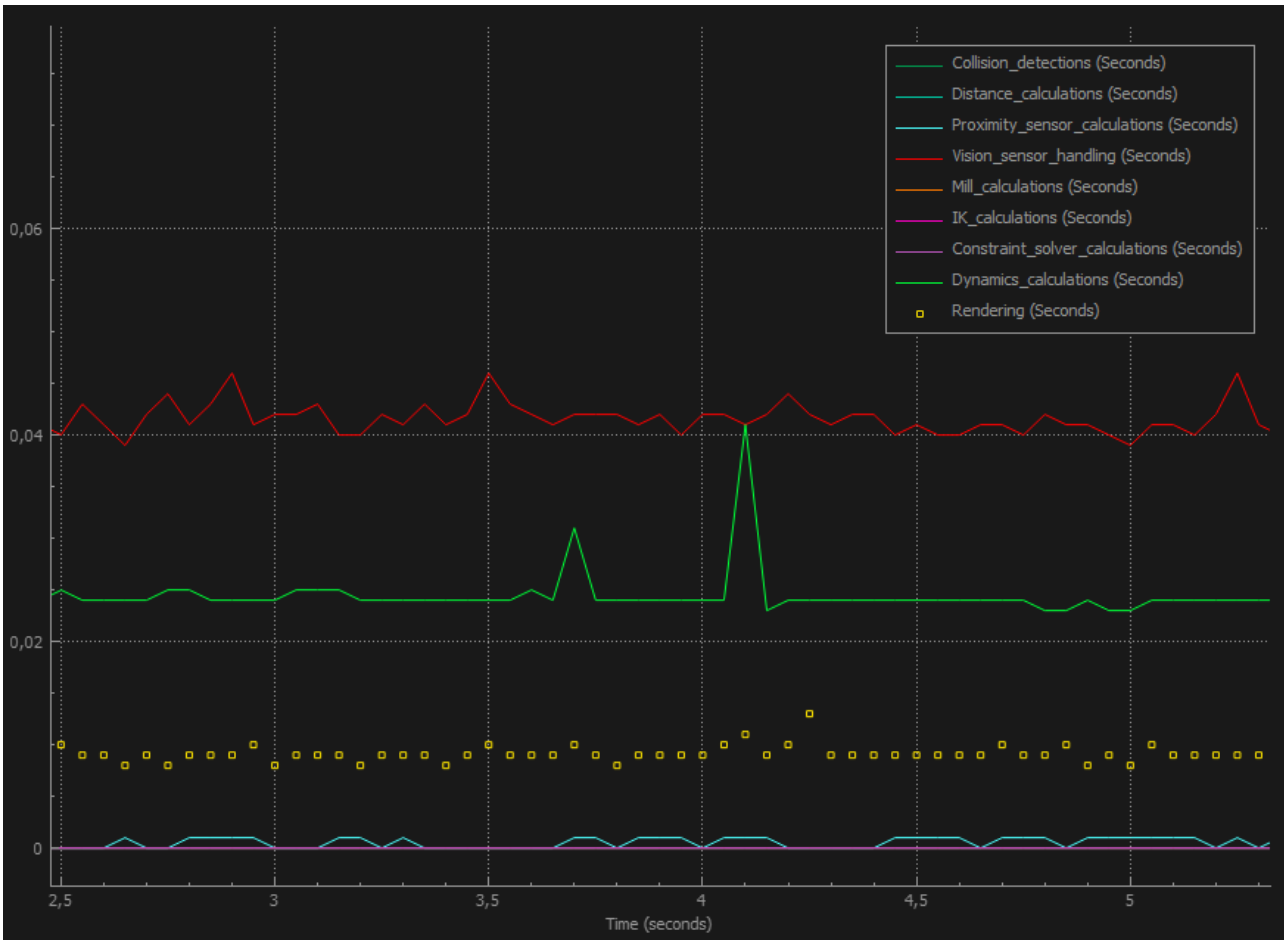


Ilustración 22: Contribución de cada uno de los elementos al retardo global de la simulación, estando el RB-1 completamente modelado

Por último, es necesario incluir en el modelo una IMU, ya que el RB-1 incorpora un sensor de este tipo para la navegación.

Bastará con incorporar un acelerómetro y un giróscopo, presentes en el apartado “componentes / sensores” del menú de modelos. Modificando sus scripts se puede leer información y utilizarla en diferentes algoritmos de navegación.

5.2.2. Modelado del sistema locomotriz

Como se ha comentado anteriormente, el sistema locomotriz del RB-1 se compone en primera instancia de dos ruedas motrices, las cuales quedan definidas en V-Rep una vez se importa el URDF del robot. En cambio, las ruedas pasivas omnidireccionales no aparecen correctamente modeladas y será necesario construirlas desde cero.

El primer problema que se presenta es el planteamiento que debe seguirse para modelar la rueda. Si se intenta realizar una interpretación lo más ceñida posible a la realidad, se deberá modelar la rueda, y posteriormente cada uno de los rodillos por separado, como si fueran pequeñas ruedas independientes. Este tipo de modelado implica una gran complejidad, ya que, con las herramientas que V-Rep ofrece, resulta prácticamente imposible salvar las restricciones dinámicas que conlleva encajar los rodillos en la rueda, y que ésta se comporte como una rueda omnidireccional pasiva.

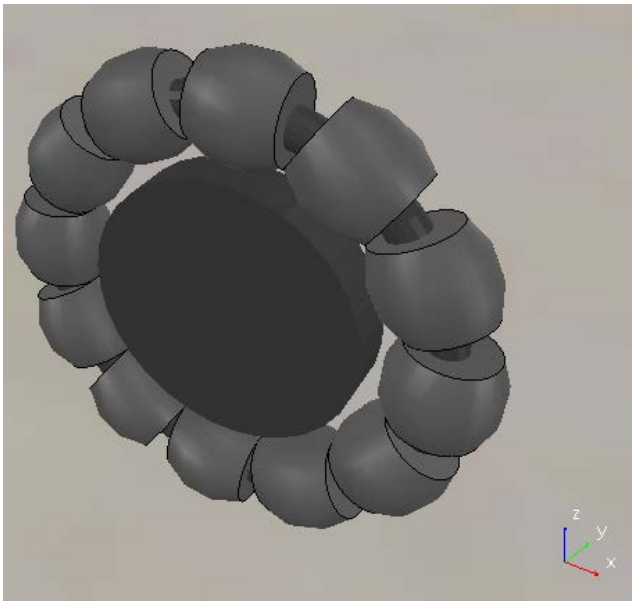


Ilustración 23: Rueda omnidireccional empleada para modelar las ruedas pasivas del RB-1

Si se analiza la forma de modelar ruedas pasivas en diferentes modelos de V-Rep, se comprueba que se utiliza otra técnica mucho más sencilla que la expuesta anteriormente. Las ruedas omnidireccionales (sean pasivas o no), se construyen como si fueran esferas,

pudiendo girar en la dirección que se desee sin que esto influya en las constricciones geométricas del robot.

La rueda se compondrá de diferentes elementos, que pueden ser separados en dinámicos y visuales. Entre los visuales se encuentra una articulación de revolución que define el movimiento de la referencia visual de la rueda (ilustración 23), la velocidad de esta articulación es controlada por un algoritmo implementado en el script principal del robot. A grandes rasgos, este algoritmo hace girar la rueda a la velocidad que dicta el modelo cinemático del robot; es decir, conociendo la velocidad de giro de las dos ruedas motrices, el modelo cinemático resuelve la velocidad de las ruedas omnidireccionales.

Por otra parte, los elementos dinámicos son los que realmente afectan al movimiento del RB-1. La principal razón de la incorporación de las ruedas omnidireccionales en el robot es la estabilidad que aportan, ya que sin ellas, el movimiento de la base sería oscilatorio debido a la inercia de su estructura.

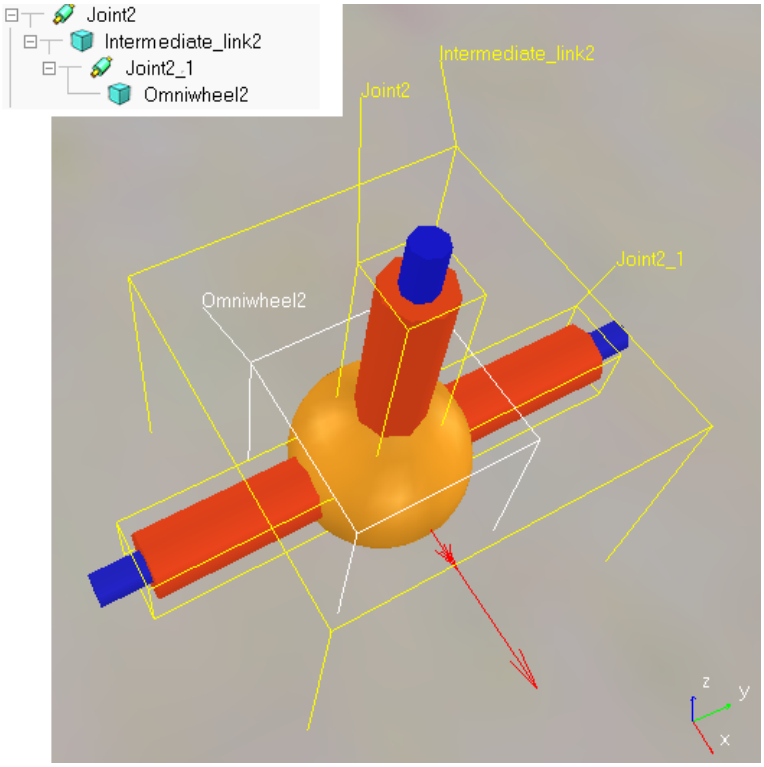


Ilustración 24: Estructura y relaciones de parentesco entre los diferentes elementos dinámicos que constituyen las ruedas omnidireccionales

El modelo se compone de dos articulaciones, la primera de ellas en jerarquía es vertical (perpendicular al plano X-Y), y será la que determine la dirección en la que girará la esfera. La segunda es la que define propiamente el giro de la esfera.

Ambas articulaciones estarán en modo “torque/force”, con el motor deshabilitado. Esta opción convertirá a las ruedas en pasivas y su velocidad de giro y dirección se determinará según la velocidad y dirección del robot, sin necesidad de tener que definirse en el script.

De esta manera, las ruedas omnidireccionales cumplirán la función de estabilizar el robot, sin afectar a su movimiento.

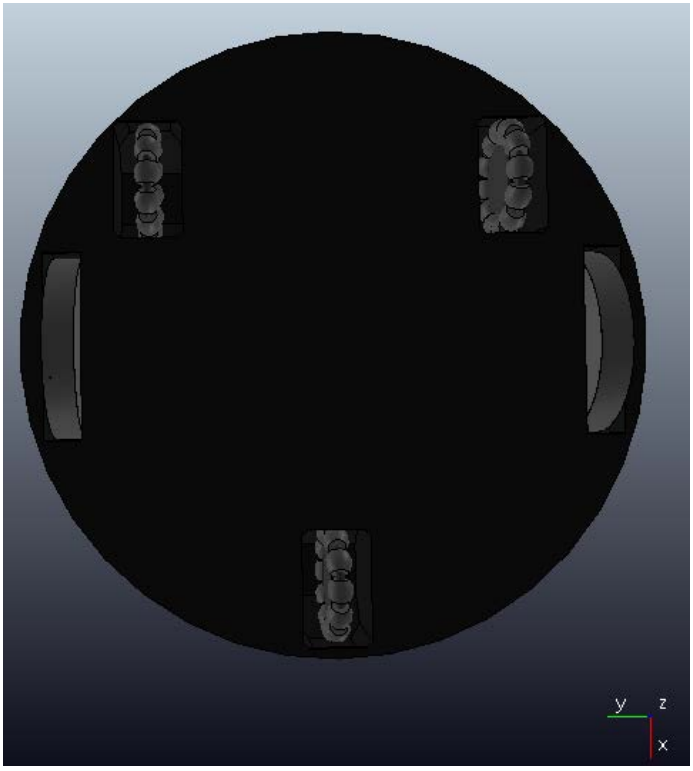


Ilustración 25: Chasis inferior de la base, en el que se puede observar la distribución de las ruedas motrices y pasivas en V-Rep

Evidentemente, la correcta implementación del sistema motriz debe ser verificada asignando valores de velocidad a las articulaciones de las ruedas motrices (cuyos motores sí que deben estar habilitados). Como se mencionó anteriormente, la máxima velocidad que puede asumir el RB-1 es 1.5 m/s, por lo que la simulación deberá verificarse a una velocidad mayor.

5.3. Modelado del torso y sistema de elevación, e inclusión del brazo robótico

Una vez concluido el modelado de la base del RB-1, el siguiente paso será construir el torso robótico que está montado sobe ella, así como su sistema de elevación. Dado que se pretende aprovechar el modelo de los brazos robóticos que V-Rep ofrece, en este apartado se prescindirá de los archivos que alberga el repositorio de Robotnik. Por lo tanto, será necesario definir las piezas geométricamente en SolidWorks, y después, importar sus modelos STL a V-Rep.

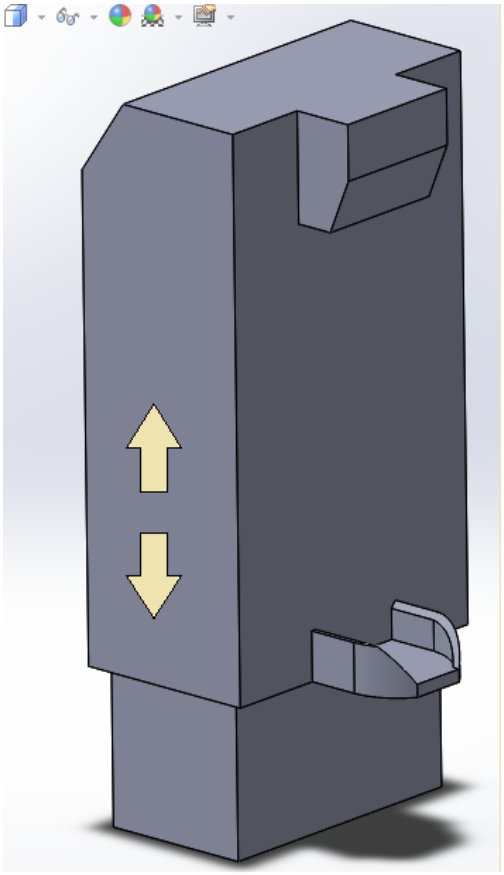


Ilustración 26: Piezas modeladas en SolidWorks pertenecientes al torso del robot

Una vez diseñadas, siguiendo las medidas que se hayan en los planos técnicos de Robotnik, e importadas a V-Rep, el siguiente paso será establecer la relación de parentesco entre la plataforma móvil y la pieza solidaria a la base. Para ello, será

necesario definir una articulación prismática, que establecerá el vínculo de unión entre estas dos piezas.

Primeramente se definirá un sensor de fuerza, que en V-Rep se utiliza para definir una relación solidaria entre dos piezas. Este sensor, o más bien unión rígida, se colocará entre la base y la pieza (1) sobre la que se desliza el torso del robot. En segundo lugar, se importarán los modelos geométricos de las piezas creadas en SolidWorks, y se relacionarán de la forma que se expone en la ilustración 27. Así, las piezas 2 y 5 deben quedar adheridas a la pieza 3, mientras que el brazo robótico (4), se vinculará a la pieza 3 mediante otro sensor de fuerza.

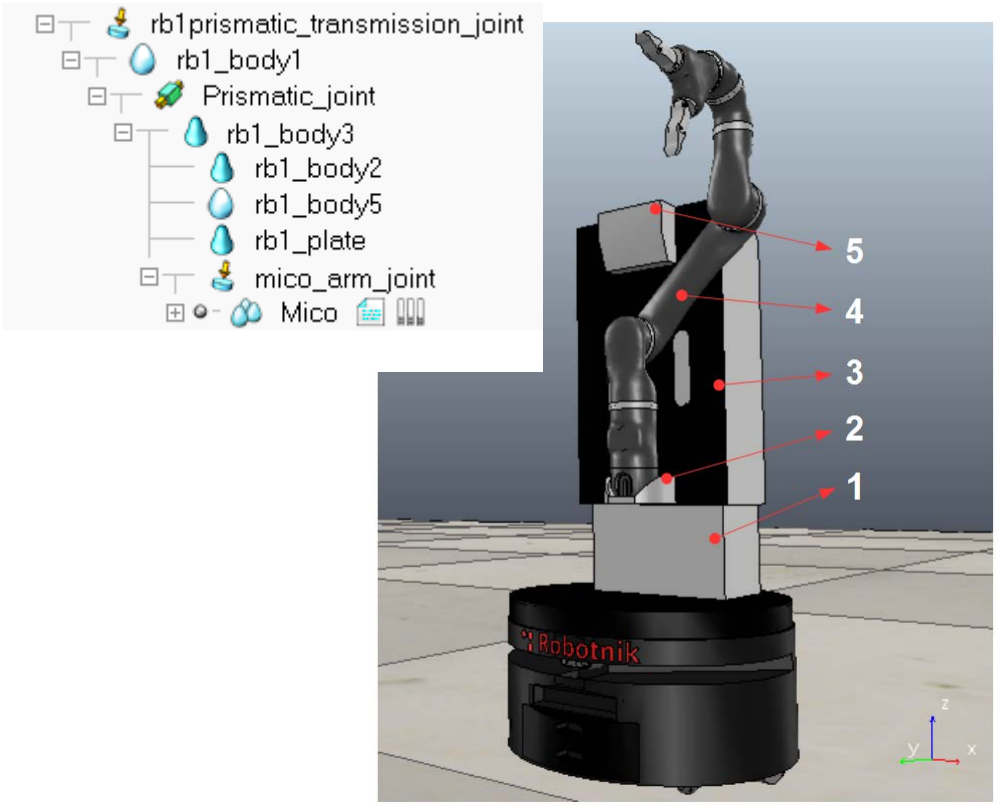


Ilustración 27: Diferentes piezas que intervienen en el torso del RB-1

Ahora quedará incluir el brazo robótico en el torso. Para ello, volverá a ser necesario utilizar un sensor de fuerza colocado sobre la pieza 2, que tendrá un grado de jerarquía mayor que el brazo robótico. Por defecto, los modelos Jaco² y Mico² incluidos en V-Rep, cuentan con un script de ejecución por defecto que define unos determinados puntos como objetivo. Estos objetivos están planteados según la cinemática directa, pero más adelante se analizará cómo es posible implementar la cinemática inversa en el script utilizando las funciones que V-Rep ofrece.

Es crucial incidir en las propiedades dinámicas con las que las piezas expuestas anteriormente deben contar. En primer lugar, la pieza 1 debe tener activada la opción “Body is dynamic” dentro del menú “Rigid Body Dynamic Properties”, ya que de lo contrario, el resto de elementos del torso se solaparán con esta pieza una vez que el robot se encuentre en movimiento. La pieza 2, también deberá tener esta opción habilitada para que la articulación prismática elevadora funcione cuando la simulación esté en ejecución. También es recomendable que ambas piezas tengan activada la opción “Body is respondable”, ya que de esta forma, las piezas pueden colisionar entre sí y esto es interesante a fin de detectar posibles problemas o, en definitiva, hacer que la simulación sea lo más realista posible.

A modo de conclusión, cabe mencionar que para conectar las pinzas de Kinova (tanto de dos como de tres dedos) incluidas en V-Rep, con los brazos robóticos, tan solo será necesario ajustar la posición de forma que el final y el principio de estas dos piezas sean coincidentes. Esto puede hacerse facilmente seleccionando primero la pinza, y posteriormente el último eslabón del brazo robótico, denominado “Mico_connection” o “Jaco_connection”; después, mediante la opción “Apply to selection” que se encuentra en el menú “Object/Item Translation/Position”. De esta forma, solo será necesario realizar unos pequeños ajustes para colocar la pinza en su posición correcta.

Una vez esté la pinza bien colocada, solo quedará arrastrar su modelo al grado de jerarquía inferior del “Jaco_connection” o “Mico_connection”.

5.4. Modelado del pan tilt y sistema de visión

Para terminar de introducir el RB-1 en V-Rep, solo quedará modelar el sistema de visión, para lo que será necesario volver a utilizar SolidWorks con el fin de diseñar las piezas pertenecientes al pan tilt del robot.

El pan tilt de la simulación se compondrá de tres piezas, como se puede observar en la ilustración 28; la pieza 1 se montará sobre la parte superior del torso robótico y será la encargada de realizar el "yaw" o movimiento de guiñada. La pieza 2 realizará el movimiento de "pitch" o cabeceo y por último, la pieza 3 simplemente servirá como recubrimiento para el sensor de visión que se haya en su interior.

La metodología empleada para desarrollar este sistema es muy similar a la del apartado anterior en lo que a relaciones de parentesco se refiere. Sin embargo, en este caso se deberán de utilizar articulaciones de revolución.

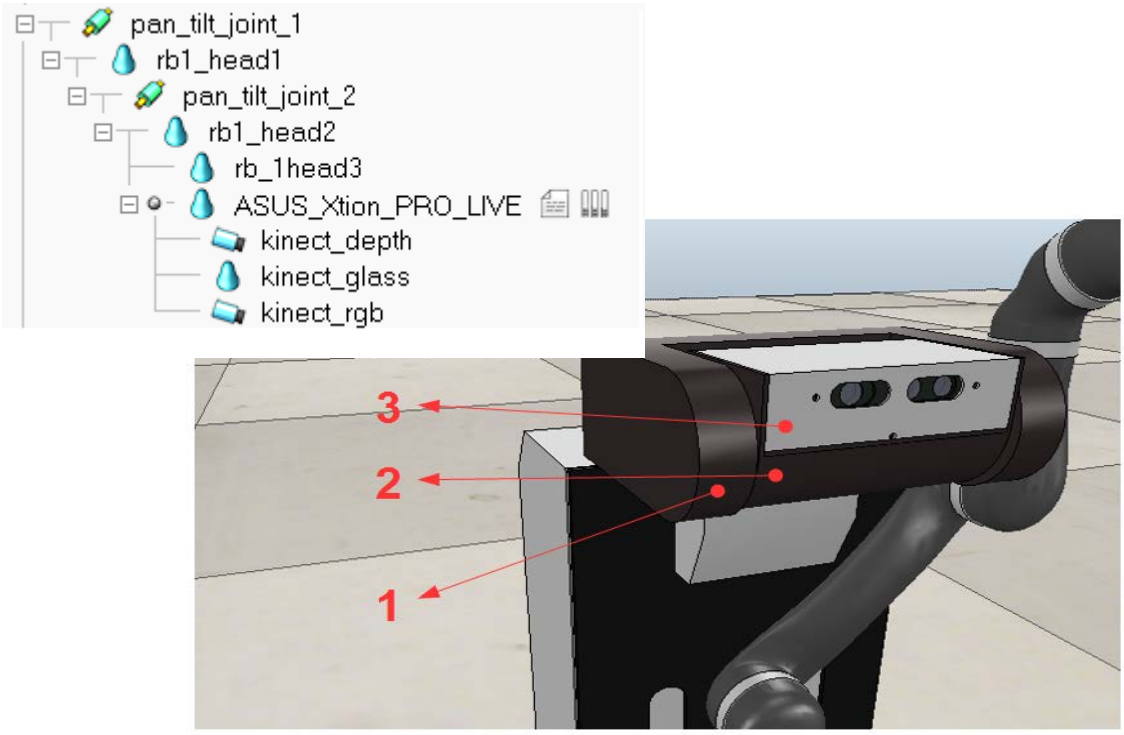


Ilustración 28: Modelado de los elementos del sistema de visión del RB-1 en V-Rep

Por último, solo quedará añadir el sensor de visión y adaptarlo a la pieza 3. Para ello, se importará un modelo STL disponible en la página *www.grabcad.com.* Posteriormente, se ha de incluir en el modelo el sensor rgb, el cual deberá ser añadido seleccionando “perspective type” entre las dos opciones disponibles.

También existe otra manera de implementar el sistema de visión, añadiendo a la simulación el modelo de kinect que V-Rep ya ofrece por defecto. De esta forma, solo habrá que modifical el “kinnect_body” y cambiarlo por el sensor ASUS que se ha importado. Este sensor predefinido, cuenta con un script el cual ofrece, una vez se ejecute la simulación, dos pantallas en las que puede observarse el funcionamiento del sensor rgb y de profundidad.

5.5. Verificación del modelo

Habiendo llegado a este punto, será necesario verificar que cada uno de los componentes incluidos en el modelo funciona correctamente. Estas comprobaciones se realizarán mediante un script global, asociado al eslabón del robot más alto en la jerarquía de elementos.

En este script simplemente se especificará la velocidad de las ruedas motrices, con el fin de comprobar si la relación dinámica entre los componentes del robot es correcta, una vez éste se encuentre en movimiento.

```
function sysCall_threadmain()

    objHandle_2=sim.getObjectAssociatedWithScript(sim.handle_self)

    leftJoint_2=sim.getObjectHandle("left_wheel_joint")
    rightJoint_2=sim.getObjectHandle("right_wheel_joint")
    v0=1
    wheelDiameter=0.152

while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do

            velocityLeft=v0
            velocityRight=v0
        sim.setJointTargetVelocity(leftJoint_2,velocityLeft*2/wheelDiameter)
        sim.setJointTargetVelocity(rightJoint_2,velocityRight*2/wheelDiameter)

    end
end
```

Ilustración 29: Script global asociado al RB-1 para verificar el correcto funcionamiento de la simulación

Mediante la función “sim.getObjectHandle”, se obtienen los manejadores de las articulaciones pertenecientes a las ruedas motrices, a las que se le asignará una velocidad angular determinada con la función “sim.setJointTargetVelocity”. Para que el RB-1 se mueva a 1 m/s, se deberá realizar la conversión a radianes/s, teniendo en cuenta el diámetro de las ruedas.

Si no existen problemas, el siguiente símbolo:  en cada una de las articulaciones, o en los elementos que tengan habilitada la opción “Body is dynamic”, o lo que es lo mismo, que intervengan en la dinámica de la simulación.

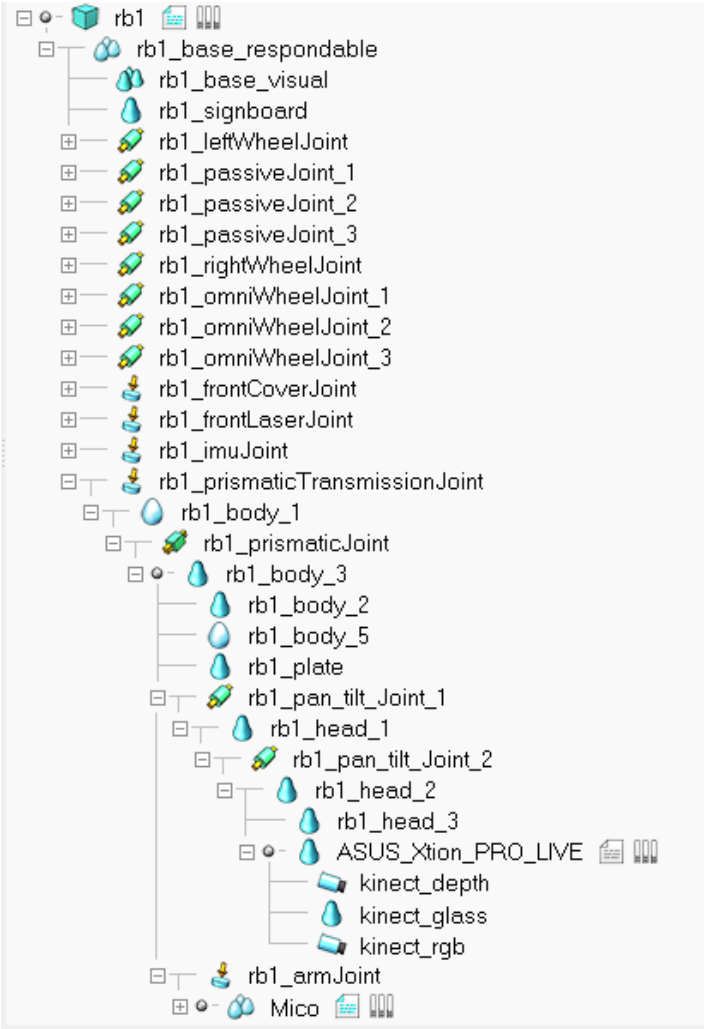


Ilustración 30: Árbol de jerarquía del RB-1 cuando la simulación se encuentra en ejecución

Teniendo en cuenta que la velocidad máxima del RB-1 es 1.5 m/s, se deberán realizar pruebas a mayores velocidades. También será preciso asignar velocidades diferentes a las articulaciones de las ruedas para observar cómo se comporta dinámicamente en curva. Otras comprobaciones interesantes serán la de la articulación elevadora en el torso, o las correspondientes al pan tilt, que seguirán un esquema similar.

Si el procedimiento se ha seguido correctamente, el árbol de jerarquía será el que aparece en la ilustración 30, y el modelo del RB-1 en el entorno visual, el que muestra la

ilustración 31.

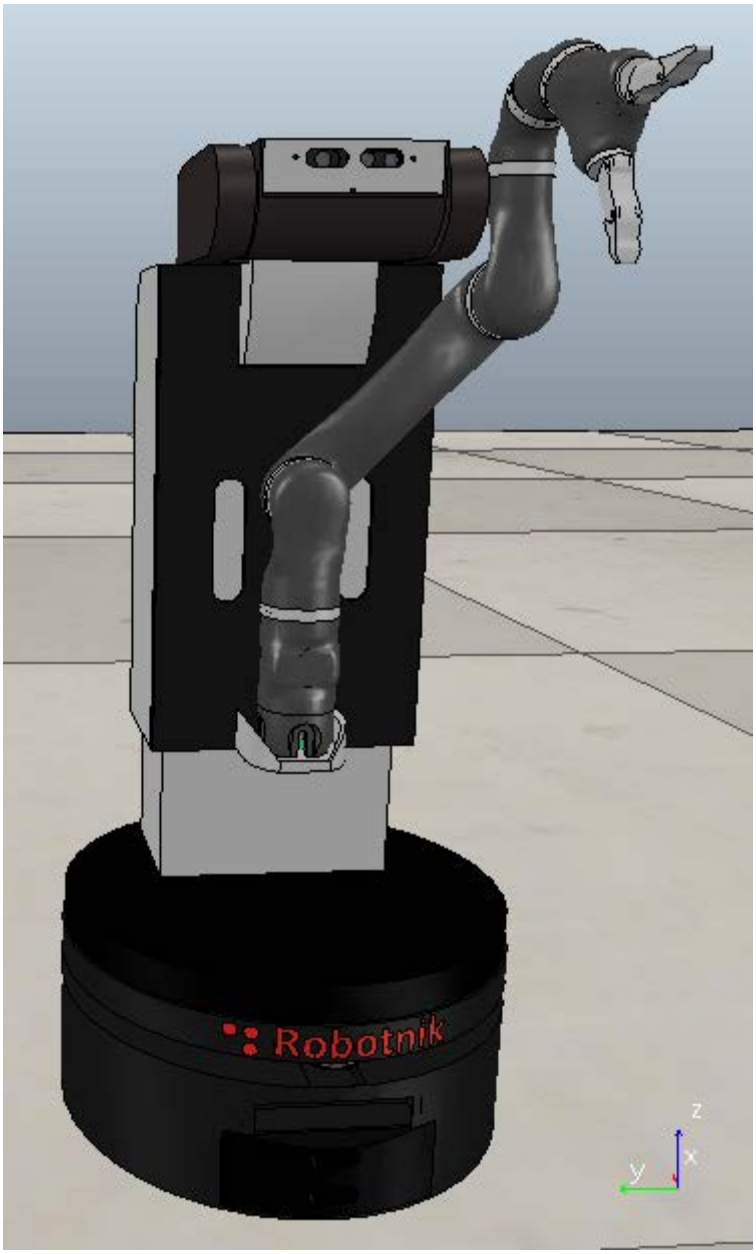


Ilustración 31: RB-1 en el entorno gráfico de V-Rep, una vez completado el modelado de todos sus componentes

6. Algoritmos desarrollados en V-Rep

6.1. Introducción y aspectos a destacar del lenguaje Lua y de su uso en V-Rep

En el apartado 5.5 se ha descrito brevemente cómo desarrollar un algoritmo para comprobar que es sistema es dinámicamente estable durante la ejecución. Sin embargo, es importante estudiar diferentes aspectos del lenguaje Lua, el cual viene predefinido en los scripts de V-Rep.

Como principales características de Lua se puede comentar que las variables no tienen tipo, sino únicamente datos lógicos, enteros, de coma flotante o cadenas. La única estructura de datos que posee es la tabla, que puede ser utilizada para reperesentar vectores, conjuntos, registros, etc. También es posible utilizar este lenguaje en programación orientada a objetos. Por otra parte, los programas de Lua no son interpretados directamente, sino que el código “bytecode” que se genera tras su complicación, es ejecutado por la máquina virtual de Lua.

V-Rep utiliza el lenguaje Lua para la programación de los scripts embebidos, no obstante, es posible programar los scripts de V-Rep en otros lenguajes diferentes como puede ser C++, utilizando la API remota. Para mayor información sobre el funcionamiento de este sistema, puede consultarse en el siguiente enlace:

www.coppeliarobotics.com/helpFiles/en/remoteApiModusOperandi.htm

Pero con el fin de simplificar el proyecto, se utilizará Lua para desarrollar los agoritmos expuestos en este apartado.

6.1.1. Tipos de scripts en V-Rep

Como primer aspecto importante a destacar, V-Rep hace una distincio entre el Main Script y los Child Scripts. El Main Scripts se crea cada vez que se genera un nuevo proyecto y controla todas las funcionalidades relativas a la simulación; es posible modificarlo, pero desde Coppelia recomiendan que la edición se realice a través de los Child Scripts, que

son scripts asociados a cualquier elemento presente en la simulación.

Existen dos tipos de Child Scripts, los “Threaded” y los “Non-Threaded”. Los del tipo Threaded se caracterizan porque un hilo de ejecución es creado cada vez que se ejecuta la simulación. El lanzamiento de los hilos es manejado por el Main Script en forma de cascada. Un Threaded Child Script puede ser ejecutado de nuevo durante la misma simulación, siempre que tenga la opción “execuje once” del menú “script properties” deshabilitada.

Los del tipo Non-Threaded funcionan como una colección de bloques de funciones, las cuales son llamadas por el Main Script dos veces por cada instante de muestreo (step time) definido en la simulación. También pueden ser llamados por otros child scripts, por ejemplo cuando se define una función “callback”. El uso de este tipo de scripts es siempre preferible al Threaded, ya que utilizan menos recursos, tiempo de procesamiento y responden inmediatamente cuando se detiene la simulación.

6.1.2. Funciones en V-Rep

Coppelia proporciona una lista con todas las funciones disponibles en V-Rep, tanto para la API regular como para una API externa:

http://www.coppeliarobotics.com/helpFiles/en/apiFunctionListCategory.htm

http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsLua.htm

Existen funciones para propósitos muy diversos, relacionadas con sensores de proximidad, de visión, cinemática inversa, transformaciones matriciales, etc. Sin embargo hay un grupo reducido cuyo uso es muy común y extrapolable a gran cantidad de aplicaciones.

La función más importante es *sim.getObjectHandle*, que almacena en una variable el manejador del elemento cuyo nombre se le pase como argumento a la función. Cada vez que se pretenda que un componente del robot, ya sea un sensor, una articulación o un cuerpo(s) sólido(s) intervengan en la simulación, será necesario aplicar esta función.

En lo referente a las articulaciones, las funciones más comunes son:

1. *sim.getJointPosition*, *sim.setJointPosition*
2. *sim.getJointTargetPosition*, *sim.setJointTargetPosition*
3. *sim.getJointTargetVelocity*, *sim.setJointTargetVelocity*

El primer grupo establece u obtiene la posición intrínseca de una articulación en el momento en el que se llame a esta función. El segundo grupo establece u obtiene la posición de una articulación siempre que ésta esté en modo *torque/force.* El último grupo establece u obtiene la velocidad a la que se debe mover (o se mueve) una articulación no esférica. La función tendrá efecto siempre que las funcionalidades dinámicas estén activadas, así como que esté en modo *torque/force* y el motor esté habilitado.

Para leer u obtener datos de los sensores presentes en la simulación, se utilizan las funciones *sim.readProximitySensor* y *sim.readVisionSensor* respectivamente.

6.2. Algoritmos de movilidad

El desarrollo de esta tipología de algoritmos, tenía como principal objetivo la verificación del modelo implementado. Se trata comprobación de la correcta interacción entre todos los elementos que integran el modelo, como son el sistema locomotriz, los sensores láser, el sistema de visión artificial, el pan-tilt y el brazo robótico.

El primero de los algoritmos implementados, tenía como objetivo la conducción del robot a través del entorno de V-Rep, de forma que éste fuera capaz de evitar diferentes obstáculos detectados a partir de los sensores láser. El esquema de funcionamiento se puede representar como un diagrama de estados en el que, al detectar un obstáculo, el robot rota hasta que las lecturas de los sensores obtienen valores superiores a una distancia mínima predefinida.

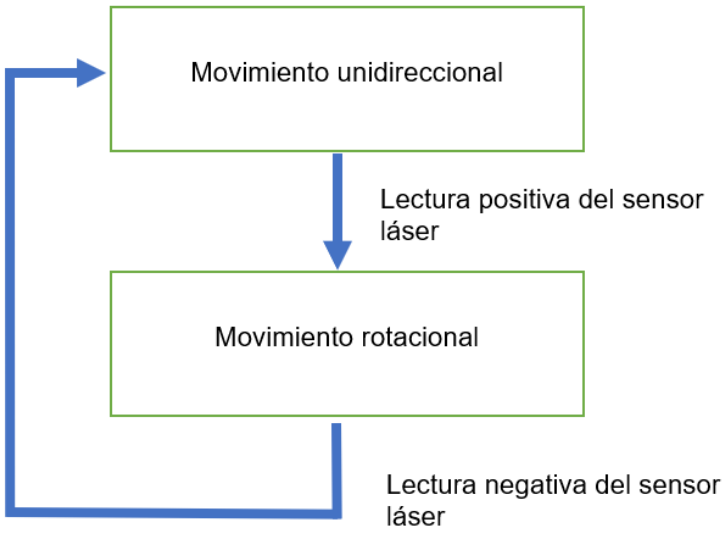


Ilustración 32. Diagrama de estados del algoritmo de movimiento

V-Rep ofrece diferentes tipos de obstáculos, tales como sillas o mesas, así como elementos constructivos como paredes o puertas.

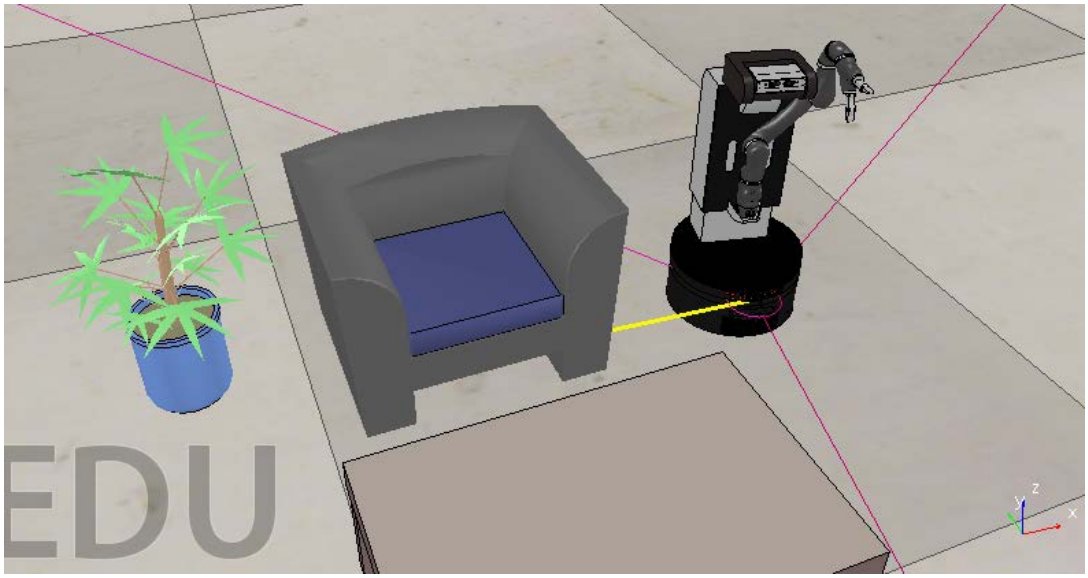


Ilustración 33. RB-1 actuando según el estado 2 del algoritmo de movimiento

El giro se produce al aplicar una velocidad menor a una de las ruedas motrices del robor. El radio de giro dependerá de la relación entre las velocidades angulares de dichas ruedas motrices.

6.3. Algoritmos de visión artificial

A través de la implementación de estos algoritmos en el modelo desarrollado, se pretendía comprobar la funcionalidad de la cámara instalada en la parte superior del robot. A través de las herramientas que ofrece V-Rep, se aplicó un filtro al sensor RGB que discriminaba el color rojo, y además, se utilizó una herramienta para determinar el centro de masas de la imagen discriminada, como se muestra en la siguiente captura.

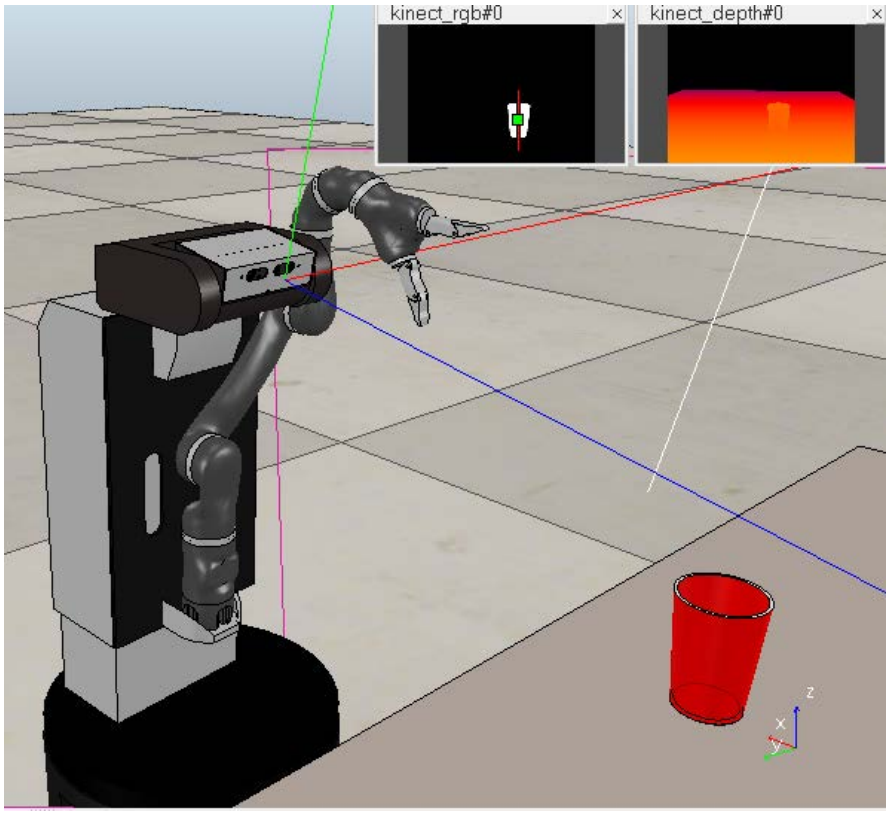


Ilustración 34. Detección de objetos a través del sensor de visión del RB-1

Las herramientas utilizadas fueron “Selective color on work image” para la discriminación

de objetos de color rojo, y “Binary work image and trigger” para la detección del centro de masas de los objetos discriminados.

6.4. Planificación de movimientos del brazo robótico

Por último, se verificó el correcto funcionamiento del brazo robótico integrado en el RB-1, a través de algoritmos de planificación de movimientos OMPL (Open Motion Planning Library). El objetivo de la implementación de estos algoritmos, era que el brazo robótico fuera capaz de agarrar y transportar un objeto, en este caso, un vaso.

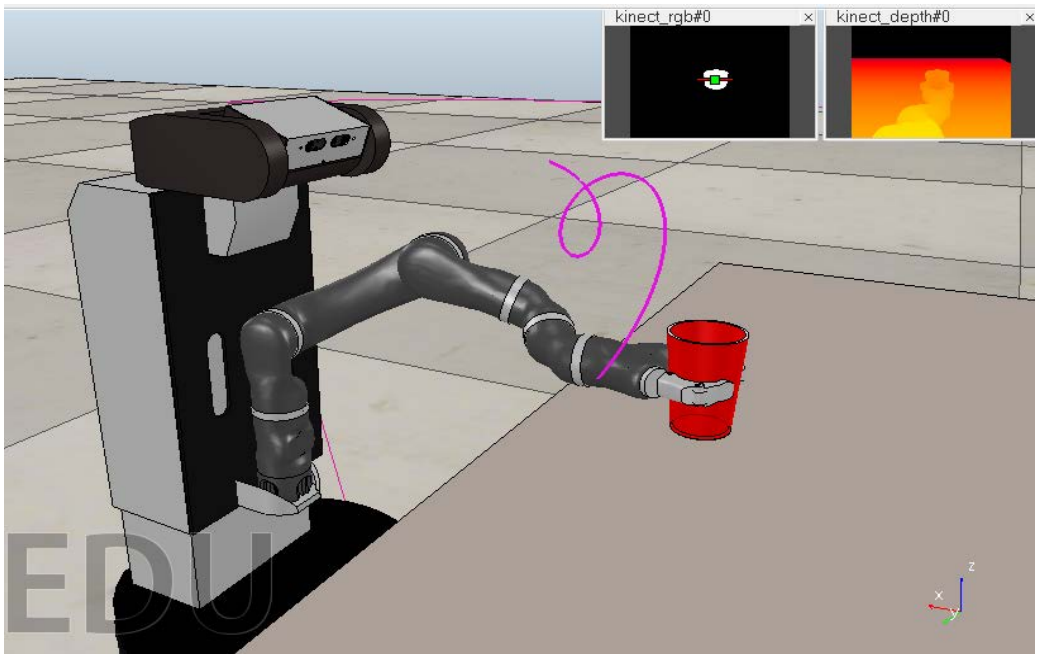


Ilustración 35. Ejecución del algoritmo de planificación de movimientos del brazo robótico incorporado en el RB-1

Cabe señalar que los algoritmos a los que se hace referencia en este apartado, se aportan como documentos anexos a esta memoria.

7. Determinación del modelo cinemático y dinámico del robot

7.1. Modelado cinemático

Este apartado surge desde la necesidad de valorar cómo afectan las velocidades angulares de las ruedas del RB-1 al movimiento global del mismo. Su objetivo es determinar un modelo cinemático que vincule la velocidad de las ruedas con la velocidad del robot.

Partimos de la definición de un origen global (XG, YG) y uno local (x,y). Como se muestra en la figura, se parte de la suposición de que el robot se mueve en el plano con una velocidad lineal expresada como: $v = (v_x, v_y, 0)^T$, y rota con una velocidad angular:

$\omega = (0,0,\omega_z)^T$.

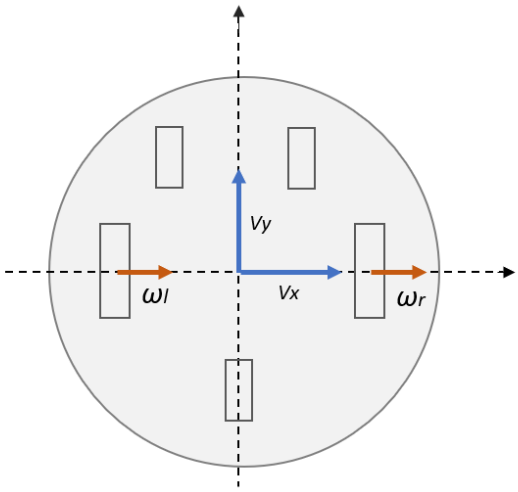


Ilustración 36. Velocidades angulares y lineales actuando en el RB-1

Siendo ωl y ωr las velocidades angulares de las rueda izquierda y derecha respectivamente, la relación cinemática se puede plantear del siguiente modo:

$$\begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix} = J_\omega \begin{bmatrix} \omega_l r \\ \omega_r r \end{bmatrix}$$

Siendo Jw:

$$J_\omega = \frac{1}{y_l - y_r} \begin{bmatrix} -y_r & y_l \\ x_G & -x_G \\ -1 & 1 \end{bmatrix}$$

Resultando en: $J_\omega = \frac{1}{2y_0}\begin{bmatrix} y_0 & y_0 \\ 0 & 0 \\ -1 & 1 \end{bmatrix}$ pues XG = 0, ya que el robot es simétrico.

Por tanto, el modelo quedaría de la siguiente manera:

$$\begin{cases} v_x = \frac{\omega_l r + \omega_r r}{2} = \frac{v_l + v_r}{2} \\ v_y = 0 \\ \omega_z = \frac{-\omega_l r + \omega_r r}{2y_0} = \frac{-v_l + v_r}{2y_0} \end{cases}$$

Esta es una aproximación sencilla, pues tiene su punto de partida en la cinemática skid steering o cinemática diferencial presente en robots de cuatro ruedas. Dicho modelo cinemático, se basa en el control de las velocidades relativas de los motores de los lados izquierdo y derecho simultáneamente, de forma que el giro se produce como resultado de la diferencia de velocidades entre un lado y otro.

La empresa Robotnik, comercializa un modelo de robot que sigue esta tipología de control de movimiento, denominado Summit.

Ilustración 37. Modelo Summit XL de la empresa Robotnik

Este robot se encuentra modelado en V-Rep, lo que nos permite analizar su sistema de movimiento. Así, se analizó la variación entre el modelo cinemático descrito anteriormente y los valores de Vx y Vy extraídos directamente de V-Rep.

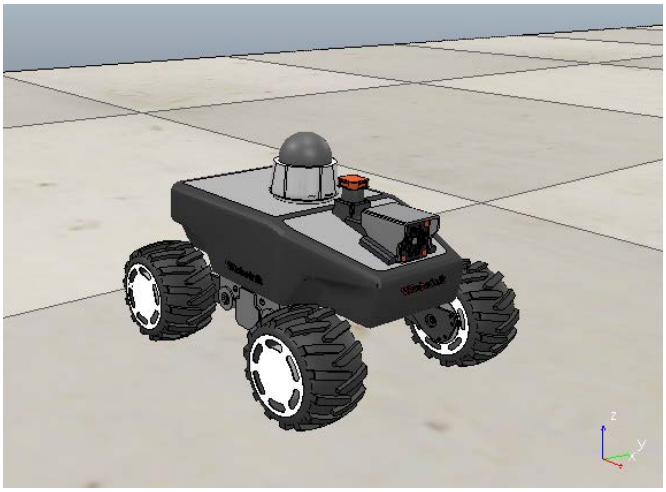


Ilustración 38. Robot SUMMIT XL modelado en V-Rep

En la siguiente gráfica se puede observar cómo el modelo cinemático se adapta de forma considerablemente aceptable a los datos extraídos de V-Rep. Sin embargo, se observa que para la velocidad de giro (alrededor de los 2 m/s), el modelo cinemático se distancia de los datos reales.

Ilustración 39. Comparación entre el modelo cinemático y la respuesta simulada del SUMMIT XL en V-Rep

De la misma forma, el modelo se adaptó para el RB-1 y se ejecutó el mismo algoritmo para poder valorar la adaptabilidad del modelo. En la siguiente gráfica se muestra que el modelo presentaba resultados similares a los obtenidos para el SUMMIT XL.

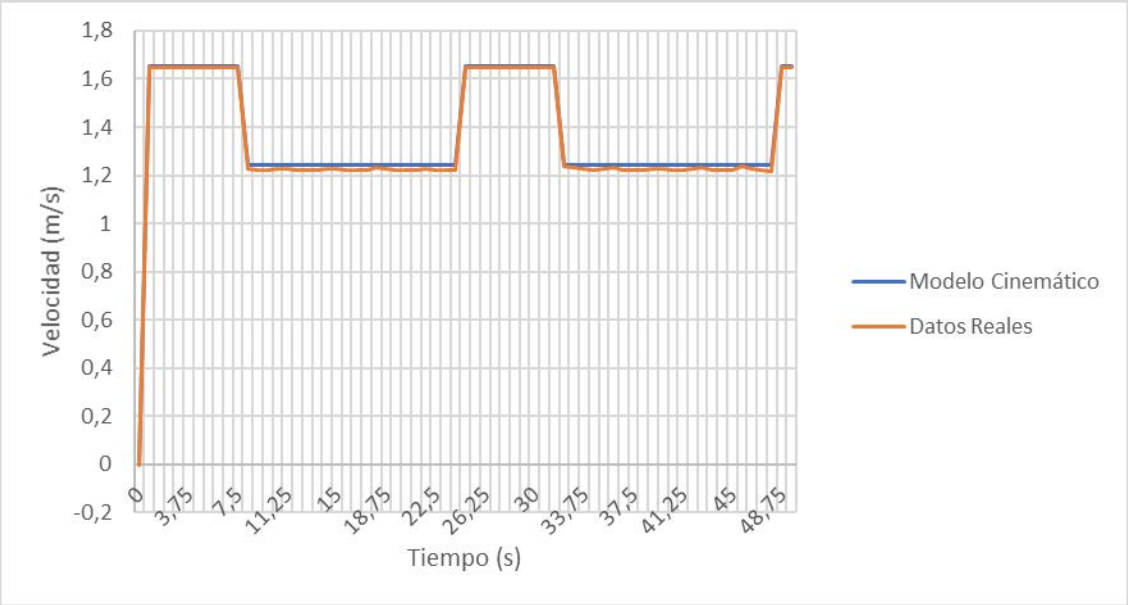


Ilustración 40. Comparación entre el modelo cinemático y la respuesta simulada del RB-1 en V-Rep

De esta forma, se puede concluir que desde el punto de vista cinemático, el error que resulta de la diferencia entre el modelo y los datos extraídos de la simulación puede considerarse despreciable, aceptando como válido el modelo cinemático expuesto anteriormente para el rango de velocidades ensayado. No obstante, teniendo en cuenta que la velocidad máxima de funcionamiento del RB-1 se encuentra en 1,5 m/s, el modelo resultará aceptable para todo su rango de velocidades.

7.2. Modelado dinámico

Para determinar el modelo dinámico del RB-1, fue necesario estudiar la teoría de la dinámica vehicular, en la que se engloban las diferentes fuerzas que interactúan durante la conducción de vehículos de cuatro ruedas.

En este sentido, será de interés el estudio de la dinámica vehicular en los estado de giro.

Es necesario diferenciar entre giros a baja velocidad y giros a alta velocidad, pues el comportamiento dinámico difiere notablemente.

7.2.1. Giros a baja velocidad

Durante la ejecución de giros a baja velocidad (< 2,5 m/s) las ruedas no experimentan fuerzas laterales y el centro de giro se establece en el mismo eje que las ruedas laterales. La cinemática de giro en automóviles se puede extrapolar al giro para vehículos robóticos con cinemática skid steering (como el SUMMIT XL) y a robots móviles con cinemática diferencial (como el RB-1).

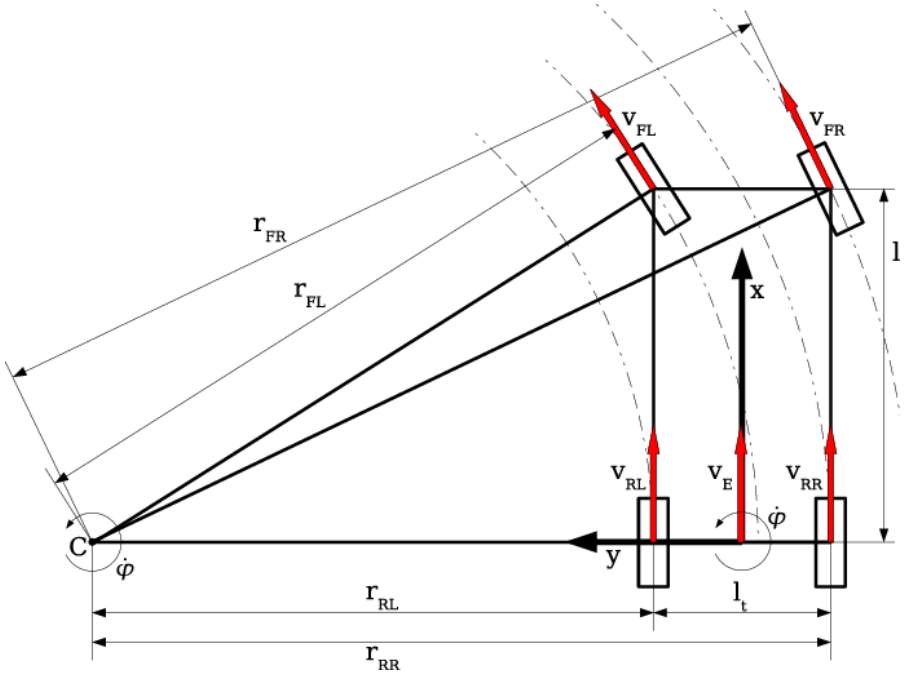


Ilustración 41. Descripción geométrica del giro de un automóvil según Ackerman

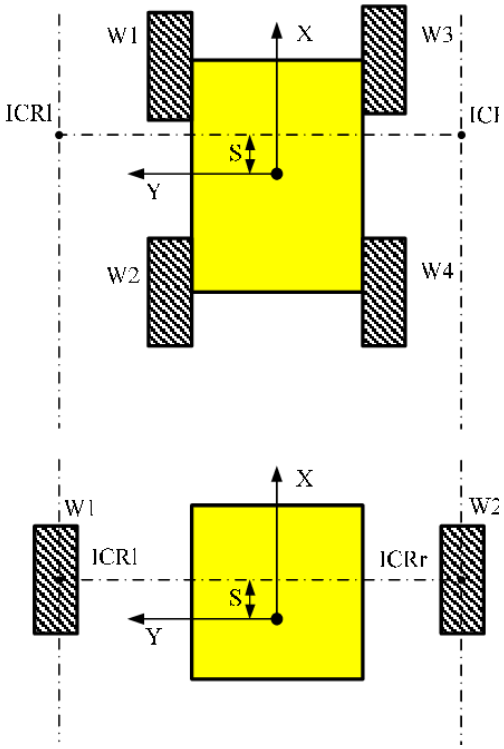


Ilustración 42. Descripción geométrica del giro de un vehículo con cinemática skid steering

En ambos modelos geométricos, el ángulo de deslizamiento resulta cero para este rango de velocidades y, en consecuencia, las ruedas no experimentan fuerzas laterales, como se comentaba anteriormente.

7.2.2. Giros a alta velocidad

Dentro de esta casuística, la aceleración lateral estará presente y por lo tanto, el ángulo de deslizamiento ya no será nulo, por lo que, como consecuencia aparecerá la denominada cornering force o fuerza lateral de curvatura. Esta fuerza es lineal para un rango de valores y directamente proporcional al ángulo de deslizamiento:

$Fy = C\alpha\ \alpha$

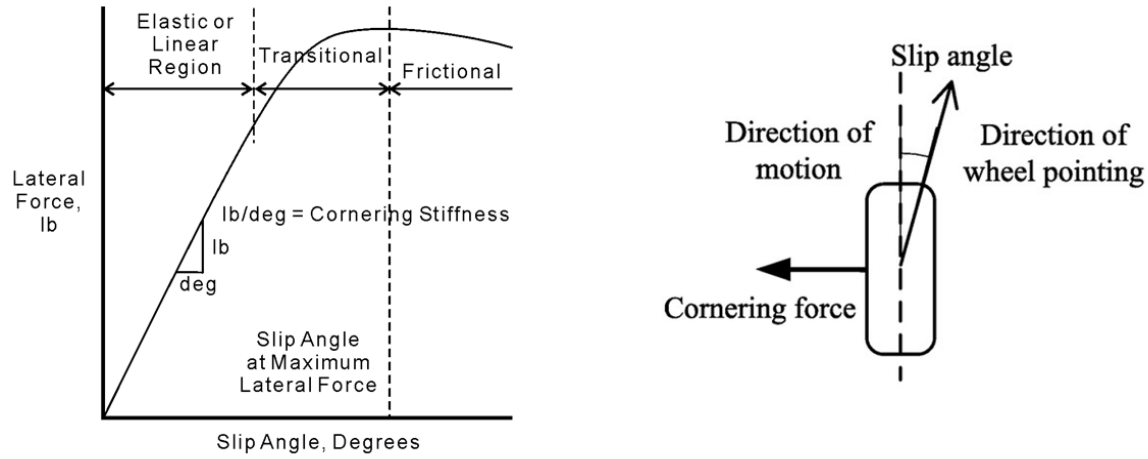


Ilustración 43. Relación entre la fuerza lateral en el centro de masas y el ángulo de deslizamiento

Siguiendo el modelo descrito en el libro Fundamentals of Vehicle Dynamics de Thomas D. Gillespie, las fuerzas laterales quedarían descritas de la siguiente forma:

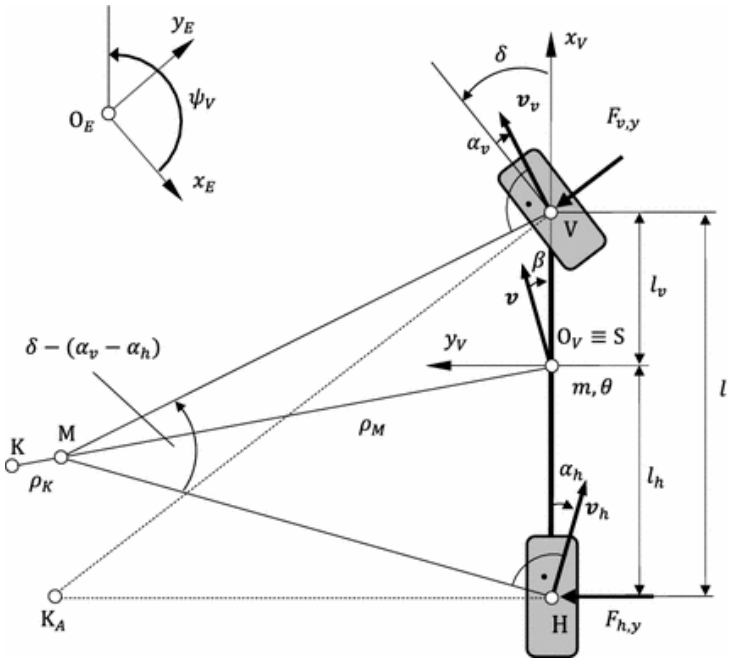


Ilustración 44. Fuerza lateral de curvatura en el modelo cinemático de una bicicleta

$Fv,y.lv – Fh,y.lh = 0$

$Fh,y = [M. lv / (lv + lh)]. (V^2 / R)$

$Fv,y = [M. lh / (lv + lh)]. (V^2 / R)$

Siendo:

Fv,y: Fuerza lateral en la rueda delantera

Fh,y: Fuerza lateral en la rueda trasera

lv : Distancia del centro de masas a la rueda delantera

lh : Distancia del centro de masas a la rueda delantera

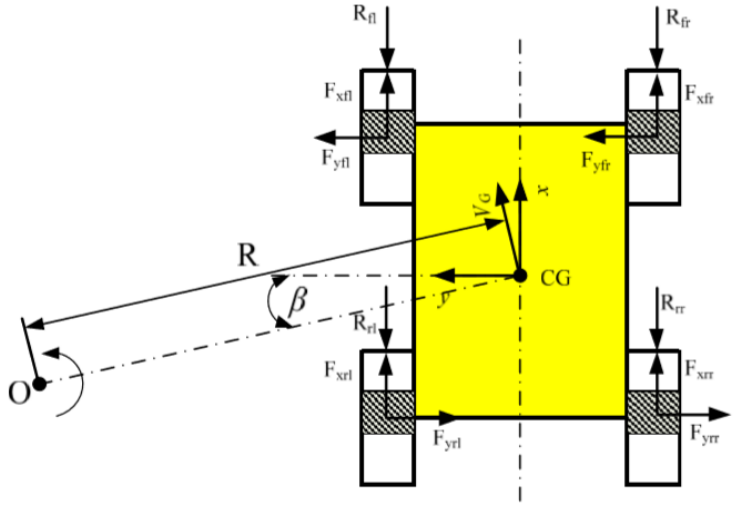
M: Masa del vehículo

V: Velocidad de avance

R: Radio de giro

Este modelo simplemente establece que la fuerza lateral que se experimenta en el eje trasero debe ser proporcional a la aceleración en ese punto e un número de veces igual a la masa cargada por el eje trasero.

Por otra parte, para tratar la dinámica de un robot con geometría skid steering se puede recurrir al modelo desarrollado por Wong (Theory of Ground Vehicles), en el que se apoya la publicación ISSN 1424-8220 de Tianmiao Wang et al. 2015 para trasladarlo a un ámbito experimental. En este modelo se describen las fuerzas que afectan a este tipo de robots en estados de giro.



$$\begin{cases} F_{xfr} + F_{xrr} + F_{xfl} + F_{xrl} - R_x - m\dfrac{v_G^2}{R} sin\,\beta = 0 \\ F_{yfr} + F_{yrr} + F_{yfl} + F_{yrl} = m\dfrac{v_G^2}{R} cos\,\beta \\ M_d - M_r = 0 \end{cases}$$

Ilustración 45. Fuerzas y momentos que actúan durante los estados de giro para robots con geometría skid steering

Ambos modelos, el de Gillespie y el de Wong, pueden combinarse en V-Rep para obtener la constante Cα. De esta manera, introduciendo el modelo de Wong en el script del SUMMIT XL, se tomarán diferentes valores de velocidad para determinados ángulos de deslizamiento.

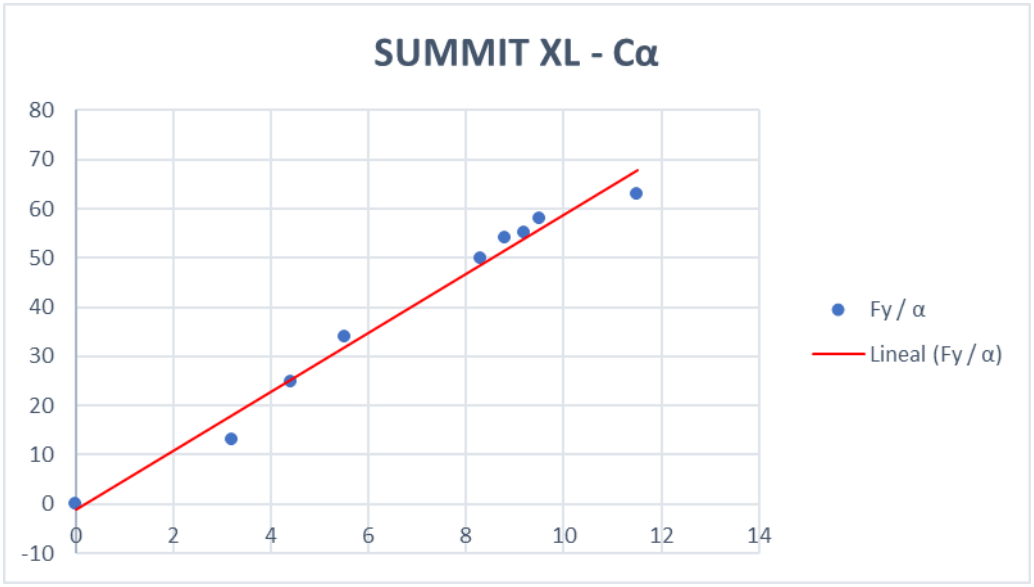


Ilustración 46. Relación entre Fuerzas laterales de curvatura (eje Y) y ángulo de deslizamiento (eje X) para el SUMMIT XL

Estos valores se tomaron para simulaciones a velocidades comprendidas en un rango de 2,5 a 3,5 m/s.

Se repitió el mismo proceso para el RB-1, para un rango de velocidades de 2 a 3 m/s. Donde los resultados se muestran en la siguiente gráfica:

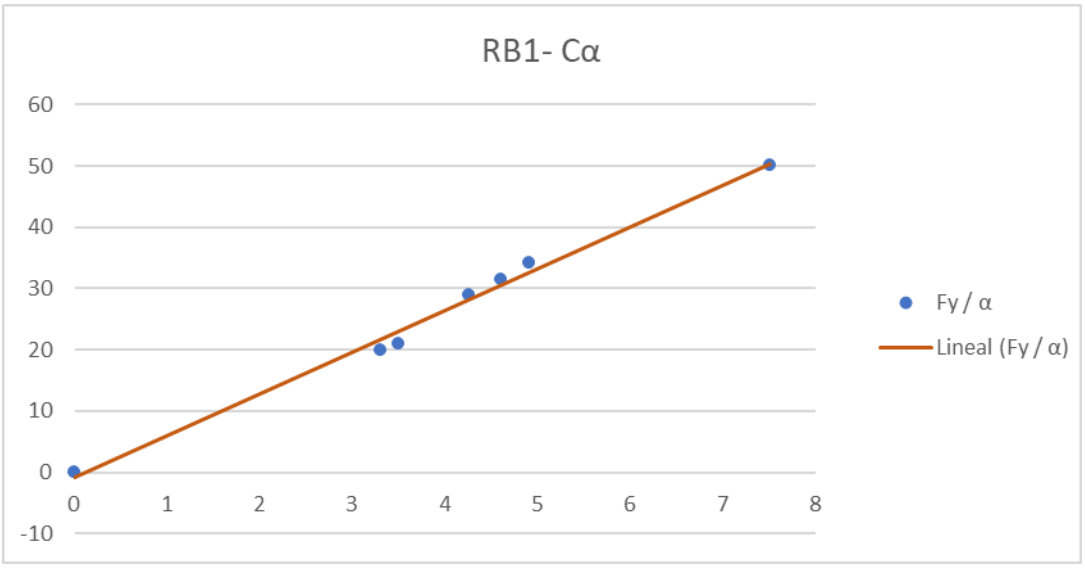


Ilustración 47. Relación entre Fuerzas laterales de curvatura (eje Y) y ángulo de deslizamiento (eje X) para el RB-1

De esta forma, se pueden relacionar los modelos descritos anteriormente. No obstante, es preciso señalar que para estos valores de velocidad angular en las ruedas, existía una varianza muy elevada, por lo que el error resultaba considerable.

Por último, es importante indicar que a pesar del interés teórico que se extrae de la simulación de estos modelos en V-Rep, para estos casos concretos, el rango de velocidades en el que tenían lugar las simulaciones era superior al rango de velocidad de funcionamiento de ambos robots, donde las fuerzas laterales que se generan en curva son despreciables.

8. Conclusiones

Se puede afirmar que en este proyecto se han obtenido los objetivos esperados de forma satisfactoria, pues se ha conseguido modelar el robot RB-1 de Robotnik en el software V-Rep a partir de su implementación en el sistema ROS. A continuación se enumeran los pasos que se han seguido para la obtención de este modelo:

- Se ha establecido el entorno software en una máquina virtual, de forma que se pudiera trabajar en V-Rep desde UNIX.
- Se ha efectuado un estudio pormenorizado de los componentes del RB-1, analizando los requisitos que debía cumplir el modelo para que las simulaciones se desarrollaran correctamente.
- Se ha construido el modelo en V-Rep, verificando que éste funcionaba correctamente a nivel mecánico, y que la respuesta de sus sensores de posición y de visión era correcta.

Además de completar estas actividades, en torno a las cuales giraba el fin último del proyecto, se han desarrollado tareas complementarias con el objetivo de ampliar los

conocimientos adquiridos durante el proyecto, y aprovechar el modelado del robot. De esta forma:

- Se han desarrollado diferentes algoritmos que interrelacionan el sistema locomotriz del robot, sus sensores de proximidad, su sistema de visión artificial y su brazo robótico, de forma que los diferentes elementos puedan interactuar en una misma simulación.
- Se ha modelado cinemáticamente el RB-1, paralelamente con el SUMMIT XL para poder comprobar la relación entre la cinemática diferencial y la cinemática skid steering. Además, se ha realizado una aproximación a un modelado dinámico en la que se han relacionado los modelos de Gillespie y Wong, con el objetivo de determinar la proporcionalidad entre la fuerza lateral y el ángulo de deslizamiento en estados de giro.

Por todo lo expuesto anteriormente, se concluye que el proyecto se ha finalizado de manera satisfactoria, cumpliendo los objetivos que se habían propuesto inicialmente, y añadiendo nuevos objetivos adicionales derivados del propio desarrollo del mismo.

9. Bibliografía

- V-Rep V.3.5.0 User Manual (2018)
- http://www.coppeliarobotics.com/helpFiles/
- Manual de ROS (2017):
- https://moodle2017-18.ua.es/moodle/mod/book/tool/print/index.php?id=2046
- ROS Wiki: es:
- http://wiki.ros.org/es
- Getting started with Ubuntu – CLEARPATH Robotics (2015)
- https://www.clearpathrobotics.com/assets/guides/ros/Getting%20Started%20with%20Ubuntu.html
- Fundamentals of vehicle Dynamics – Thomas D. Gillespie (1992)
- Theory of Ground Vehicles – J.Y. Wong (1978)
- Modelling and simulation of Skid-Steer wheeled vehicles – Coyler R.E., Economou J.T. (1998)
- Trajectory tracking control of a four-wheel differentialy driven mobile robot – Carraciolo L., De Luca A., Ianniti S. (1999)
- Kinetic modelo f a skid steered robot – Frantisek Solc, Jaroslav Sembera (2008)

10. Anexos

10.1 Script algoritmo RB-1 – Objetos. En este algoritmo, el RB-1 esquiva los objetos que encuentra a su paso.

```
function sysCall_threadmain()

    objHandle_2=sim.getObjectAssociatedWithScript(sim.handle_self)
    --laser1=sim.getObjectHandle("Hokuyo_URG_04LX_UG01_laser0")
    --laser2=sim.getObjectHandle("Hokuyo_UST_10LX_UG01_laser")
    laser1=sim.getObjectHandle("Proximity_sensor")
    laser2=sim.getObjectHandle("Proximity_sensor0")
    rb1=sim.getObjectHandle("rb1base")
    leftJoint_2=sim.getObjectHandle("left_wheel_joint")
    rightJoint_2=sim.getObjectHandle("right_wheel_joint")
    joint2_2=sim.getObjectHandle("passiveJoint_1")
    joint3_2=sim.getObjectHandle("passiveJoint_2")
    joint4_2=sim.getObjectHandle("passiveJoint_3")
    yaw=sim.getObjectHandle("pan_tilt_joint_1")
    pitch=sim.getObjectHandle("pan_tilt_joint_2")
    elevator=sim.getObjectHandle("Prismatic_joint")
    cam = sim.getObjectHandle("kinect_rgb")
    out = sim.auxiliaryConsoleOpen("Debug",10000,1,{1,1},{300,600})
        velocityLeft=0
        velocityRight=0
    simulationIsKinematic=false -- we want a dynamic simulation here
    ready = true


    s=sim.getObjectSizeFactor(objHandle_2)
    CM_x = 0
    CM_y = 0
```

```
    v0=0.4*s
    wheelDiameter=0.152*s
    wheelDiameter2=0.0936*s
    interWheelDistance=0.4333*s
    noDetectionDistance=1.5*s
    sensReading={noDetectionDistance,noDetectionDistance}
    sensReading2={0,0,0}
while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do
    res,dist,detected=sim.readProximitySensor(laser1)
    result,table_1,table_2 = sim.readVisionSensor(cam)
    if result>0 then
        CM_x = table_2[2]
        CM_y = table_2[3]
    end
    if (res>0) and (dist<noDetectionDistance) then
    sensReading[1]=dist
    sensReading2[1]=detected[1]
    sensReading2[2]=detected[2]
    sensReading2[3]=detected[3]
    if result>0 then
        CM_x = table_2[2]
        CM_y = table_2[3]
    end
    else sensReading[1]=noDetectionDistance end

    res,dist,detected=sim.readProximitySensor(laser2)
    if (res>0) and (dist<noDetectionDistance) then
    sensReading[2]=dist
    sensReading2[1]=detected[1]
    sensReading2[2]=detected[2]
    sensReading2[3]=detected[3]



    else sensReading[2]=noDetectionDistance end
```

```
    -- First calculate the normalized correction velocities:

    if (sensReading[1] < 0.75) or (sensReading[2] < 0.75) then
            velocityLeft=0.1*v0
            velocityRight=1*v0
            --sim.setIntegerSignal('yourTurnMico',1)

            --ready = false
            -- sim.waitForSignal('yourTurnBody')
            -- sim.waitForSignal('yourTurnBody')
    else if (ready == true) then
    -- Now combine the correction velocities with the default velocities (v0):
    --velocityLeft=v0*(1+2*velocityLeft)
    --velocityRight=v0*(1+2*velocityRight)
            velocityLeft=v0
            velocityRight=v0
    end
end
        -- Simulation is dynamic

p=sim.boolOr32(sim.getModelProperty(objHandle_2),sim.modelproperty_not_dynamic)-
sim.modelproperty_not_dynamic
        sim.setModelProperty(objHandle_2,p)
        sim.setJointTargetVelocity(leftJoint_2,velocityLeft*2/wheelDiameter)
        sim.setJointTargetVelocity(rightJoint_2,velocityRight*2/wheelDiameter)
        sim.setJointTargetVelocity(joint2_2,velocityRight*2/wheelDiameter)
        sim.setJointTargetVelocity(joint3_2,velocityRight*2/wheelDiameter)
        sim.setJointTargetVelocity(joint4_2,velocityRight*2/wheelDiameter)
        --sim.setJointTargetVelocity(yaw,0.5)
        --sim.setJointTargetVelocity(pitch,0.1)
        --sim.setJointTargetVelocity(elevator,0.1)
        -- print = printToConsole("velocityLeft: ", velocityLeft)
        -- print = printToConsole("velocityRight: ", velocityRight)
```

```
        -- print = printToConsole("Hokuyo_UTM_30LX_UG01_laser: ", sensReading[1])
        sim.auxiliaryConsolePrint(out, string.format("velocityLeft: %0.2f\n", velocityLeft))
        sim.auxiliaryConsolePrint(out, string.format("velocityRight: %0.2f\n", velocityRight))
        sim.auxiliaryConsolePrint(out, string.format("Position 1: %0.2f\n", sensReading[1]))
        sim.auxiliaryConsolePrint(out, string.format("Position 2: %0.2f\n", sensReading[2]))
        sim.auxiliaryConsolePrint(out, string.format("X = %0.2f, Y = %0.2f\n", CM_x, CM_y))
        sim.auxiliaryConsolePrint(out, string.format("Result: %d\n", result))
        -- print = printToConsole("infrared1_2: ", sensReading_2[4])
        -- print = printToConsole("infrared2_2: ", sensReading_2[5])

    end
end
```

10.2 Script algoritmo RB-1 – Vaso. En este algoritmo, el RB-1 se detiene junto a la mesa de forma que el brazo robótico pueda coger un vaso.

```
function sysCall_threadmain()

    objHandle_2=sim.getObjectAssociatedWithScript(sim.handle_self)
    --laser1=sim.getObjectHandle("Hokuyo_URG_04LX_UG01_laser0")
    --laser2=sim.getObjectHandle("Hokuyo_UST_10LX_UG01_laser")
    laser1=sim.getObjectHandle("Proximity_sensor")
    laser2=sim.getObjectHandle("Proximity_sensor0")
    rb1=sim.getObjectHandle("rb1base#0")
    leftJoint_2=sim.getObjectHandle("left_wheel_joint")
    rightJoint_2=sim.getObjectHandle("right_wheel_joint")
    wheel=sim.getObjectHandle('passiveJoint_1')
    wheel0=sim.getObjectHandle('passiveJoint_2')
    wheel1=sim.getObjectHandle('passiveJoint_3')
    yaw=sim.getObjectHandle("pan_tilt_joint_1")
    pitch=sim.getObjectHandle("pan_tilt_joint_2")
    elevator=sim.getObjectHandle("Prismatic_joint")
    cam = sim.getObjectHandle("kinect_rgb")
    out = sim.auxiliaryConsoleOpen("Debug",10000,1,{1,1},{300,600})
        velocityLeft=0
        velocityRight=0
    simulationIsKinematic=false -- we want a dynamic simulation here
    ready = true


    s=sim.getObjectSizeFactor(objHandle_2)
    CM_x = 0
    CM_y = 0
    v0=0.5*s
    wheelDiameter=0.152*s
    wheelDiameter2=0.0936*s
```

```
    interWheelDistance=0.4333*s
    noDetectionDistance=1.5*s
    sensReading={noDetectionDistance,noDetectionDistance}
    sensReading2={0,0,0}

while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do
    res,dist,detected=sim.readProximitySensor(laser1)
    result,table_1,table_2 = sim.readVisionSensor(cam)
    if result>0 then
        CM_x = table_2[2]
        CM_y = table_2[3]
    end
    if (res>0) and (dist<noDetectionDistance) then
    sensReading[1]=dist
    sensReading2[1]=detected[1]
    sensReading2[2]=detected[2]
    sensReading2[3]=detected[3]
    if result>0 then
        CM_x = table_2[2]
        CM_y = table_2[3]
    end
    else sensReading[1]=noDetectionDistance end

    res,dist,detected=sim.readProximitySensor(laser2)
    if (res>0) and (dist<noDetectionDistance) then
    sensReading[2]=dist
    sensReading2[1]=detected[1]
    sensReading2[2]=detected[2]
    sensReading2[3]=detected[3]



    else sensReading[2]=noDetectionDistance end

    -- First calculate the normalized correction velocities:
```

```
    if (sensReading[1] < 0.65) or (sensReading[2] < 0.65) then
            velocityLeft=0
            velocityRight=0
            sim.setIntegerSignal('yourTurnMico',1)

            ready = false
            -- sim.waitForSignal('yourTurnBody')
            -- sim.waitForSignal('yourTurnBody')
    else if (ready == true) then
    -- Now combine the correction velocities with the default velocities (v0):
    --velocityLeft=v0*(1+2*velocityLeft)
    --velocityRight=v0*(1+2*velocityRight)
            velocityLeft=v0
            velocityRight=v0
    end
end
        -- Simulation is dynamic
        p=sim.boolOr32(sim.getModelProperty(objHandle_2),sim.modelproperty_not_dynamic)-
sim.modelproperty_not_dynamic
        sim.setModelProperty(objHandle_2,p)
        sim.setJointTargetVelocity(leftJoint_2,velocityLeft*2/wheelDiameter)
        sim.setJointTargetVelocity(rightJoint_2,velocityRight*2/wheelDiameter)
        sim.setJointTargetVelocity(wheel,velocityLeft*2/wheelDiameter2)
        sim.setJointTargetVelocity(wheel0,velocityLeft*2/wheelDiameter2)
        sim.setJointTargetVelocity(wheel1,velocityLeft*2/wheelDiameter2)
        --sim.setJointTargetVelocity(yaw,0.5)
        sim.setJointTargetVelocity(pitch,0.1)
        --sim.setJointTargetVelocity(elevator,0.1)
        -- print = printToConsole("velocityLeft: ", velocityLeft)
        -- print = printToConsole("velocityRight: ", velocityRight)
        -- print = printToConsole("Hokuyo_UTM_30LX_UG01_laser: ", sensReading[1])
        sim.auxiliaryConsolePrint(out, string.format("velocityLeft: %0.2f\n", velocityLeft))
        sim.auxiliaryConsolePrint(out, string.format("velocityRight: %0.2f\n", velocityRight))
```

```
        sim.auxiliaryConsolePrint(out, string.format("Position: %0.2f\n", sensReading[2]))
        sim.auxiliaryConsolePrint(out, string.format("X = %0.2f, Y = %0.2f\n", CM_x, CM_y))
        sim.auxiliaryConsolePrint(out, string.format("Result: %d\n", result))
        -- print = printToConsole("infrared1_2: ", sensReading_2[4])
        -- print = printToConsole("infrared2_2: ", sensReading_2[5])

    end
end
```

10.3 Script algoritmo MICO – Vaso. Este es el algoritmo mediante el que el brazo robótico establece la planificación de movimientos y agarra el vaso.

```
visualizePath=function(path)
    if not _lineContainer then
        _lineContainer=sim.addDrawingObject(sim.drawing_lines,3,0,-1,99999,{1,0,1})
    end
    sim.addDrawingObjectItem(_lineContainer,nil)
    if path then
        forbidThreadSwitches(true)
        local initConfig=getConfig()
        local l=#jh
        local pc=#path/l
        for i=1,pc-1,1 do
            local config1={path[(i-1)*l+1],path[(i-1)*l+2],path[(i-1)*l+3],path[(i-1)*l+4],path[(i-1)*l+5],path[(i-1)*l+6]}
            local config2={path[i*l+1],path[i*l+2],path[i*l+3],path[i*l+4],path[i*l+5],path[i*l+6]}
            setConfig(config1)
            local lineDat=sim.getObjectPosition(ikTip,-1)
            setConfig(config2)
            local p=sim.getObjectPosition(ikTip,-1)
            lineDat[4]=p[1]
            lineDat[5]=p[2]
            lineDat[6]=p[3]
            sim.addDrawingObjectItem(_lineContainer,lineDat)
```

```
        end
        setConfig(initConfig)
        forbidThreadSwitches(false)
    end
    sim.switchThread()
end

displayInfo=function(txt)
    if dlgHandle then
        sim.endDialog(dlgHandle)
    end
    dlgHandle=nil
    if txt and #txt>0 then
        local dlgCols={0.5,0.7,0.5,0,0,0}
        dlgHandle=sim.displayDialog('Mico info',txt,sim.dlgstyle_message,false,nil,nil,dlgCols)
        sim.switchThread()
    end
end

_getJointPosDifference=function(startValue,goalValue,isRevolute)
    local dx=goalValue-startValue
    if (isRevolute) then
        if (dx>=0) then
            dx=math.mod(dx+math.pi,2*math.pi)-math.pi
        else
```

```
            dx=math.mod(dx-math.pi,2*math.pi)+math.pi
        end
    end
    return(dx)
end

_applyJoints=function(jointHandles,joints)
    for i=1,#jointHandles,1 do
        sim.setJointTargetPosition(jointHandles[i],joints[i])
    end
end

generatePathLengths=function(path)
    -- Returns a table that contains a distance along the path for each path point
    local d=0
    local l=#jh
    local pc=#path/l
    local retLengths={0}
    for i=1,pc-1,1 do
        local config1={path[(i-1)*l+1],path[(i-1)*l+2],path[(i-1)*l+3],path[(i-1)*l+4],path[(i-1)*l+5],path[(i-1)*l+6],path[(i-1)*l+7]}
        local config2={path[i*l+1],path[i*l+2],path[i*l+3],path[i*l+4],path[i*l+5],path[i*l+6],path[i*l+7]}
        d=d+getConfigConfigDistance(config1,config2)
        retLengths[i+1]=d
    end
```

```
    return retLengths
end

getShiftedMatrix=function(matrix,shift,dir,absoluteShift)
    -- Returns a pose or matrix shifted by vector shift
    local m={}
    for i=1,12,1 do
        m[i]=matrix[i]
    end
    if absoluteShift then
        m[4]=m[4]+dir*shift[1]
        m[8]=m[8]+dir*shift[2]
        m[12]=m[12]+dir*shift[3]
    else
        m[4]=m[4]+dir*(m[1]*shift[1]+m[2]*shift[2]+m[3]*shift[3])
        m[8]=m[8]+dir*(m[5]*shift[1]+m[6]*shift[2]+m[7]*shift[3])
        m[12]=m[12]+dir*(m[9]*shift[1]+m[10]*shift[2]+m[11]*shift[3])
    end
    return m
end

forbidThreadSwitches=function(forbid)

    if forbid then
        forbidLevel=forbidLevel+1
```

```
        if forbidLevel==1 then
            sim.setThreadAutomaticSwitch(false)
        end
    else
        forbidLevel=forbidLevel-1
        if forbidLevel==0 then
            sim.setThreadAutomaticSwitch(true)
        end
    end
end

findCollisionFreeConfig=function(matrix)

    sim.setObjectMatrix(ikTarget,-1,matrix)



    local cc=getConfig()
    local jointLimitsL={}
    local jointRanges={}
    for i=1,#jh,1 do
        jointLimitsL[i]=cc[i]-360*math.pi/180
        if jointLimitsL[i]<-10000 then jointLimitsL[i]=-10000 end
        jointRanges[i]=720*math.pi/180
        if cc[i]+jointRanges[i]>10000 then jointRanges[i]=10000-cc[i] end
    end
```

```
    jointLimitsL[2]=47*math.pi/180
    jointRanges[2]=266*math.pi/180
    jointLimitsL[3]=19*math.pi/180
    jointRanges[3]=322*math.pi/180
    local
c=sim.getConfigForTipPose(ikGroup,jh,0.65,10,nil,collisionPairs,nil,jointLimitsL,jointRange
s)
    return c
end

findSeveralCollisionFreeConfigs=function(matrix,trialCnt,maxConfigs)

    sim.setObjectMatrix(ikTarget,-1,matrix)
    local cc=getConfig()
    local cs={}
    local l={}
    for i=1,trialCnt,1 do
        local c=findCollisionFreeConfig(matrix)
        if c then
            local dist=getConfigConfigDistance(cc,c)
            local p=0
            local same=false
            for j=1,#l,1 do
                if math.abs(l[j]-dist)<0.001 then

                    same=true
```

```
                for k=1,#jh,1 do
                    if math.abs(cs[j][k]-c[k])>0.01 then
                        same=false
                        break
                    end
                end
            end
            if same then
                break
            end
        end
        if not same then
            cs[#cs+1]=c
            l[#l+1]=dist
        end
    end
    if #l>=maxConfigs then
        break
    end
  end
  if #cs==0 then
    cs=nil
  end
  return cs
end
```

```
getConfig=function()
    -- Returns the current robot configuration
    local config={}
    for i=1,#jh,1 do
        config[i]=sim.getJointPosition(jh[i])
    end
    return config
end

setConfig=function(config)
    -- Applies the specified configuration to the robot
    if config then
        for i=1,#jh,1 do
            sim.setJointPosition(jh[i],config[i])
        end
    end
end

getConfigConfigDistance=function(config1,config2)
    -- Returns the distance (in configuration space) between two configurations
    local d=0
    for i=1,#jh,1 do
        local dx=(config1[i]-config2[i])*metric[i]
        d=d+dx*dx
```

```
    end
    return math.sqrt(d)
end

getPathLength=function(path)
    -- Returns the length of the path in configuration space
    local d=0
    local l=#jh
    local pc=#path/l
    for i=1,pc-1,1 do
        local config1={path[(i-1)*l+1],path[(i-1)*l+2],path[(i-1)*l+3],path[(i-1)*l+4],path[(i-1)*l+5],path[(i-1)*l+6]}
        local config2={path[i*l+1],path[i*l+2],path[i*l+3],path[i*l+4],path[i*l+5],path[i*l+6]}
        d=d+getConfigConfigDistance(config1,config2)
    end
    return d
end

findPath=function(startConfig,goalConfigs,cnt)

    local jointLimitsL={}
    local jointLimitsH={}
    for i=1,#jh,1 do
        jointLimitsL[i]=startConfig[i]-360*math.pi/180
        if jointLimitsL[i]<-10000 then jointLimitsL[i]=-10000 end
```

```
        jointLimitsH[i]=startConfig[i]+360*math.pi/180
        if jointLimitsH[i]>10000 then jointLimitsH[i]=10000 end
    end
    jointLimitsL[2]=47*math.pi/180
    jointLimitsH[2]=313*math.pi/180
    jointLimitsL[3]=19*math.pi/180
    jointLimitsH[3]=341*math.pi/180

    local task=simOMPL.createTask('task')
    simOMPL.setAlgorithm(task,OMPLAlgo)
    local jSpaces={}
    for i=1,#jh,1 do
        local proj=i
        if i>3 then proj=0 end

jSpaces[#jSpaces+1]=simOMPL.createStateSpace('j_space'..i,simOMPL.StateSpaceType.
joint_position,jh[i],{jointLimitsL[i]},{jointLimitsH[i]},proj)
    end
    simOMPL.setStateSpace(task,jSpaces)
    simOMPL.setCollisionPairs(task,collisionPairs)
    simOMPL.setStartState(task,startConfig)
    simOMPL.setGoalState(task,goalConfigs[1])
    for i=2,#goalConfigs,1 do
        simOMPL.addGoalState(task,goalConfigs[i])
    end
    local path=nil
```

```
    local l=999999999999
--    forbidThreadSwitches(true)
    for i=1,cnt,1 do
        local res,_path=simOMPL.compute(task,maxOMPLCalculationTime,-1,200)
        if res and _path then
            local _l=getPathLength(_path)
            if _l<l then
                l=_l
                path=_path
            end
        end
        if path then
            visualizePath(path)
        end
    end
--    forbidThreadSwitches(false)
    simOMPL.destroyTask(task)
    return path,l
end

findShortestPath=function(startConfig,goalConfigs,searchCntPerGoalConfig)
    -- This function will search for several paths between the specified start configuration,
    -- and several of the specified goal configurations. The shortest path will be returned
    local
onePath,onePathLength=findPath(startConfig,goalConfigs,searchCntPerGoalConfig)
```

```
    return onePath,generatePathLengths(onePath)
end

generateIkPath=function(startConfig,goalPose,steps,ignoreCollisions)
    -- Generates (if possible) a linear, collision free path between a robot config and a target
pose
    forbidThreadSwitches(true)
    local currentConfig=getConfig()
    setConfig(startConfig)
    sim.setObjectMatrix(ikTarget,-1,goalPose)
    local coll=collisionPairs
    if ignoreCollisions then
        coll=nil
    end
    local c=sim.generateIkPath(ikGroup,jh,steps,coll)
    setConfig(currentConfig)
    forbidThreadSwitches(false)
    if c then
        return c, generatePathLengths(c)
    end
end

executeMotion=function(path,lengths,maxVel,maxAccel,maxJerk)
    dt=sim.getSimulationTimeStep()
```

```
    jointsUpperVelocityLimits={}
    for j=1,6,1 do

res,jointsUpperVelocityLimits[j]=sim.getObjectFloatParameter(jh[j],sim.jointfloatparam_upper_limit)
    end
    velCorrection=1


    sim.setThreadSwitchTiming(200)
    while true do
        posVelAccel={0,0,0}
        targetPosVel={lengths[#lengths],0}
        pos=0
        res=0
        previousQ={path[1],path[2],path[3],path[4],path[5],path[6]}
        local rMax=0
        rmlHandle=sim.rmlPos(1,0.0001,-1,posVelAccel,{maxVel*velCorrection,maxAccel,maxJerk},{1},targetPosVel)
        while res==0 do
            res,posVelAccel,sync=sim.rmlStep(rmlHandle,dt)
            if (res>=0) then
                l=posVelAccel[1]
                for i=1,#lengths-1,1 do
                    l1=lengths[i]
                    l2=lengths[i+1]
                    if (l>=l1)and(l<=l2) then
```

```
                t=(l-l1)/(l2-l1)
                for j=1,6,1 do
                    q=path[6*(i-1)+j]+_getJointPosDifference(path[6*(i-
1)+j],path[6*i+j],jt[j]==sim.joint_revolute_subtype)*t

dq=_getJointPosDifference(previousQ[j],q,jt[j]==sim.joint_revolute_subtype)
                    previousQ[j]=q
                    r=math.abs(dq/dt)/jointsUpperVelocityLimits[j]
                    if (r>rMax) then
                        rMax=r
                    end
                end
                break
            end
        end
    end
    sim.rmlRemove(rmlHandle)
    if rMax>1.001 then
        velCorrection=velCorrection/rMax
    else
        break
    end
end
sim.setThreadSwitchTiming(2)
```

```
    -- 2. Execute the movement:
    posVelAccel={0,0,0}
    targetPosVel={lengths[#lengths],0}
    pos=0
    res=0
    jointPos={}
    rmlHandle=sim.rmlPos(1,0.0001,-
1,posVelAccel,{maxVel*velCorrection,maxAccel,maxJerk},{1},targetPosVel)
    while res==0 do
        dt=sim.getSimulationTimeStep()
        res,posVelAccel,sync=sim.rmlStep(rmlHandle,dt)
        if (res>=0) then
            l=posVelAccel[1]
            for i=1,#lengths-1,1 do
                l1=lengths[i]
                l2=lengths[i+1]
                if (l>=l1)and(l<=l2) then
                    t=(l-l1)/(l2-l1)
                    for j=1,6,1 do
                        jointPos[j]=path[6*(i-1)+j]+_getJointPosDifference(path[6*(i-
1)+j],path[6*i+j],jt[j]==sim.joint_revolute_subtype)*t
                    end
                    _applyJoints(jh,jointPos)
                    break
                end
            end
```

```
        end
        sim.switchThread()
    end
    sim.rmlRemove(rmlHandle)
end

savePath=function(filename,path,lengths)
    sim.writeCustomDataBlock(micoHandle,filename..'.pathData1',sim.packFloatTable(path))

sim.writeCustomDataBlock(micoHandle,filename..'.pathLength1',sim.packFloatTable(lengt
hs))
end

loadPath=function(filename)
    path=sim.readCustomDataBlock(micoHandle,filename..'.pathData1')
    if (not path) then return nil end
    path=sim.unpackFloatTable(path)
    lengths=sim.readCustomDataBlock(micoHandle,filename..'.pathLength1')
    if (not lengths) then return nil end
    lengths=sim.unpackFloatTable(lengths)
    return path,lengths
end

function sysCall_threadmain()
    jh={-1,-1,-1,-1,-1,-1}
    jt={-1,-1,-1,-1,-1,-1}
```

```
    for i=1,6,1 do
        jh[i]=sim.getObjectHandle('Mico_joint'..i)
        jt[i]=sim.getJointType(jh[i])
    end
    micoHandle=sim.getObjectHandle('Mico')
    ikTarget=sim.getObjectHandle('Mico_target')
    ikTip=sim.getObjectHandle('Mico_tip')
    ikGroup=sim.getIkGroupHandle('Mico_ik')
    target0=sim.getObjectHandle('micoTarget0')
    target1=sim.getObjectHandle('micoTarget1')

collisionPairs={sim.getCollectionHandle('Mico'),sim.getCollectionHandle('Mico'),sim.getCollectionHandle('Mico'),sim.handle_all}
    maxVel=1
    maxAccel=1
    maxJerk=8000
    forbidLevel=0
    metric={0.2,1,0.8,0.1,0.1,0.1}
    ikSteps=20
    maxOMPLCalculationTime=6 -- for one calculation. Higher is better, but takes more time
    OMPLAlgo=simOMPL.Algorithm.BKPIECE1 -- the OMPL algorithm to use
    numberOfOMPLCalculationsPasses=4 -- the number of OMPL calculation runs for a same goal config. The more, the better results, but slower
        sim.waitForSignal('yourTurnMico')

    -- Move to the first cup (with motion planning):
```

```
    path,lengths=loadPath('MicoPath_1')
    if not path then
        local m=getShiftedMatrix(sim.getObjectMatrix(target1,-1),{0.08,0,0},1,true)
        displayInfo('searching for several valid goal configurations...')
        local configs=findSeveralCollisionFreeConfigs(m,300,5)
        displayInfo('searching for several valid paths between the current configuration and
found goal configurations...')

path,lengths=findShortestPath(getConfig(),configs,numberOfOMPLCalculationsPasses)
        displayInfo(nil)
        if path then
            savePath('MicoPath_1',path,lengths)
        end
    end
    if path then
        visualizePath(path)
        executeMotion(path,lengths,maxVel,maxAccel,maxJerk)
    end

    -- Move closer to the cup (with IK):
    path,lengths=loadPath('MicoPath_2')
    if not path then
        local m=getShiftedMatrix(sim.getObjectMatrix(target1,-1),{-0.04,0,0},1,true)
        path,lengths=generateIkPath(getConfig(),m,ikSteps,true)
        if path then
            savePath('MicoPath_2',path,lengths)
```

```
        end
    end
    if path then
        executeMotion(path,lengths,maxVel,maxAccel,maxJerk)
    end

    -- close the hand
    sim.setIntegerSignal("hand2",1)
    sim.wait(1.25)

    -- sim.setIntegerSignal('yourTurnBody',1)
        path,lengths=loadPath('MicoPath_3')
    if not path then
        local m=getShiftedMatrix(sim.getObjectMatrix(target1,-1),{0,0,0.1},1,true)
        path,lengths=generateIkPath(getConfig(),m,ikSteps,true)
        if path then
            savePath('MicoPath_3',path,lengths)
        end
    end
    if path then
        executeMotion(path,lengths,maxVel,maxAccel,maxJerk)
    end

end
```

10.4 Script algoritmo SUMMIT – Dynamics. En este algoritmo, se determina la fuerza lateral y el ángulo de deslizamiento en curva para una velocidad determinada.

```
function sysCall_threadmain()
    motorHandles={-1,-1,-1,-1}
    ww={-1,-1,-1,-1}
    out = sim.auxiliaryConsoleOpen("Debug",10000,1,{1,1},{300,400})
    motorHandles[1]=sim.getObjectHandle('joint_front_left_wheel')
    motorHandles[2]=sim.getObjectHandle('joint_front_right_wheel')
    motorHandles[3]=sim.getObjectHandle('joint_back_right_wheel')
    motorHandles[4]=sim.getObjectHandle('joint_back_left_wheel')
    ww[4]=sim.getObjectHandle('SummitXLRearLeftWheel')
    ww[3]=sim.getObjectHandle('SummitXLRearRightWheel')
    ww[1]=sim.getObjectHandle('SummitXLFrontLeftWheel')
    ww[2]=sim.getObjectHandle('SummitXLFrontRightWheel')
    force1=sim.getObjectHandle('Accelerometer_forceSensor1')
    force2=sim.getObjectHandle('Accelerometer_forceSensor2')
    force3=sim.getObjectHandle('Accelerometer_forceSensor3')
    force4=sim.getObjectHandle('Accelerometer_forceSensor4')
    body = sim.getObjectHandle('Robotnik_Summit_XL')
    reference = sim.getObjectHandle('ResizableFloor_25_100_element')

    --  Parameters
    L = 1.3561e-01    -- Base of the wheel
```

```
    r = 2.3367e-01/2  -- Radius of the wheel
    a = 0.482845      -- Wheel track
    b = 0.468004      -- Wheel base
    C = 5000          -- Stiffness of the tyre
    u = 0.061         -- Coefficient of friction
    W = 45            -- Weight of the robot
    p = 0.5           -- <0,1> Specifies position of CG with respect to GC
    const = (u*W)/C

while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do

    --  Translation loop
    for i=1,10,1 do
        w1 = 1245.00*math.pi/180
        w2 = -1245.00*math.pi/180
        w3 = -1245.00*math.pi/180
        w4 = 1245.00*math.pi/180
        velocity2 = (w1*r - w2*r)/2
        sim.setJointTargetVelocity(motorHandles[1],w1)
        sim.setJointTargetVelocity(motorHandles[2],w2)
        sim.setJointTargetVelocity(motorHandles[3],w3)
        sim.setJointTargetVelocity(motorHandles[4],w4)

        V_body, W_body = sim.getObjectVelocity(body)
        Vx = V_body[1]
```

```
        Vy = V_body[2]
        angle = sim.getObjectOrientation(body,reference)
        angle_vel = 180*(math.atan(Vy/Vx))/math.pi
        d = (w1*r+w2*r)
        alpha = math.atan(d/b)
        RoT = ((w1*r)/math.tan(alpha)) - (b/2)
        angle_x = 180*(angle[3]/math.pi)
        velocity = Vx*math.cos(angle[3]) + Vy*math.sin(angle[3])                -- Longitudinal
velocity of the robot
        JointForce1 = sim.getJointForce(motorHandles[1])
        Fy = W*(velocity^2)/(RoT)
        Fyr = (W*(a/2)*(velocity^2))/(RoT*L)
        slip_angle = 0

        sim.auxiliaryConsolePrint(out, string.format("Angle body: %0.4f\n", velocity))

        sim.wait(0.75)
    end

    --  Curve loop
    for i=1,22,1 do
        w1 = 480*math.pi/180
        w2 = -1245*math.pi/180
        w3 = -1245*math.pi/180
        w4 = 480*math.pi/180
```

```
velocity2 = (w1*r - w2*r)/2
sim.setJointTargetVelocity(motorHandles[1],w1)
sim.setJointTargetVelocity(motorHandles[2],w2)
sim.setJointTargetVelocity(motorHandles[3],w3)
sim.setJointTargetVelocity(motorHandles[4],w4)
num1, force_sensor1 = sim.readForceSensor(force1)
num2, force_sensor2 = sim.readForceSensor(force2)
num3, force_sensor3 = sim.readForceSensor(force3)
num4, force_sensor4 = sim.readForceSensor(force4)
V_body, W_body = sim.getObjectVelocity(body)
Vx = V_body[1]
Vy = V_body[2]
angle = sim.getObjectOrientation(body,reference)
angle_vel = 180*(math.atan(Vx/Vy))/math.pi
if Vy > 0 then
angle_vel2 = 90 - angle_vel
else angle_vel2 = 270 - angle_vel
end
d = (w1*r+w2*r)
alpha = math.atan(d/b)
RoT = ((w1*r)/math.tan(alpha)) - (b/2)
angle_x = 180*(angle[3]/math.pi)
if angle_x < 0 then
angle_xx = 360 + angle_x
else angle_xx = angle_x
```

```
        end
        velocity = Vx*math.cos(angle[3]) + Vy*math.sin(angle[3])
        JointForce1 = sim.getJointForce(motorHandles[1])

        Fyr = (W*(a/2)*(velocity^2))/(RoT*a)
        slip_angle = angle_xx - angle_vel2
        slip_angle2 = math.pi*slip_angle/180
        Fy = W*(velocity^2)*math.sin(slip_angle2)/(RoT)
        Fy_exp = force_sensor1[2] + force_sensor2[2] + force_sensor3[2] + force_sensor4[2]

        sim.auxiliaryConsolePrint(out, string.format("Slip angle: %0.4f\n", slip_angle))
        sim.auxiliaryConsolePrint(out, string.format("Fy: %0.4f\n", Fy))

        sim.wait(0.75)
    end

    end
end
```

10.5 Script algoritmo RB-1 – Dynamics. En este algoritmo, se determina la fuerza lateral y el ángulo de deslizamiento en curva para una velocidad determinada.

```
 function sysCall_threadmain()
    motorHandles={-1,-1,-1,-1}
```

```
    ww={-1,-1,-1,-1}
    out = sim.auxiliaryConsoleOpen("Debug",10000,1,{1,1},{300,400})
    motorHandles[1]=sim.getObjectHandle('left_wheel_joint#0')
    motorHandles[2]=sim.getObjectHandle('right_wheel_joint#0')

    body = sim.getObjectHandle('rb1base_link_respondable#0')
    reference = sim.getObjectHandle('ResizableFloor_25_100_element#0')

    --  Parameters
    L = 1.3561e-01    -- Base of the wheel
    r = 2.3367e-01/2  -- Radius of the wheel
    a = 0.482845      -- Wheel track
    b = 0.433      -- Wheel base
    C = 5000          -- Stiffness of the tyre
    u = 0.061         -- Coefficient of friction
    W = 54            -- Weight of the robot
    p = 0.5          -- <0,1> Specifies position of CG with respect to GC
    const = (u*W)/C

while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do

    --  Translation loop
    for i=1,4,1 do
       w1 = 2100.00*math.pi/180
       w2 = 2100.00*math.pi/180
```

```
        velocity2 = (w1*r - w2*r)/2
        sim.setJointTargetVelocity(motorHandles[1],w1)
        sim.setJointTargetVelocity(motorHandles[2],w2)
        V_body, W_body = sim.getObjectVelocity(body)
        Vx = V_body[1]
        Vy = V_body[2]
        angle = sim.getObjectOrientation(body,reference)
        angle_vel = 180*(math.atan(Vy/Vx))/math.pi
        d = (w1*r+w2*r)
        alpha = math.atan(d/b)
        RoT = ((w1*r)/math.tan(alpha)) - (b/2)
        angle_x = 180*(angle[3]/math.pi)
        velocity = Vx*math.cos(angle[3]) + Vy*math.sin(angle[3])                -- Longitudinal
velocity of the robot

        Fy = W*(velocity^2)/(RoT)
        Fyr = (W*(a/2)*(velocity^2))/(RoT*L)

        sim.auxiliaryConsolePrint(out, string.format("velocity Real: %0.6f\n", velocity))
--        sim.auxiliaryConsolePrint(out, string.format("velocity Modelo: %0.6f\n", velocity2))

        sim.wait(0.75)
    end
```

```
--  Curve loop
for i=1,22,1 do
    w1 = 1100*math.pi/180
    w2 = 2100*math.pi/180

    velocity2 = (w1*r - w2*r)/2
    sim.setJointTargetVelocity(motorHandles[1],w1)
    sim.setJointTargetVelocity(motorHandles[2],w2)
    V_body, W_body = sim.getObjectVelocity(body)
    Vx = V_body[1]
    Vy = V_body[2]
    angle = sim.getObjectOrientation(body,reference)
    angle_vel = 180*(math.atan(Vx/Vy))/math.pi
    if Vy > 0 then
    angle_vel2 = 90 - angle_vel
    else angle_vel2 = 270 - angle_vel
    end
    d = (w2*r-w1*r)
    v1 = w1*r
    v2 = w2*r
    alpha = math.atan(d/b)
    RoT = ((w1*r)/math.tan(alpha)) - (b/2)
    angle_x = 180*(angle[3]/math.pi)
    if angle_x < 0 then
    angle_xx = 360 + angle_x
```

```
        else angle_xx = angle_x
        end
        velocity = Vx*math.cos(angle[3]) + Vy*math.sin(angle[3])
        JointForce1 = sim.getJointForce(motorHandles[1])

        Fyr = (W*(a/2)*(velocity^2))/(RoT*a)
        slip_angle = angle_xx - angle_vel2
        slip_angle2 = math.pi*slip_angle/180
        Fy = W*(velocity^2)*math.sin(slip_angle2)/(RoT)

--        Fy_exp = force_sensor1[2] + force_sensor2[2] + force_sensor3[2] +
force_sensor4[2]
--        sim.auxiliaryConsolePrint(out, string.format("Radius of turn: %0.4f\n", Fyr))
        sim.auxiliaryConsolePrint(out, string.format("Fy: %0.4f\n", Fy))
--        sim.auxiliaryConsolePrint(out, string.format("Angle: %0.4f\n", slip_angle))

        sim.wait(0.75)
    end
    end
end
```