UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica y Automática Industrial

TRABAJO FIN DE GRADO

SIMULACIÓN CINEMÁTICA Y DINÁMICA DEL ROBOT HEXÁPODO ESCALADOR ROMERIN

Pablo Martínez Campos

Cotutor: Dr. Alberto Brunete González
Departamento: Ingeniería Eléctrica, Eléctrónica Automática y Física Aplicada

Tutor: Dr. Miguel Hernando Gutiérrez
Departamento: Ingeniería Eléctrica, Eléctrónica Automática y Física Aplicada

Madrid, Febrero, 2019

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica y Automática Industrial

TRABAJO FIN DE GRADO

SIMULACIÓN CINEMÁTICA Y DINÁMICA DEL ROBOT HEXÁPODO ESCALADOR ROMERIN

Firma Autor

Firma Cotutor *Firma Tutor*

Copyright ©2019. Pablo Martínez Campos
Esta obra está licenciada bajo la licencia Creative Commons
Atribución-NoComercial-SinDerivadas 3.0 Unported (CC BY-NC-ND 3.0). Para ver una copia de esta licencia, visite http://creativecommons.org/licenses/by-nc-nd/3.0/deed.es o envíe una carta a Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, EE.UU.
Todas las opiniones aquí expresadas son del autor, y no reflejan necesariamente las opiniones de la Universidad Politécnica de Madrid.

Título: Simulación cinemática y dinámica del robot hexápodo escalador ROMERIN
Autor: Pablo Martínez Campos
Tutor: Dr. Miguel Hernando Gutiérrez
Cotutor: Dr. Alberto Brunete González

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el día ..... de ..... de ... en ....., en la Escuela Técnica Superior de Ingeniería y Diseño Industrial de la Universidad Politécnica de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

En primer lugar, quiero agradecer a mis padres todo el apoyo y el cariño que me han brindado siempre, respetando mis decisiones y ayudándome en los momentos más complicados de mi vida, siempre dándome ánimos para seguir hacia delante a pesar de todos los baches que me he encontrado a lo largo del camino. Sin ellos no sería posible estar aquí, cerrando una etapa tan importante. También quiero agradecer a mi hermano la alegría y el cariño que siempre me da cada día.

Le doy las gracias al resto de mi familia, en especial a mi abuelo, por darme siempre buenos consejos y apoyarme cada vez que le veo.

Por otro lado, también quiero agradecer a Laura, que ha sido un gran apoyo para mí a lo largo de casi toda la carrera y que ha conseguido que los palos fuesen menos dolorosos y las interminables jornadas en la universidad fuesen mucho más llevaderas.

Gracias a mi compañero Juan Luis, con quien he compartido muchas de las frustraciones que este proyecto me ha dado, y que me ha ofrecido su ayuda en todo momento.

Por último, pero no por ello menos importante, quiero darle las gracias a muchos de los profesores que he tenido a lo largo de mi carrera académica, especialmente a Adolfo Barreno, por enseñarme valores que transcienden los contenidos de una asignatura concreta, y a Miguel Hernando, Cecilia García y Pablo San Segundo, por su paciencia y por conseguir despertar en mí un interés en ciertas materias que nunca imaginé que tendría.

Resumen

Este proyecto surge de la necesidad de simular el comportamiento de un robot hexápodo del cual se dispone previamente de un prototipo real. Para ello, se ha decidido utilizar el simulador de robots V-REP, desarrollado por la empresa *Coppelia Robotics*.

Puesto que este robot tendrá la compleja tarea de inspección de conductos de difícil o nulo acceso, es necesario que el modelo simulado responda correctamente a los diferentes imprevistos que pueda encontrarse en su camino, así como ser capaz de moverse por paredes o techos con gran fiabilidad.

El objetivo principal es crear el robot en el simulador mencionado, dotándolo de las características propias del hexápodo real, para que responda de forma similar a la realidad.

En el modelo creado dentro del simulador se llevarán a cabo varios tipos de pruebas e implementaciones y, una vez comprobado su funcionamiento, se procederá a su integración en el robot real.

A su vez, este desarrollo sentará las bases para la posterior creación e implementación de un robot escalador modular, también destinado a tareas de inspección, el cual podrá unir diferentes módulos para crear un robot con características únicas y fácilmente modificables.

Palabras clave: V-REP, robot hexápodo, robot escalador.

Abstract

This project arises from the necessity of simulating the behaviour of an hexapod robot which had a previous real prototype. To achieve this, it has been decided to use the V-REP robot simulator developed by Coppelia Robotics company.

This robot will have the complex task of inspecting difficult or null access pipes, so it is required to respond correctly to the different unexpected events, as well as being able to climb up the walls and roofs with a great reliability.

The main goal is to develop the robot in the afore mentioned simulator, giving it the features of the real hexapod so that it responds in a similar way to reality.

In the model created on the simulator, several types of tests and implementations will be carried out and implemented in the real robot after verifying its funcionality.

At the same time, this project will lay down the basis for the future creation and implementation of a modular climbing robot, which will be destined for inspection tasks and will be able to join different modules in order to create an unique and adaptable robot.

Keywords: V-REP, hexapod robot, climbing robot.

Índice general

Agradecimientos ix

Resumen xi

Abstract xiii

Índice xvi

1. Introducción **1**
1.1. Motivación del proyecto 1
1.2. Objetivos 1
1.3. Materiales utilizados 2
1.4. Planificación del proyecto 2
1.5. Estructura del documento 4

2. Estado del arte **5**
2.1. Robots escaladores 5
2.1.1. Clasificación según la superficie de desplazamiento 5
2.1.2. Mecanismos de sujeción 7
2.1.3. Campos de aplicación 8
2.2. Simuladores de robots 12
2.2.1. Simuladores disponibles en el mercado 13
2.2.2. Comparativa entre los simuladores más completos: V-REP, Gazebo, ArGoS 15

3. Virtual Robot Experimentation Platform (V-REP) **21**
3.1. Principales características de V-REP 21
3.1.1. Técnicas de control de la simulación 21
3.1.2. Seis enfoques de programación 22
3.1.3. Elementos de escena en V-REP 25
3.1.4. Modelado dinámico 28
3.2. Construcción del modelo en V-REP 29
3.2.1. Configuración de los objetos de tipo *shape* 29
3.2.2. Configuración de los objetos de tipo *joint* 30
3.2.3. Modelado de la ventosa 32
3.2.4. Modelado de la Euclid y su soporte 35
3.2.5. Workspace de la pata 38

4. API Remota **41**
4.1. Motivos de utilización 41
4.2. Modo de funcionamiento 41
4.3. Modelo cinemático 45
4.3.1. Cinemática directa 46
4.3.2. Cinemática inversa 47
4.4. Matrices de transformación homogéneas 50
4.5. Descripción de la API de ROMERIN 52

5. Resultados **55**
5.1. Resultados de la cinemática 55
5.1.1. Análisis sobre la pata 2 56
5.1.2. Análisis sobre la pata 4 58
5.1.3. Análisis sobre la pata 6 61
5.2. Resultados del algoritmo de movimiento 63
5.3. Resultados de las ventosas 66

6. Conclusiones **71**
6.1. Conclusión 71
6.2. Desarrollos futuros 71

A. Configuración de la API Remota **73**
A.1. Configuración del cliente 73
A.2. Configuración del servidor 74

B. Configuración de la escena de simulación **75**

C. Código software **77**
C.1. Código en Lua 77
C.2. Código en Python 79

Bibliografia **91**

Índice de figuras

1.1. Diagrama de Gantt (Primera mitad) . . . 3
1.2. Diagrama de Gantt (Segunda mitad) . . . 3

2.1. Prototipo de robot escalador . . . 6
2.2. Robot ROMA . . . 6
2.3. Robot modular MICROTUB . . . 7
2.4. Esquema de una ventosa de vacío . . . 7
2.5. Esquema de un electroimán . . . 8
2.6. Robot que utiliza pinzas para escalar . . . 8
2.7. Estructura mecánica del robot escalador bípedo . . . 9
2.8. Simulación de movimiento entre dos superficies del robot bípedo . . . 9
2.9. Robot pintor OutoBot . . . 10
2.10. Robot WallWalker . . . 10
2.11. Robot RiSE . . . 11
2.12. Robot PIKo . . . 12
2.13. Ascenso vertical por una tubería del robot PIKo . . . 12
2.14. Movimiento de un operario alrededor de un brazo robótico . . . 13
2.15. V-REP . . . 15
2.18. Escenas de simulación . . . 16
2.16. Gazebo . . . 16
2.17. ArGOS . . . 16

3.1. Los 6 métodos de control en V-REP (Fuente: [48]) . . . 22
3.2. Asociación de *child scripts* con el *main script* . . . 24
3.3. Flujo de ejecución de los diferentes mecanismos de personalización de V-REP (Fuente: [48]) . . . 24
3.4. Tipos de *joints* (Fuente: [16]) . . . 25
3.5. Tipos de objetos de escena en V-REP (Fuente: [27] . . . 27
3.6. Librería *Bullet physics* . . . 28
3.7. Motor dinámico ODE . . . 28
3.8. Librería dinámica *Vortex* . . . 28
3.9. Librería dinámica *Newton* . . . 29
3.10. Ejemplo de jerarquía entre *convex shapes* y *random shapes*. . . . 30
3.11. Ejemplo de configuración de las máscaras colisionables locales y globales 30
3.12. Jerarquía de los tres primeros grados de libertad de la pata 1 . . . 31
3.13. Jerarquía y montaje de una ventosa . . . 32
3.14. Intel Euclid Development Kit . . . 35
3.15. Soporte de la *Intel Euclid* y de la *PCB* . . . 36

3.16. Adaptación de la *Intel Euclid* en V-REP 36
3.17. Jerarquía de montaje del soporte y la *Intel Euclid* 37
3.18. Soporte con la *Intel Euclid* acoplada 37
3.19. Captura de los sensores RGB y de profundidad de la *Intel Euclid* dentro de V-REP 38
3.20. *Workspace* de la pata desde diferentes perspectivas 39
3.21. *Collection* de objetos para el cálculo del *workspace* 39

4.1. Modo de funcionamiento bloqueante (Fuente: [23]) 42
4.2. Modo de funcionamiento no bloqueante (Fuente: [23]) 43
4.3. Envío de múltiples instrucciones a la vez desde el cliente (Fuente: [23]) 44
4.4. Modo de funcionamiento *data streaming* (Fuente: [23]) 44
4.5. Modo de funcionamiento síncrono (Fuente: [23]) 45
4.6. Relación entre la cinemática directa y la cinemática inversa 46
4.7. Esquema cinemática directa 46
4.8. Esquema cinemática inversa 47
4.9. Perspectiva de la pata en el plano XY 47
4.10. Primer triángulo obtenido de la Figura 4.9 48
4.11. Segundo triángulo obtenido de la Figura 4.9 48
4.12. Perspectiva frontal de la pata en el plano ZR 49
4.13. Ejes de coordenadas de cada pata y del centro del robot 51
4.14. Distancia entre la primera articulación de cada pata y el centro del robot 52

5.1. Posición de reposo de la pata 2 desde diferentes perspectivas 56
5.2. Posición objetivo de la pata 2 desde diferentes perspectivas (Codo arriba) 56
5.3. Gráficas de los motores de la pata 2 (Codo arriba) 57
5.4. Posición objetivo de la pata 2 desde diferentes perspectivas (Codo abajo) 57
5.5. Gráficas de los motores de la pata 2 (Codo abajo) 58
5.6. Posición de reposo de la pata 4 desde diferentes perspectivas 58
5.7. Posición objetivo de la pata 4 desde diferentes perspectivas (Codo arriba) 59
5.8. Gráficas de los motores de la pata 4 (Codo arriba) 59
5.9. Posición objetivo de la pata 4 desde diferentes perspectivas (Codo abajo) 60
5.10. Gráficas de los motores de la pata 4 (Codo abajo) 60
5.11. Posición de reposo de la pata 6 desde diferentes perspectivas 61
5.12. Posición objetivo de la pata 6 desde diferentes perspectivas (Codo arriba) 61
5.13. Gráficas de los motores de la pata 6 (Codo arriba) 62
5.14. Posición objetivo de la pata 6 desde diferentes perspectivas (Codo abajo) 62
5.15. Gráficas de los motores de la pata 6 (Codo abajo) 63
5.16. Algoritmo de movimiento por el suelo de ROMERIN 64
5.17. Algoritmo de movimiento *tripod gait* 64
5.18. Gráficas de los motores de la pata 1 utilizando el algoritmo de movimiento 65

5.19. Robot ROMERIN pegado a la pared 66
5.19. Gráficas de las fuerzas de cada una de las ventosas de ROMERIN . . 67
5.19. Gráficas de las fuerzas de cada una de las ventosas de ROMERIN . . 68
5.19. Gráficas de las fuerzas de cada una de las ventosas de ROMERIN . . 69

A.1. Esquema API Remota 73

Índice de tablas

2.1. Características integradas en el simulador (Fuente: [60]) 17
2.2. Interfaz de usuario (GUI) (Fuente: [60]) 17
2.3. Modelos de robots y objetos (Fuente: [60]) 18
2.4. Métodos de programación (Fuente: [60]) 18

3.1. Posiciones límite Motor 1 31
3.2. Posiciones límite Motor 2 31
3.3. Posiciones límite Motor 3 31

5.1. Coordenadas de los puntos de prueba y de los puntos finales 55

Índice de fragmentos de código

1. Child Script de la primera ventosa de ROMERIN en V-REP 32
2. Establecer posicion de 3 joints a la vez (Fuente: [23]) 43
3. Cabecera constructor de ROMERIN_API 52
4. Cabecera moveLeg de ROMERIN_API 52
5. Cabecera getPosition de ROMERIN_API 53
6. Cabecera setGripper de ROMERIN_API 53
7. Cabecera getGripper de ROMERIN_API 53
8. Cabecera setPosition de ROMERIN_API 53
9. Child Script del Workspace de una pata de ROMERIN 77
10. Clase RomerinAPI 79
11. Clase Leg 82
12. Clase Joint 84
13. Graficar posiciones de los motores 85
14. Graficar fuerzas de las ventosas 86

Capítulo 1

Introducción

1.1. Motivación del proyecto

El constante avance tecnológico y, en especial, del mundo de la robótica, permite que poco a poco se estén diseñando robots que pueden cubrir ciertas tareas que ponen en peligro la integridad de las personas. El proyecto ROMERIN (Robot Modular Escalador de Inspección de Infraestructuras) nace en este contexto, pues su principal tarea será realizar la inspección de conductos cuyo acceso es prácticamente nulo.

Para ello, se parte de un robot comercializado en el mercado por la empresa XYZ Robot, concretamente el modelo XYZ Bolide Y-01. A partir de este se desarrollan proyectos paralelos que se encargarán de realizar las modificaciones pertinentes del robot para dotarle de las características necesarias para las tareas de inspección, como son las ventosas necesarias para adherirse a las paredes, el sistema de percepción en 3D, el diseño del firmware o la simulación del robot escalador.

A partir de la necesidad de realizar pruebas y comprobar que las implementaciones son seguras, nace el proyecto que se describe a lo largo de este documento, donde se construye un modelo en el simulador V-REP (Virtual Robot Experimentation Platform), un simulador de robots muy versátil y lleno de potentes funcionalidades que permitirá reproducir con gran exactitud el modelo real.

1.2. Objetivos

El objetivo principal de este proyecto es **diseñar e implementar un robot hexápodo en el simulador V-REP**, que permita validar las diferentes estrategias que se quieran implementar en el modelo real. Para ello, una objetivo fundamental a conseguir era modelar correctamente el **sistema de agarre** con sus correspondientes sensores y *scripts* de control. Además, también era un objetivo primario el desarrollar todo el control del robot de manera externa en **Python**, haciendo uso de la **API remota** que permite controlar una simulación de V-REP o el propio simulador desde una aplicación externa o un hardware remoto.

Entre los objetivos secundarios, se encontraba la implementación de un algoritmo de movimiento básico, la modelización de entornos reales y la incoporación al modelo de otros elementos como la cámara de visión 3D.

A continuación se resumen los principales puntos necesarios para la consecución existosa del proyecto:

- Modelado del robot sobre V-REP.
- Modelado del sistema de agarre y su sensorización.
- Desarollo de la programación del robot en Python.
- Comunicación desde la API remota con V-REP.
- Implementación de un sistema de zancada básico.
- Estado del arte de robots escaladores y simuladores de robótica existentes en la actualidad.
- Programación de estrategias y validación sobre el simulador.

1.3. Materiales utilizados

Para la realización de este proyecto, se han empleado una serie de herramientas software:

- V-REP PRO EDU, Version 3.5.0 (rev. 4)
- PyCharm CE, Version 2018.2.4
- API Remota de V-REP
- Librería *vrep* de Python
- Librería *numpy* de Python
- Librería *enum* de Python
- Librería *matplotlib.pyplot* de Python
- Modelado de la ventosa en 3D
- Modelado del soporte de la Euclid en 3D

1.4. Planificación del proyecto

La planificación del proyecto se puede observar con detalle en el **diagrama de Gantt** de la Figura 1.1 (primera mitad del proyecto) y la Figura 1.2 (segunda mitad del proyecto).

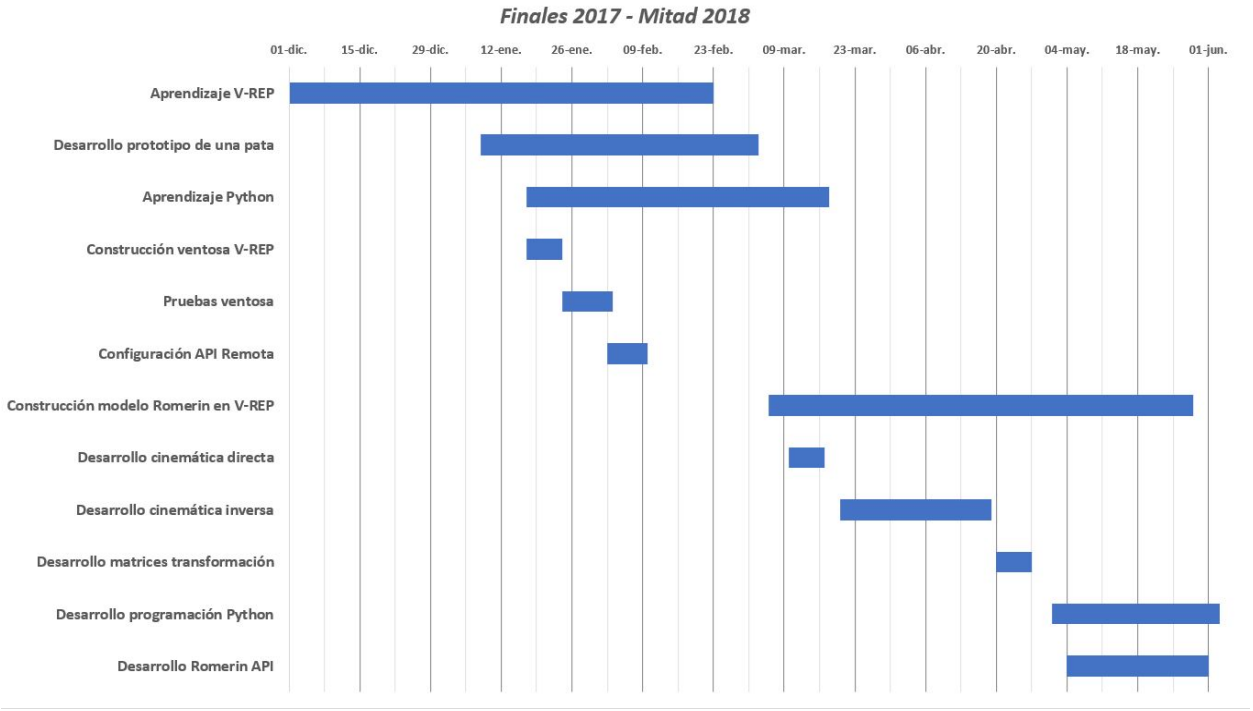


Figura 1.1: Diagrama de Gantt (Primera mitad)

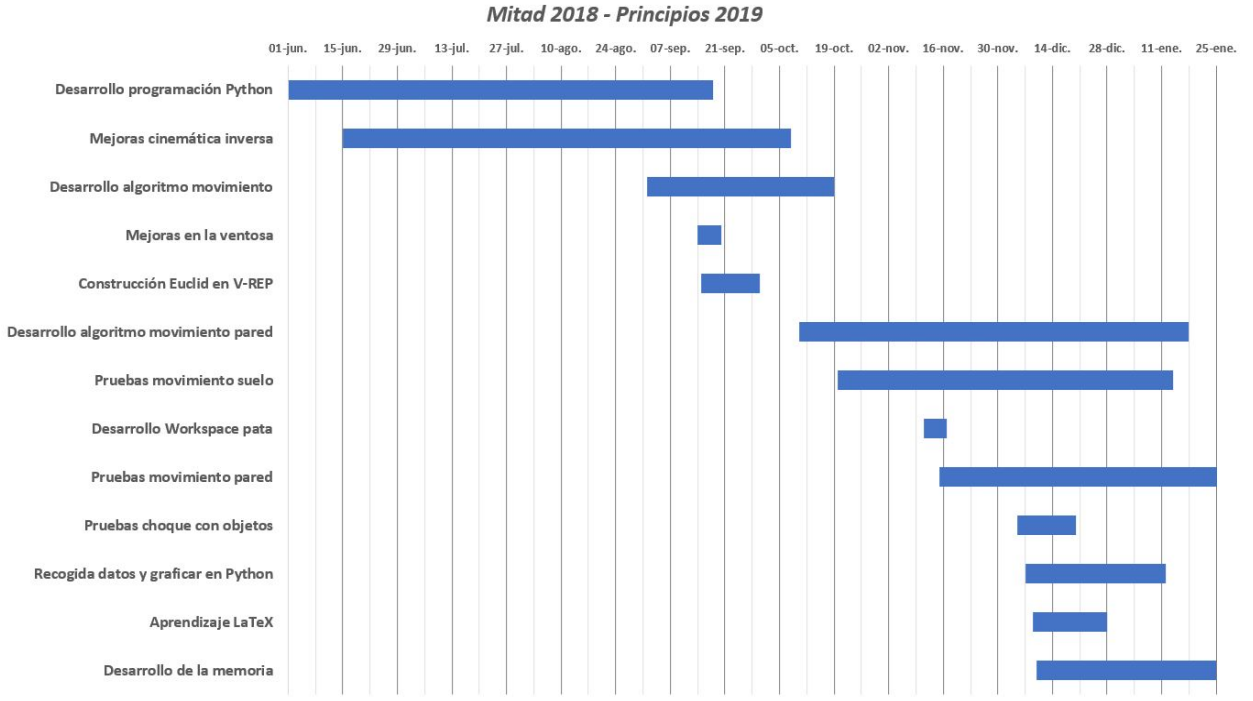


Figura 1.2: Diagrama de Gantt (Segunda mitad)

1.5. Estructura del documento

A continuación y para facilitar la lectura del documento, se detalla el contenido de cada capítulo.

- En el **capítulo 1** se realiza una breve introducción del proyecto y de los principales objetivos y materiales utilizados para llevarlo a cabo.
- En el **capítulo 2** se hace un repaso del estado del arte de los robots escaladores y de los diferentes simuladores de robots existentes en la actualidad, así como una comparación de los más adecuados para el desarrollo de este proyecto.
- En el **capítulo 3** se detallan las principales características del simulador escogido y la construcción de los diferentes elementos del robot ROMERIN dentro del simulador.
- En el **capítulo 4** se muestran los diferentes modos de funcionamiento de la API Remota, las ventajas que proporciona y la forma de implementarla en este proyecto. Además, se detallan todos los cálculos matemáticos referentes al hexápodo y una descripción de su API.
- En el **capítulo 5** se exponen las diferentes pruebas que se han llevado a cabo con el modelo simulado y los diferentes datos que se han recogido de las mismas.
- En el **capítulo 6** se muestran las conclusiones a las que se han llegado tras la consecución del proyecto y se presentan los posibles desarrollos futuros.

Capítulo 2

Estado del arte

2.1. Robots escaladores

Existen una serie de requisitos que debe cumplir un robot escalador para poder considerarse como tal. En primer lugar, debe ser capaz de desplazarse por entornos que posean un elevado porcentaje de inclinación. Además, no es suficiente con que se mueva sólo hacia delante, sino que tiene que ser capaz de desplazarse en cualquier dirección. Por último, otro requisito indispensable es que sea capaz de soportar su propio peso, pues al desplazarse por superficies verticales tiene que vencer constantemente a la gravedad sin que esta provoque que el robot se deslice o se caiga.

A lo largo de este capítulo se van a describir los diferentes tipos de robots escaladores en función de la superficie por la que se desplazan, así como el tipo de mecanismos de sujeción que se utilizan o los campos de aplicación donde tiene sentido la utilización de estos robots.

2.1.1. Clasificación según la superficie de desplazamiento

Los robots escaladores pueden estar diseñados para desplazarse por diferentes tipos de superficies.

- **Robots que se desplazan por superficies planas**
  - *Robots que se mueven por una única superficie*: en esta categoría se encuentran aquellos robots que son capaces de moverse por una única superficie, por lo que se asemejan mucho a los robots móviles normales, con la salvedad de que pueden moverse por superficies con un elevado grado de inclinación. Sin embargo, no están preparados para realizar la transición de un plano a otro que posea una inclinación muy diferente. Por ejemplo, un robot de este tipo es el prototipo que se puede observar en la Figura 2.1, el cual es un robot capaz de desplazarse por diferentes superficies, sin embargo, no es capaz de realizar el cambio de una a otra de manera autónoma. Este robot [58] está compuesto por cuatro ruedas y por un sistema de adhesión colocado en el centro del robot, el cual se encarga de realizar una presión de succión contra la superficie por la cual se va a desplazar.
  - *Robots que pueden cambiar de superficie*: estos robots, a diferencia de los anteriores, tienen una cinemática que les permite realizar el cambio de una superficie a otra que tenga un elevado grado de inclinación diferente al

Figura 2.1: Prototipo de robot escalador

Figura 2.2: Robot ROMA

plano por el que se está desplazando el robot por ese momento, por lo que pueden pasar de moverse por el suelo a hacerlo por la pared de manera autónoma. Un claro ejemplo de este robot es el robot bípedo de la Figura 2.8.

- **Robots que se mueven por vigas o columnas**: estos robots suelen estar provistos de pinzas en sus extremos, de manera que sólo pueden desplazarse por superficies con una anchura no muy grande. El robot ROMA [50] (Figura 2.2), diseñado en la Universidad Carlos III de Madrid, se ha diseñado con el propósito de realizar tareas de inspección en entornos peligrosos, como son vigas metálicas de una escasa anchura.

- **Robots que se desplazan por conductos**: este tipo de robots deben tener unas dimensiones bastante reducidas para poder moverse por el interior de tuberías de pequeño diámetro. Por ello, una de las soluciones más utilizadas a la hora de diseñar un robot escalador que tenga estos propósitos es optar por un diseño en forma de serpiente, donde se acoplan varios módulos que permiten adaptarse y moverse a través de las paredes de la tubería.

  Un claro ejemplo de este tipo de robots es el robot modular MICROTUB [51]

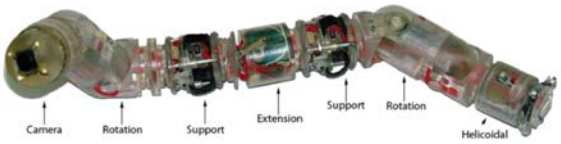


Figura 2.3: Robot modular MICROTUB

desarrollado por investigadores de la Universidad Politécnica de Madrid. Este robot (Figura 2.3), diseñado para realizar inspecciones en tuberías de un diámetro muy reducido, posee hasta seis tipos diferentes de módulos, gracias a los cuales es capaz de alcanzar velocidades de hasta 12 mm/s a lo largo de una tubería recta de 30 cm con una inclinación de 90 grados.

2.1.2. Mecanismos de sujeción

A la hora de escoger un mecanismo de sujeción, es necesario tener en cuenta el tipo de superficie por la que se va a desplazar el robot. Entre los mecanismos más utilizados se encuentran:

- **Ventosa de vacío**: este mecanismo, a través de un eyector de vacío, consigue aspirar el aire que existe entre la ventosa y la superficie con la que se encuentra en contacto, formando así una depresión (vacío). Este dispositivo resulta de gran eficiencia cuando las superficies de contacto son **poco porosas** (como el cristal o el acero), puesto que en las superficies porosas el nivel de vacío puede llegar como mucho a un 60 % [49].

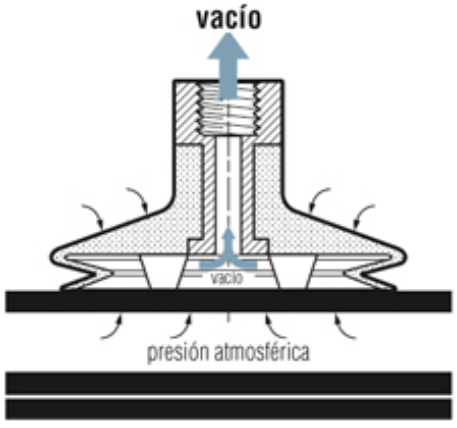


Figura 2.4: Esquema de una ventosa de vacío

- **Electroimán**: un electroimán está formado por un imán que contiene un núcleo de hierro puro enrollado por una bobina por la cual circula la corriente eléctrica. De esta manera, cuando se hace circular la corriente eléctrica por la bobina, se consigue que las moléculas del núcleo se reordenen y se alineen, formando un campo magnético que atrae a otros imanes y objetos metálicos. Cuando deja de pasar corriente por la bobina, desaparece el campo magnético y por tanto la fuerza de atracción desaparece.

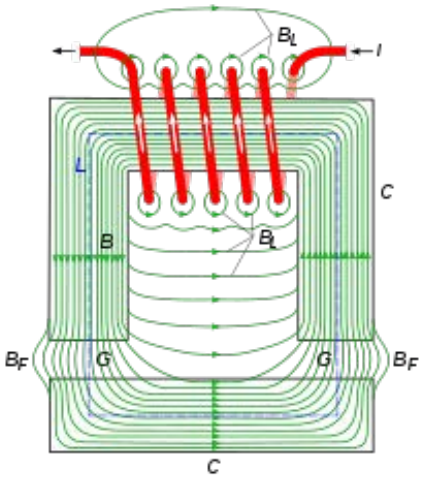


Figura 2.5: Esquema de un electroimán

Además del inconveniente de que este mecanismo sólo se puede utilizar en caso de desplazamientos por superficies metálicas, hay que tener en cuenta que el peso de estos dispositivos es bastante elevado.

- **Pinzas**: este mecanismo permite sujetarse con gran fiabilidad a vigas y columnas. Sin embargo, no es el adecuado cuando el desplazamiento se realiza por superficies planas. Para evitar que las pinzas se resbalen y mejorar el agarre, se suelen utilizar adhesivos que ejerzan una pequeña fuerza de adhesión a la superficie.

Figura 2.6: Robot que utiliza pinzas para escalar

Para controlar correctamente la apertura y el cierre de las pinzas, se suelen utilizar accionamientos eléctricos o cilindros neumáticos, siendo preferible utilizar los primeros pues permiten tener un control más preciso y pueden actuar en la fuerza que se ejerce contra la superficie.

2.1.3. Campos de aplicación

Con la creciente implantación de soluciones automatizadas dentro de la industria, no resulta extraño encontrarse cada vez más con este tipo de robots en diferentes ámbitos.

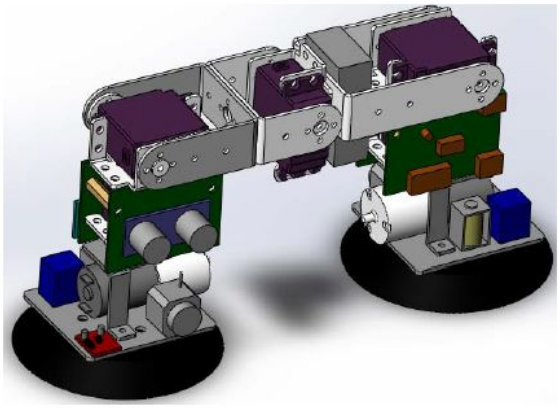


Figura 2.7: Estructura mecánica del robot escalador bípedo

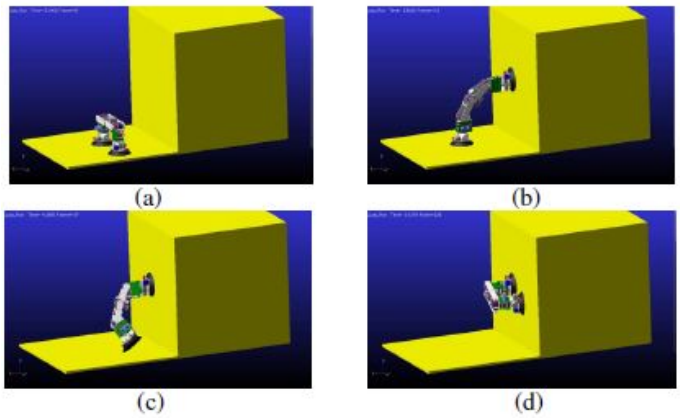


Figura 2.8: Simulación de movimiento entre dos superficies del robot bípedo

- Una de las industrias que más uso hace de los robots escaladores es la **nuclear**, puesto que la mayoría de las tareas de inspección se realizan de forma remota, por lo que resulta de gran utilidad usar robots que sean capaces de escalar paredes en una atmósfera contaminada.

  Un claro ejemplo es el robot escalador desarrollado por el programa de investigación del Instituto Tecnológico de Shenzhen [54], donde se ha desarrollado un robot bípedo con cinco grados de libertad, el cual puede escalar gracias a las dos ventosas que posee en sus extremidades, como se puede apreciar en su estructura mecánica en la Figura 2.7.

  Gracias al sistema de absorción que incopora en sus patas, este robot puede desplazarse por diferentes superficies de manera fiable. Además, su diseño permite rotar completamente todo el cuerpo del robot manteniendo una ventosa pegada a la superficie sobre la que se está desplazando, dotándole de la capacidad de cambiar entre dos superficies con inclinaciones diferentes, como se puede observar en una simulación en la Figura 2.8.

- La industria de la **construcción** es otra industria que está implementando cada vez más robots escaladores con propósitos de inspección, con el objetivo de que estos robots se muevan a lo largo de las fachadas de los edificios y puedan encontrar imperfecciones. Además, también se pueden utilizar para pintar las

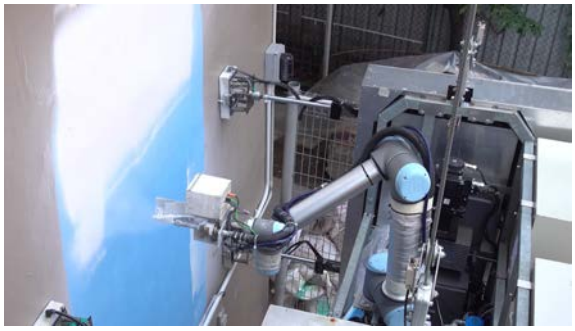


Figura 2.9: Robot pintor OutoBot

Figura 2.10: Robot WallWalker

fachadas, como es el caso del robot *OutoBot* (Figura 2.9), diseñado con una plataforma que le permite un desplazamiento vertical y con un brazo robótico que posee una cámara y un pulverizador, que le permiten pintar estructuras o expulsar chorros de agua a alta presión para limpiar la superficie deseada.

Otro uso muy común tanto en la industria de la construcción como en la industria del hogar es utilizar robots limpia-cristales, como es el caso del *WallWalker* [56]. Este robot (Figura 2.10) se desplaza por las cristaleras verticales gracias a dos ruedas y una bomba que realiza el vacío entre la superficie y el cuerpo del robot.

- Otro campo de aplicación de estos robots es en la industria **civil** donde, de forma parecida a la industria de la construcción, se utilizan robots escaladores para realizar inspecciones periódicamente de estructuras como muros o puentes.

- Dentro de la industria **naval**, se utilizan los robots escaladores para realizar tareas de soldadura o de pintura de los cascos de los barcos. Debido a que la superficie por la que se mueven es metálica, los robots más adecuados para estas tareas son aquellos que incorporan un mecanismo de sujeción mediante electroimanes.

- Dentro de la industria **minera**, los robots escaladores pueden resultar útiles para realizar tareas de inspección cuando se ha producido un escape de gas o una explosión en el interior de la mina, evitando que un ser humano tenga que realizar esta tarea exponiéndose a un ambiente altamente contaminado.

Figura 2.11: Robot RiSE

- La industria **química** también es un campo de aplicación de este tipo de robots, puesto que pueden realizar tareas de inspección y reparación dentro de tanques que contengan restos de residuos químicos.

- Otra aplicación interesante es la de utilizar los robots escaladores para **rescates de emergencia**, puesto que podrían entrar en una estructura que ha sufrido un incendio o un terremoto con el propósito de encontrar supervivientes. Un ejemplo de este tipo de robots es el modelo **RiSE** [62] (Robots in Scansorial Environments) (Figura 2.11) , un robot desarrollado por la empresa *Boston Dynamics*. Este robot hexápodo emplea motores de corriente continua *brushless* que aumentan la densidad de potencia, haciendo posible su labor de escalada.

  Para el desarrollo de RiSE, sus creadores se han inspirado en algunas habilidades que presentan los animales escaladores que se pueden encontrar en la naturaleza. Algunas de estas características son:

  - Los animales a la hora de escalar aplastan el cuerpo, de manera que el centro de masa se sitúa muy cercano a la superficie vertical por la que se están desplazando, reduciendo así el momento *pitch* [29].

Figura 2.12: Robot PIKo

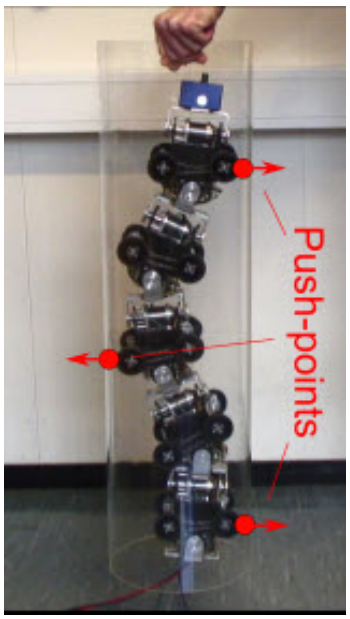


Figura 2.13: Ascenso vertical por una tubería del robot PIKo

- Un cuerpo largo y una cola permiten reducir la fuerza de tracción ejercida en las extremidades frontales.
- Las patas, tobillos y dedos permiten distribuir las fuerzas de contacto.

Aunque por el momento las aplicaciones de salvamento de este robot son muy limitadas y de importancia menor, es posible que en un futuro pueda colaborar con las fuerzas de salvamento para tareas de mayor complejidad.

- Para la inspección de **tuberías** y **conductos**, estos robots también son muy apropiados, ya que pueden desplazarse por conductos muy estrechos de calefacción o aire acondicionado y realizar una limpieza de los mismos.

  El robot con forma de serpiente **PIKo** [53] (Pipe Inspecting Konda) (Figura 2.12) permite realizar inspecciones por tuberías tanto horizontales como verticales. Además, gracias a su construcción, puede adaptarse a tuberías de diferentes diámetros, pues su funcionamiento es similar al de un tren, donde cada módulo sigue al anterior. A la hora de realizar un ascenso vertical, PIKo utiliza su cuerpo para ejercer fuerzas de empuje contra las paredes de la tubería, como se puede observar en la Figura 2.13.

2.2. Simuladores de robots

La **simulación de un robot** comprende el desarrollo del diseño y de la programación de un modelo virtual que sea capaz de emular el comportamiento real de ese

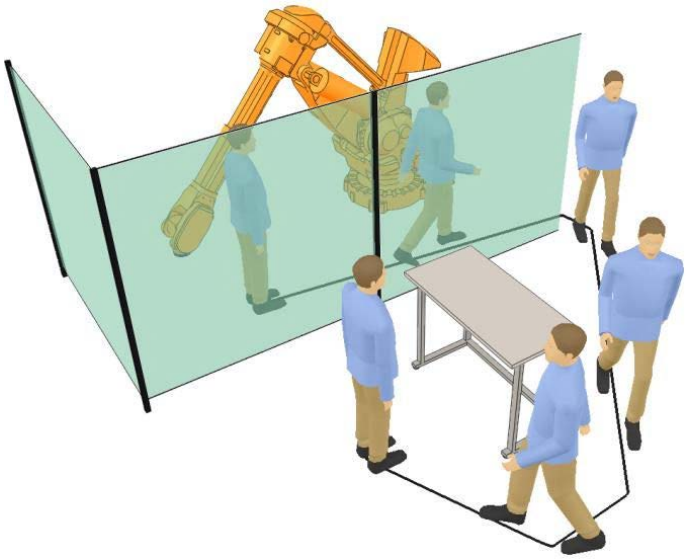


Figura 2.14: Movimiento de un operario alrededor de un brazo robótico

robot.

La simulación es el primer paso en un proyecto que tenga como objetivo la creación de un robot, y cobra un papel fundamental puesto que permite desarrollar y probar que funcionan correctamente todos los componentes y todos los algoritmos antes de proceder con el desarrollo del robot real.

Aunque la simulación proporciona una gran cantidad de ventajas, como poder simular los comportamientos del robot sin ningún tipo de coste o probar cualquier elemento del robot por separado, también tiene la desventaja de que el modelo real puede encontrarse con situaciones que no se puedan recrear en el simulador, por lo que hay que tener en cuenta esta limitación a la hora de diseñar su estructura y su control.

2.2.1. Simuladores disponibles en el mercado

Hoy en día existe multitud de software de simulación de robots que permite crear un modelo en 3D y recrear un entorno muy parecido a la realidad, como el que representa la Figura 2.14, donde se utiliza esa simulación para validar que el brazo robótico cumpla la medidas de seguridad cuando un operario entra dentro de su zona de trabajo.

A continuación se describen brevemente la mayoría de los simuladores de robots que se encuentran disponibles en la actualidad [19]:

- **Microsoft Robotics Developer Studio** [22]: se trata de un simulador en 3D basado en el entorno de programación *.NET* [1] de licencia gratuita desarrollado por Microsoft, el cual es compatible con todas las versiones de Windows.

- **RoboLogix** [34]: se trata de un software de simulación de robots industriales de cinco ejes desarrollado por la empresa *Logic Design Inc.* No dispone de licencias gratuitas, aunque sí que ofrece descuentos educacionales.

[1] Plataforma de desarrollo de Microsoft para crear aplicaciones de escritorio o móviles

- **Anykode** [2]: este simulador está principalmente enfocado a robots destinados a desempeñar tareas del hogar. Es compatible con Windows y Linux, y provee licencias educacionales gratuitas.
- **Webots** [46]: desarrollado por la empresa *Cyberbotics*, este simulador multiplataforma (compatible con Windows, Linux y Mac OS) es uno de los simuladores de robótica más utilizados para propósitos educativos o de investigación.
- **MotoSim** [24]: se trata de un simulador dedicado para los robots de la empresa *Motoman*. Es una herramienta de pago, aunque es posible utilizar una prueba gratuita durante unos días.
- **RobotExpert** [35]: es un simulador 3D diseñado por *Siemens* con el propósito de simular de una manera muy precisa entornos industriales donde existan robots de esta empresa. Dispone de una prueba de 30 días, tras la cual es necesario adquirir una licencia si se desea continuar utilizando el simulador.
- **RobotStudio** [36]: esta herramienta está desarrollada por la conocida empresa de robots *ABB* y está diseñada para simular el comportamiento de los robots de esta empresa. Dispone de una licencia básica gratuita y una modalidad *premium* de pago.
- **WorkSpace** [47]: se trata de un simulador 3D desarrollado por *WAT Solutions* y es compatible con multitud de robots de fabricantes como *ABB, Fanuc* o *Mitsubishi*.
- **RoboWorks** [37]: este simulador 3D, desarrollado por *Newtonian*, permite simular robots industriales y de servicios, ofreciendo soporte para C/C++, LabView, VB o VB.NET, entre otros.
- **Blender** [6]: se trata de una herramienta muy potente multiplataforma dedicada al modelado, renderizado y creación de gráficos 3D con licencia gratuita. Aunque se trata de un simulador que destaca por su potencia y versatilidad, no está especializado en robots, por lo que tiene importantes carencias frente a otros simuladores dedicados específicamente a la robótica.
- **Gazebo** [14]: este simulador multiplataforma es uno de los más completos que se pueden encontrar. Permite simular de manera muy precisa y eficiente complejos modelos robóticos en entornos cerrados o en campo abierto. Es compatible con *ROS*, así como con una gran variedad de sensores y *plugins*.
- **Simbad** [41]: es un simulador 3D basado en Java, de licencia gratuita, enfocado a la investigación y a la educación. No está adaptado para simular situaciones del mundo real, sino para que los estudiantes o investigadores puedan aprender las bases de la inteligencia artificial y el *Machine Learning*.
- **Lpzrobots** [20]: desarrollado por la universidad de Leipzig, este simulador programado en C++ permite simular de manera bastante realista diferentes tipos de robots 3D. Es totalmente compatible con Linux y con Mac OS.
- **V-REP** [43]: desarrollado por la empresa *Coppelia Robotics*, se trata de uno de los simuladores más potentes y versátiles que existen en la actualidad. Tiene compatibilidad con una gran variedad de lenguajes de programación, como son

C/C++, Python, Matlab, Lua o Java. Además, permite controlar cada objeto de la simulación de manera individual a través de multitud de herramientas como nodos *ROS*, *plugins* o *scripts*, entre otros.

- **AristoSim** [3]: se trata de una herramienta con una interfaz de usuario muy sencilla que permite simular y programar robots de carácter industrial. Se puede ejecutar en local o mediante un navegador web.
- **ANVEL** [4]: se trata de un simulador especializado en el modelado y la simulación de herramientas que permiten evaluar y probar vehículos inteligentes.
- **ARGoS** [5]: este simulador de licencia gratuita compatible con Linux y Mac OS está bajo la licencia del *Instituto Tecnológico de Massachusetts (MIT)* y se trata de uno de los simuladores más optimizados que existen en la actualidad. Permite añadir nuevas características y componentes a través de *plugins*, así como la posibilidad de programar en diferentes lenguajes de programación.

2.2.2. Comparativa entre los simuladores más completos: V-REP, Gazebo, ArGoS

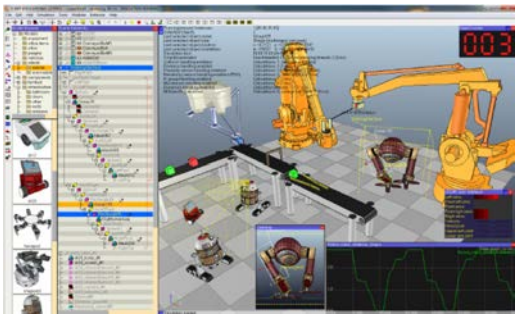
Tal y como se ha comentado en el capítulo 2.2.1, a día de hoy existe una gran variedad de simuladores adecuados para la simulación de robots. Sin embargo, para el desarrollo de este proyecto es necesario que el simulador cumpla con una serie de requisitos, como son la capacidad para realizar simulaciones dinámicas lo más realistas posibles, la facilidad de implementar nuevos componentes externos a los que se encuentran dentro del simulador, o la posibilidad de comunicarse de manera externa con el simulador, pudiendo realizar todo el control de manera externa.

Por ello, entre la gran cantidad de simuladores existentes, los que más se adecúan a la potencia y versatilidad requerida son el simulador **Virtual Robot Experimentation Platform (V-REP)**, el simulador **Gazebo** y el simulador **Autonomous Robots Go Swarming (ARGoS)**. Estos tres simuladores son posiblemente las opciones más potentes y que mayores funcionalidades presentan a la hora de realizar simulaciones robóticas.

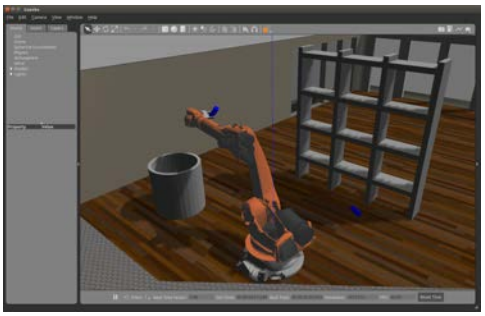
V-REP es un simulador desarrollado por la empresa *Coppelia Robots* situada en Suiza. Se trata de una herramienta *software* de uso comercial, sin embargo, posee una licencia gratuita para la educación. Es un simulador multiplataforma, por lo que es compatible con Windows, Linux y Mac OS. Su lenguaje de programación por defecto es *Lua*, aunque permite la programación en seis tipos diferentes de lenguajes, entre otras muchas características que se detallarán a lo largo de este capítulo. Es uno de los simuladores mejor puntuados por los usuarios, como se puede comprobar en [55].

Figura 2.15: V-REP

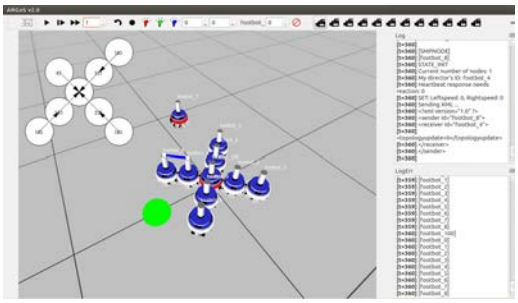
Gazebo es un simulador que nació como un proyecto de la Universidad del Sur de California. Posteriormente, pasó a formar parte de *Open Source Robotics*



(a) V-REP



(b) Gazebo



(c) ARGoS

Figura 2.18: Escenas de simulación

Foundation, una *spin-off* encargada también del desarrollo de *ROS*. Actualmente, es el simulador más conocido de los que se encuentran disponibles, tal y como refleja la encuesta realizada en [55]. Gazebo es un simulador de licencia gratuita, compatible únicamente con Linux, aunque en un futuro también será compatible con Windows.

Figura 2.16: Gazebo

ARGoS [59] es un simulador robótico desarrollado dentro del proyecto EU-funded Swarmanoid [33]. Posteriormente, ARGoS pasó a formar parte del proyecto ASCENS [32]. ARGoS es compatbile con Linux y con Mac OS. Su principal propósito es para simular enjambres de robots.

Figura 2.17: ArGOS

En la Figura 2.18 se puede observar la interfaz de cada simulador y diferentes entornos de ejemplo donde se han insertado varios robots y elementos de escena.

A continuación se va a realizar una comparativa entre las principales características de cada uno de estos simuladores, las cuales se encuentran recogidas en las Tablas 2.1, 2.2, 2.3 y 2.4.

Tabla 2.1: Características integradas en el simulador (Fuente: [60])

Características integradas en el simulador		
V-REP	**Gazebo**	**ARGoS**
Incluidas diferentes librerías dinámicas (Bullet 2.78, Bullet 2.83, ODE, Newton y Vortex).	Incluida solamente la librería física ODE por defecto, aunque se permiten incluir otras librerías.	Librerías físicas muy limitadas.
Incluido un editor de código y un editor de escenas.	Incluido un editor de código y un editor de escenas.	Incluido un editor de código pero no un editor de escenas.
Los objetos pueden ser manipulados por robots en tiempo real.	No permite la manipulación de objetos.	No permite la manipulación de objetos.
Durante la simulación, se pueden manipular fácilmente los objetos de la escena y, una vez termina la simulación, todo vuelve a su estado inicial.	Los objetos de escena pueden ser manipulados durante la simulación, y todo vuelve a su estado de inicio cuando se termina la simulación.	Los objetos se pueden mover durante la simulación.
Se incluyen multitud de complementos para observar gráficas o vídeo obtenido por los sensores durante la simulación.	Cierta variedad de complementos para observar gráficas o textos.	Complementos para vídeo e imágenes.
Incluye la posibilidad de simular sistemas de partículas, como se puede observar en la simulación de un dron.	No incluye simulación de sistemas de partículas.	No incluye simulación de sistemas de partículas.

Tabla 2.2: Interfaz de usuario (GUI) (Fuente: [60])

Interfaz de usuario		
V-REP	**Gazebo**	**ARGoS**
No experimenta problemas de funcionamiento.	La interfaz se congela en varias ocasiones y, en ocasiones, es necesario reiniciar el programa.	No experimenta problemas de funcionamiento.
Todas las funcionalidades son bastante intuitivas.	La usabilidad de la interfaz de usuario es relativamente baja.	La interfaz de usuario es bastante limitada, pero las funcionalidades son muy intuitivas.
La librería de modelos se distribuye con V-REP y está totalmente disponible.	La librería de modelos no se distribuye con Gazebo, y en ocasiones presenta problemas para cargar modelos de dicha librería debido a que Gazebo no puede conectar con su servidor de internet.	La librería de modelos de robots se distribuye con ARGoS y está totalmente disponible.
Los modelos incluidos en la librería están ordenados correctamente por carpetas y categorías, permitiendo al usuario organizar correctamente sus propios modelos.	La librería de modelos es una larga lista en la cual resulta complicado encontrar el modelo requerido.	La librería de modelos proporcionada nativamente por ARGoS permite buscar el modelo requerido mediante la línea de comandos.

Tabla 2.3: Modelos de robots y objetos (Fuente: [60])

Modelos de robots y objetos		
V-REP	**Gazebo**	**ARGoS**
Incluye una gran variedad de modelos robóticos incluidos dentro del simulador, así como actuadores y sensores.	Integra una colección menor de robots integrados. Aunque permite la integración de modelos de terceros, la documentación existente sobre ellos es bastante escasa.	Es el que menos robots integra de los tres simuladores.
Permite la importación de objetos externos, los cuales son insertados como colecciones de subcomponentes. Además, es posible manipular gran variedad de propiedades de las diferentes partes de los objetos importados.	Los objetos externos importados se importan como objetos simples.	No es posible importar figuras externas.
Es posible simplificar, dividir y combinar objetos.	Los objetos importados no se pueden modificar.	No aplica.

Tabla 2.4: Métodos de programación (Fuente: [60])

Métodos de programación		
V-REP	**Gazebo**	**ARGoS**
Las escenas se guardan con un formato especial de V-REP (extensión .ttt). Esto implica que sólo se pueden ejecutar escenas desde la interfaz de V-REP.	Cada escena es guardada como un archivo XML, por lo que se puede cambiar la escena desde una interfaz distinta.	Cada escena es guardada como un archivo XML, por lo que se puede cambiar la escena desde una interfaz distinta.
Es compatible con diferentes funcionalidades de programación, como *scripts* incluidos en los robots, *plugins*, nodos ROS o programas externos que se comunican con V-REP a través de la API Remota.	Las funcionalidades pueden ser programadas en forma de plugins en C++ o en programas en *ROS*.	Se pueden programar los *scripts* de los robots en C++ o en Lua.
Todos los *scripts* y *plugins* contenidos dentro de los modelos de robot dentro del simulador funcionan sin problemas.	Muchos de los *plugins* que se encuentran dentro de los robots no funcionan.	Los ejemplos proporcionados por la página web de ARGoS funcionan correctamente.
Existe una completa documentación y muy detallada, incluyendo multitud de tutoriales y ejemplos de código. Posee un foro bastante activo donde los usuarios pueden realizar sus consultas. Además, recibe con frecuencia actualizaciones desde 2013.	Posee una bastante extensa documentación, incluyendo tutoriales y una comunidad de soporte.	La documentación que proporciona es buena, pero carece de una buena comunidad de soporte. Además, las actualizaciones no se producen de manera regular.

Tras la comparativa entre las diferentes características de los tres simuladores, se puede apreciar fácilmente cómo **V-REP** ofrece un mayor repertorio, así como una gran cantidad de modelos de robots y una documentación muy bien detallada. Sin embargo, una de sus debilidades es la imposibilidad de guardar las escenas en un formato diferente al de V-REP, tal y como hacen Gazebo y ARGoS, que sí permiten guardar las escenas en XML. Por otro lado, aunque Gazebo ofrece también muchas características (no tantas como V-REP), presenta problemas en el funcionamiento del programa, con cierres inesperados o congelaciones de la interfaz. Por último, **ARGoS** es el que menos características presenta de los tres, sin embargo, se encuentra muy bien optimizado y su interfaz funciona de manera fluida y sin problemas.

Por otra parte, uno de los requisitos indispensables para desarrollar este proyecto es que el simulador empleado tenga compatibilidad con *ROS*, puesto que es necesario que se puedan acoplar fácilmente nuevos módulos (llamados **nodos** en *ROS*) y que estos puedan interactuar todos a la vez y de manera independiente, pues el objetivo a futuro es poder desarrollar un robot modular. Aunque desgraciadamente **ARGoS** no ofrece por el momento compatibilidad con *ROS*, **Gazebo** y **V-REP** sí que la tienen, aunque con ciertas diferencias [57].

Por un lado, **Gazebo** es el simulador utilizado por defecto en el *framework* de *ROS*. Existe un repositorio oficial que contiene *plugins* de *ROS* y Gazebo, los cuales están adheridos a una escena de simulación, proporcionando métodos de comunicación muy sencillos de *ROS* como *topics* y *services*.

En contraste, **V-REP** no dispone de un **nodo ROS** nativo. Sin embargo, sí que ofrece un *plugin* de *ROS* que puede ser utilizado dentro de los *scripts* de Lua para crear nodos *publishers* y nodos *suscribers*. Además, existe un repositorio creado por un grupo de investigación del instituto de Francia que se asemeja a las funciones que ofrece el paquete *gazebo_ros*.

Por tanto, **Gazebo** se sitúa como el simulador con mejor compatibilidad con *ROS*, ya que posee una gran comunidad y el desarrollo del simulador lo lleva a cabo la misma organización que desarrolla *ROS* (la *Open Source Robotics Foundation*), por lo que su total compatibilidad está garantizada por el momento. Sin embargo, **V-REP** no se encuentra muy lejos en cuanto a compatibilidad con *ROS*, por lo que las diferencias en este apartado no son tan grandes como para determinar la decisión de elegir un simulador u otro.

Otro factor a tener en cuenta a la hora de utilizar un simulador de robots es el rendimiento que tenga durante la simulación. Por ello, en [60] se realizaron dos comparativas entre estos tres simuladores: en la primera, se puso a prueba la simulación de un robot situado sobre un plano en 2D; en la segunda, se creó una escena con aproximadamente 41600 vértices.

Los resultados obtenidos indican que **ARGoS** es el simulador que mayor velocidad de simulación obtiene en la primera prueba (hasta 50 robots) y en la segunda (hasta 5 robots) en cuanto a rendimiento de la interfaz de usuario (**GUI**), además haciendo uso de la menor cantidad de recursos. Por otro lado, **Gazebo** obtiene mejores resultados cuando la cantidad de robots es más grande (hasta 50 robots en la segunda prueba), sin embargo, es el simulador que más memoria necesita a la hora de ejecutar la simulación. Finalmente, **V-REP** obtiene los peores resultados en ambas pruebas utilizando la librería física *ODE*, sin embargo, utilizando *Bullet 2.78* los resultados mejoran notablemente, aunque sin llegar al óptimo rendimiento

de **ARGoS**.

Además, si se analiza el uso de la **CPU**, **V-REP** es el que mejores resultados obtiene, puesto que utiliza nuevos *threads* cuando es necesario y utiliza de manera adecuada todos los núcleos de la CPU. **Gazebo**, por otro lado, sólo hace uso de un núcleo de la CPU por proceso. Por último, **ARGoS** sí es capaz de utilizar varios *threads*, sin embargo, es el usuario quién tiene que especificar el número deseado de *threads*, pues el programa no es capaz de obtenerlos de manera independiente.

Por tanto, una vez observados los puntos fuertes y los débiles de cada uno de los tres simuladores, se puede llegar a la siguiente conclusión:

- **V-REP** es el simulador que más funcionalidades proporciona, entre las que se encuentra la posibilidad de programar en diferentes lenguajes, las diferentes librerías dinámicas con las que tiene compatibilidad y la multitud de componentes que posee. Aunque se trata del simulador que más recursos exige al ordenador donde se ejecuta, esto se debe principalmente a la complejidad de sus modelos y a todo el abanico de opciones que ofrece. Por lo tanto, es la mejor opción si se necesita implementar un modelo complejo y con un alto grado de precisión.
- **ARGoS** se encuentra en el extremo opuesto a V-REP, pues es el que menos funcionalidades ofrece de los tres, sin embargo es el más optimizado en cuanto a rendimiento y recursos del sistema. Se trata de una gran opción si los modelos que incluye el simulador se adaptan a las necesidades del usuario y se busca un rendimiento óptimo.
- **Gazebo** se encuentra a medio camino entre V-REP y ARGoS, puesto que ofrece muchas más opciones que ARGoS, y tiene un mejor rendimiento que V-REP. Sin embargo, el principal problema que presenta es su mal rendimiento en cuanto a interfaz de usuario, los cuales pueden empañar la experiencia a la hora de realizar un proyecto con este simulador.

Finalmente, una vez analizadas todas las características de estos simuladores y teniendo en cuenta los requisitos necesarios para poder cumplir con los objetivos de este proyecto, se hizo la elección del **simulador V-REP** debido a que era un requisito imprescindible poder realizar una comunicación con el simulador de manera externa, tener compatibilidad con nodos *ROS* y la posibilidad de importar objetos externos, entre otros requisitos.

Capítulo 3

Virtual Robot Experimentation Platform (V-REP)

3.1. Principales características de V-REP

A lo largo de este capítulo se van a mostrar las principales características del simulador elegido para el desarrollo de este trabajo, así como las directrices que se han seguido para construir correctamente el modelo del robot hexápodo **ROMERIN**.

3.1.1. Técnicas de control de la simulación

Es de vital importancia que los simuladores sean capaces de otorgar comportamientos síncronos y asíncronos, incluir modelos externos con facilidad y proporcionar técnicas de control nativas y no nativas.

Las tres técnicas más utilizadas para el control de la simulación se exponen a continuación: [61]

- **Ejecución del código de control en otra máquina.** Puede tratarse de una máquina diferente o de un robot, conectado al simulador a través de un socket, puerto serie, etc. La principal ventaja de esta técnica reside en que el código de control puede ser nativo y ser ejecutado en el hardware original. Sin embargo, esta técnica presenta un mayor retardo en la sincronización con el bucle de simulación.

- **El código de control es ejecutado en la misma máquina, pero en otro proceso (*thread*) diferente al del bucle de simulación.** Esta técnica, generalmente implementada mediante ejecutables externos o plug-ins cargados por el simulador, tiene un menor impacto en la carga de la CPU de la máquina sobre la que esté corriendo el simulador, pero por contra, al igual que la técnica anterior, presenta una falta de sincronización con el bucle de simulación.

- **El código de control se ejecuta en la misma máquina y en el mismo *thread* que el bucle de simulación.** A pesar de que este método implica una mayor carga en la CPU, presenta la gran ventaja de una sincronización instantánea con el bucle de simulación, sin existencia de ningún tipo de retraso en la comunicación o en la ejecución.

	Embedded script	Add-on	Plugin	Remote API client	ROS node	BlueZero node
Control entity is external (i.e. can be located on a robot, different machine, etc.)	No	No	No	Yes	Yes	Yes
Difficulty to implement	Easiest	Easiest	Relatively easy	Easy	Relatively easy	Relatively easy
Supported programming language	Lua	Lua	C/C++	C/C++, Python, Java, Matlab, Octave, Lua	Any [1]	C++
Simulator functionality access (available API functions)	500+ functions, extendable	500+ functions, extendable	500+ functions	>100 functions + indirectly all embedded script functions	Indirectly all embedded script functions	Indirectly all embedded script functions
The control entity can control the simulation and simulation objects (models, robots, etc.)	Yes	Yes	Yes	Yes	Yes	Yes
The control entity can start, stop, pause and step a simulation	Start, stop, pause	Start, stop, pause	Start, stop, pause, step	Start, stop, pause, step	Start, stop, pause, step	Start, stop, pause, step
The control entity can customize the simulator	Yes	Yes	Yes	No	No	No
Code execution speed	Relativ. slow [2] (fast with JiT compiler)	Relativ. slow [2] (fast with JiT compiler)	Fast	Depends on programming language	Depends on programming language	Fast
Communication lag	None	None	None	Yes, reduced [3]	Yes, reduced	Yes, reduced
Control entity is fully contained in a scene or model, and is highly portable	Yes	No	No	No	No	No
API mechanism	Regular API	Regular API	Regular API	Remote API	ROS	BlueZero
API can be extended	Yes, with custom Lua functions	Yes, with custom Lua functions	Yes, V-REP is open source	Yes, Remote API is open source	Yes, via embedded scripts	Yes, via embedded scripts
Control entity relies on	V-REP	V-REP	V-REP	Sockets + Remote API plugin	Sockets + ROS framework	Sockets + BlueZero framework
Synchronous operation [4]	Yes, inherent. No delays	Yes, inherent. No delays	Yes, inherent. No delays	Yes. Slower due to comm. Lag	Yes. Slower due to comm. Lag	Yes. Slower due to comm. Lag
Asynchronous operation [4]	Yes, via threaded scripts	No	No (threads available, but API access forbidden)	Yes, default operation mode	Yes, default operation mode	Yes, default operation mode

[1] Depends on ROS binding
[2] The execution of API functions is however very fast. Additionally, there is an optional JiT (Just in Time) compiler option that can be activated
[3] Lag reduced via streaming and data partitioning modes
[4] *Synchronous* in the sense that each simulation pass runs synchronously with the control entity, i.e. simulation step by step

Figura 3.1: Los 6 métodos de control en V-REP (Fuente: [48])

Las más comunes implementaciones de las técnicas expuestas tienen un impacto directo en la portabilidad y adaptabilidad de los modelos de simulación. Esto quiere decir que si el código de control del modelo de simulación no se encuentra unido al mismo, se tendrá que distribuir y compilar de manera separada.

3.1.2. Seis enfoques de programación

V-REP soporta **seis tipos** diferentes de técnicas de programación [48], tal y como se puede observar en la Figura 3.1: scripts embebidos, add-on, plugins, API remota, nodos ROS [1] y nodos BlueZero [26].

- Los **script embebidos** son scripts muy *sencillos* y *flexibles*, escritos en el lenguaje de programación *Lua*, que garantizan la compatibilidad con cualquier otra instalación de V-REP. Esta es una de las características más potentes de V-REP, puesto que permite crear un **script principal** (*"main script"*) que contiene las funcionalidades generales del modelo y, a su vez, este script principal

[1] *Robot Operating System*

puede llamar a **scripts hijos** ("*child scripts*") en una secuencia determinada, acorde a la jerarquía que posean los elementos dentro del modelo. Estos *child scripts* están ligados a un objeto de la simulación, y pueden ser ejecutados como *threaded scripts* o *non-threaded*. Los *threaded child scripts* se ejecutan dentro de un *thread* y están manejados por el script principal a través de la función *sim.launchThreadedChildScripts* [2], a modo de cascada. Por tanto, si un *threaded child script* se encuentra aún en ejecución, no se iniciará una segunda vez. Para que se ejecute de nuevo una vez terminada su ejecución es necesario haber desmarcado la opción de ejecución una vez dentro de las propiedades del script. Por otro lado, los *non-threaded child scripts* contienen una colección de **funciones bloqueantes**, es decir, que cada vez que son llamados deben realizar alguna tarea y volver al script principal. Estos scripts son llamados desde el *main script* dos veces por paso de simulación desde las funciones principales de actuación y detección [8]. Gracias a las posibilidades que brindan estos **scripts**, los elementos de la escena son perfectamente adaptables y portables. En la Figura 3.2 se puede observar la jerarquía que siguen estos scripts.

- El **add-on** [1], el cual es un método que también conssite en scripts de *Lua*, es una herramienta que permite personalizar el simulador rápidamente. Pueden utilizarse automáticamente y correr en segundo plano, o pueden ser llamados como funcionnes. Los *add-on* que se ejecutan como funciones, se ejecutan una vez cuando el usuario lo elige (por ejemplo, las herramientas de importación y de exportación). Sin embargo, los *add-on scripts* son aquellos que se están ejecutando constantemente cuando el simulador está en funcionamiento y contienen pequeños fragmentos de código para evitar que el simulador se ralentice.

- EL **plugin** [30] es una librería compartida (por ejemplo, *dll*) que se carga automáticamente por el cliente principal de V-REP en su arranque. También puede ser dinámicamente cargado/descargado a través de *sim.loadModule* [3] / *sim.unloadModule*. Permite que el usuario pueda extender las funcionalidades del simulador de manera similar a los *add-on*. Tanto la **API remota** como la interfaz de **ROS** están implementados en V-REP a través de *plugins*.

- El cliente de la **API remota** permite conectar una aplicación externa al simulador de una manera muy sencilla, de manera que se pueden invocar cientos de funciones específicas y una función genérica desde una aplicación de C/C++, un script de Python, una aplicación de Java, un programa de Matlab/Octave, o un script de Lua (para el desarrollo de este proyecto se ha optado por utilizar un script de Python, tal y como se verá con detalle más adelante).Para ello, se utilizarán las funciones de la API remota, que interactúan con V-REP mediante un socket de una manera que se reducen retrasos en la comunicación (se pueden observar todos los comandos disponibles de Python para la API remota en [13]).

- El **nodo ROS** [15]: este método permite conectar una aplicación externa a V-REP vía ROS. Hay diferentes interfaces de ROS disponibles: *The RosInterface* [39], *The ROS Plugin Skeleton* [38], y las interfaces de ROS desarrolladas por externos. Todas estas interfaces pueden funcionar normalmente en conjunto,

[2] http://www.coppeliarobotics.com/helpFiles/en/regularApi/simLaunchThreadedChildScripts.htm
[3] http://www.coppeliarobotics.com/helpFiles/en/regularApi/simLoadModule.htm

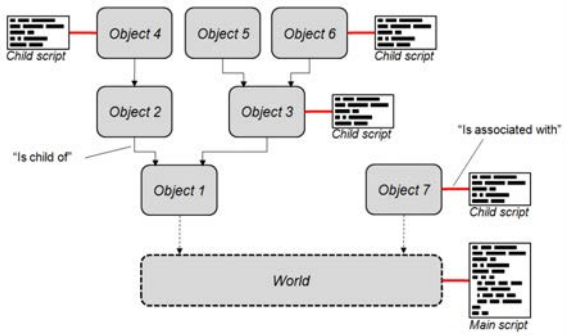


Figura 3.2: Asociación de *child scripts* con el *main script*

pero es recomendable empezar probando con la *RosInterface* debido a que es la más natural y la más flexible.

- EL **nodo BlueZero**, que permite conectar una aplicación externa con V-REP vía *BlueZero*. *BlueZero* es una aplicación multiplataforma que permite interconectar piezas de software corriendo en múltiples hilos, procesos o máquinas.

En la Figura 3.3 se pueden observar los diferentes mecanismos que se acaban de describir. A continuación se detalla el flujo de comunicación entre cada uno de ellos:

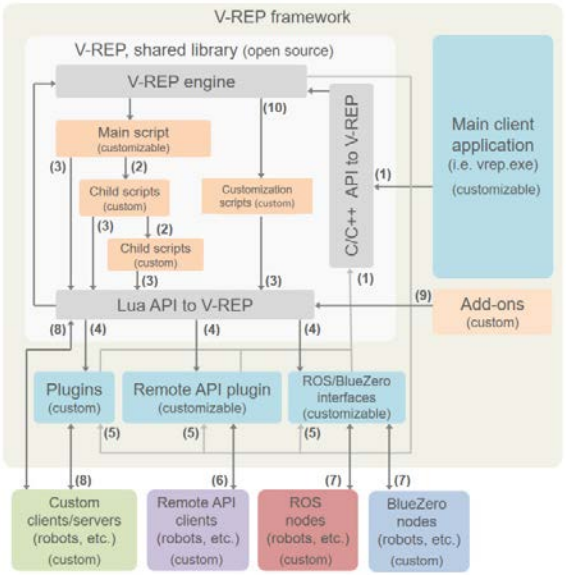


Figura 3.3: Flujo de ejecución de los diferentes mecanismos de personalización de V-REP (Fuente: [48])

1. El cliente principal de la aplicación llama a la API de C/C++ o un *plugin* de la API regular. Puede originarse desde una aplicación que no sea C/C++ si el lenguaje proporciona un mecanismo para llamar a las funciones de C.

2. Ejecución en cascada de los *child scripts*, llamados desde el *main script*.

3. La API de *Lua* es llamada desde el *main script*, *child script* o un script personalizado de la API regular.

4. Desde el simulador se llama a un *plugin*, a la API remota o a un nodo *ROS* o *BlueZero*.
5. Desde el motor de V-REP se realizan las llamadas a los *plugins*.
6. La API remota llama a aplicaciones externas, robots, etc.
7. Los nodos *ROS* o *BlueZero* realizan el intercambio de datos entre V-REP y aplicaciones externas, robots, etc.
8. El *plugin* establece comunicación con sockets, puertos series, etc.
9. La API de *Lua* llama a la API regular a través de un *add-on*.
10. El motor de V-REP llama, a través de llamadas de ejecución, a los scripts personalizados.

3.1.3. Elementos de escena en V-REP

Existen múltiples tipos de objetos dentro de una escena de V-REP [61]. Aunque más adelante se verán con más profundidad muchos de los utilizados a lo largo del desarrollo de este proyecto, a continuación se van a describir brevemente cada uno de ellos:

- **Joints** [16]: son elementos que sirven para unir dos o más objetos entre sí, proporcionando desde uno hasta tres grados de libertad (en función de si es un joint prismático, de revolución, de rosca o esférica, como se pueden apreciar en la Figura 3.4. Pueden operar en diferentes modos, por lo que el usuario deberá establecer el modo que mejor se adapte a sus necesidades de simulación.

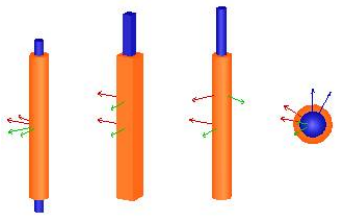


Figura 3.4: Tipos de *joints* (Fuente: [16])

  - *Force/torque mode*: en este modo, la articulación es simulada de manera **dinámica**. De esta manera, el *joint* puede ser controlado en fuerza, torque, velocidad o posición. Los *joints* esféricos sólo pueden ser articulaciones libres si operan en este modo. En el caso de que el motor esté desactivado, entonces la articulación será libre y solo estará limitada por sus límites.
  - *Passive mode*: en este modo, la articulación no puede estar activamente controlada. Sirve para representar enlaces entre elementos, aunque mediante la función *sim.setJointPosition* [4] es posible controlar la posición.
  - *Inverse kinematics mode*: la articulación de este tipo se comporta de la misma manera que en el anterior modo, con la diferencia de que se ajusta a través de los cálculos de la cinemática inversa.

[4] `http://www.coppeliarobotics.com/helpFiles/en/regularApi/simSetJoint`

  - *Dependent mode*: en este modo, la posición de la articulación es dependiente de la posición de otra articulación a través de una ecuación lineal.
  - *Motion mode*: a pesar de que el simulador permite aún establecer un *joint* con esta configuración, es un modo que se encuentra obsoleto y no se recomienda su utilización, ya que se puede obtener un mejor comportamiento configurando la articulación en modo pasivo y un *child script* asociado que se encargue de actualizar correctamente la articulación.

- **Shapes**: son redes triangulares que se utilizan para dar forma a los objetos físicos que se encuentran en la escena de simulación. Existen tres tipos de formas, que a su vez pueden agruparse entre si (siempre que sean del mismo tipo, sino al agruparse tomarán la categoría de la forma menos pura). En V-REP existen diferentes tipos de *shapes* [40]:

  - *Simple random shape*: puede representar cualquier tipo de forma. Poseen color y una serie de atributos visuales. No están recomendadas para la simulación de colisiones dinámicas, ya que tienen una respuesta muy lenta e inestable.
  - *Compound random shape*: están formadas por un conjunto de las anteriores. Tienen las mismas características que una simple.
  - *Simple convex shape*: representa una forma convexa con un color y una serie de atributos visuales. Están optimizadas para las respuestas dinámicas, aunque son menos precisas que las *pure shapes*.
  - *Compound convex shapes*: están compuestas por un conjunto de *simple convex shapes*.
  - *Pure simple shape*: representan formas básicas, como son cubos, cilindros o esferas. Son la mejor opción para la simulación de respuestas dinámicas, por lo que siempre que sea posible se elegirán este tipo de objetos para la representación del modelo.
  - *Pure compound shape*: formada por un conjunto de las anteriores mencionadas.
  - *Heightfield shape*: pueden representar diferentes tipos de terrenos, como por ejemplo una cuadrícula regular con variaciones de altura. Al igual que las *pure shapes*, están optimizadas para las respuestas dinámicas.

- **Sensores de proximidad**: permiten detectar objetos con gran precisión dentro de su volumen de detección. V-REP soporta varios tipos de sensores de proximidad (tipo *pyramid, cylinder, disk, cone, ray*).

- **Sensores de visión**: son sensores de tipo cámara, que permiten extraer información compleja de una escena de simulación (por ejemplo, colores, mapas de profundidad, tamaño de objetos, etc.). Utilizan *OpenGL* [5] para obtener las imágenes en formato *raw*.

- **Sensores de fuerza**: permiten representar uniones rígidas entre diferentes *shapes* y recoger las fuerzas y los torques que se aplican a estos objetos. También permiten establecer una fuerza de ruptura para que, en caso de someterse a una fuerza mayor, el enlace se rompa.

[5] API multiplataforma y multilenguaje utilizada para desarrollar aplicaciones que posean gráficos 2D y 3D

- **Gráficos**: permiten recoger una gran variedad de puntos y representarlos en una gráfica en tiempo real, la cual se muestra durante la simulación.
- **Cámaras**: permiten la visualización de la escena desde cualquier perspectiva.
- **Luces**: sirven para iluminar la escena o para iluminar objetos individuales. Influyen en el comportamiento de las cámaras y de los sensores de visión.
- **Paths**: permiten establecer una trayectoria predefinida en el espacio la cual debe recorrer el robot. Otro uso muy común de estos elementos es para simular los movimientos de las cintas transportadoras.
- **Dummies**: un *dummy* es un punto con orientación que se combina con otros objetos para establecer diferentes ejes de referencia a los que poseen los propios objetos.
- **Mills**: son volúmenes convexos que se puede utilizar para simular operaciones de corte en objetos (shapes), como por ejemplo simular una fresadora, una cortadora láser, etc.
- **Nube de puntos**: estructura que contiene puntos en el espacio.
- **Octrees**: son estructuras de particiones de datos en el espacio formadas por píxeles volumétricos (*voxels* [45]).
- **Espejos**: pueden reflectar imágenes o luz, aunque también pueden funcionar como un plano de recorte auxiliar.

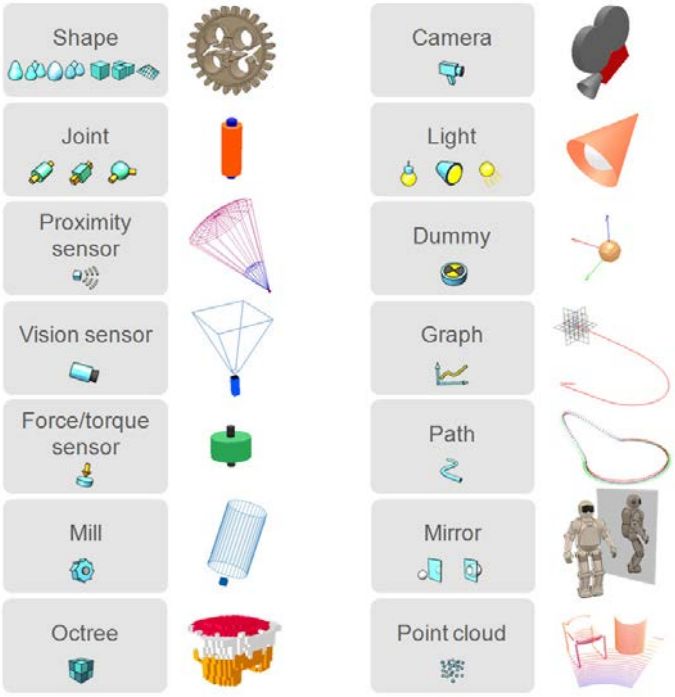


Figura 3.5: Tipos de objetos de escena en V-REP (Fuente: [27]

3.1.4. Modelado dinámico

Otra de las grandes características de este simulador es su compatibilidad con diferentes librerías de dinámica, por lo que se permite al usuario cambiar, en función de las necesidades que tenga para su simulación, entre cualquiera de estas librerías: [17]

- *Bullet physics library* [7]: se trata de un motor de física de código abierto que permite gestionar las colisiones entre objetos 3D. Es ampliamente utilizada en videojuegos o en efectos de películas.

Figura 3.6: Librería *Bullet physics*

- *Open Dynamics Engine* [28]: es otro motor de física de libre licencia que tiene dos características fundamentales: la dinámica de objetos rígidos y la detección de colisiones. También se utiliza en multitud de aplicaciones y videojuegos.

Figura 3.7: Motor dinámico ODE

- *Vortex Dynamics* [44]: es una librería comercial que ofrece parámetros del mundo real para una gran cantidad de propiedades físicas, por lo que se trata de una librería muy realista y fiable. Se utiliza principalmente en aplicaciones de precisión industrial y en investigación. Para poder utilizar esta librería con V-REP, es necesario registrarse en la plataforma *CM Labs* [6], y obtener una clave de licencia gratuita.

Figura 3.8: Librería dinámica *Vortex*

- *Newton Dynamics* [25]: se trata de una librería multiplataforma que integra un algoritmo determinista que no está basado en el tradicional LCP (*Linear Complementarity Problem* [18]) o en métodos iterativos, pero que posee la estabilidad y la velocidad de ambos. Aunque se puede utilizar también para simulaciones físicas en entornos reales, su principal aplicación se encuentra dentro del mundo de los videojuegos.

Utilizando cualquiera de las cuatro librerías mencionadas se pueden simular interacciones entre los diferentes objetos de una escena de forma bastante similar a la realidad, recreando caídas, colisiones, etc. Sin embargo, V-REP es un simulador que combina la cinemática y la dinámica con el objetivo de obtener el mejor resultado para la escena de simulación. Actualmente, los motores de física aún se basan

[6] `https://www.cm-labs.com/blog/2017/03/23/get-copy-vortex-studio-essentials/`

Figura 3.9: Librería dinámica *Newton*

en muchas aproximaciones, por lo que son relativamente lentos e imprecisos. Por lo tanto, es recomendable utilizar la cinemática siempre que se pueda y reservar la dinámica para aquellos donde una implementación de la cinemática no es viable.

Aunque a lo largo de este documento se han expuesto algunas de las principales características del simulador, existen otras muchas que aportan un gran valor añadido y justifican la gran potencia y versatilidad que posee V-REP, motivo por el cual se diferencia de muchos de sus competidores. En [31] el lector puede encontrar el resto de las principales características.

3.2. Construcción del modelo en V-REP

3.2.1. Configuración de los objetos de tipo *shape*

Para realizar el montaje del modelo en V-REP, es necesario seguir una serie de pautas, las cuales vienen explicadas con detalle en [42].

Tal y como se ha podido observar en la sección 3.1.3, existen diferentes tipos de *shapes*, por lo que es muy importante tener claras las características de cada una y la manera de ordenarlas en la **jerarquía** del modelo para que no se produzcan problemas en el funcionamiento del robot.

Todas estas *shapes* pueden ser objetos **colisionables, medibles, detectables, renderizables** y **cortables**. Esto quiere decir que:

- Pueden ser utilizadas para detectar colisiones entre otros objetos que tengan la propiedad de colisionables.
- Pueden ser utilizados para calcular distancias mínimas con otros objetos medibles.
- Pueden ser detectados por sensores de proximidad.
- Pueden ser detectados por sensores de visión.
- Pueden ser cortados por *mills* (objetos que simulan una cortadora láser o una fresadora). Sin embargo, sólo las *shapes* que no son puras pueden tener esta propiedad.

El principal concepto que hay que tener en cuenta es que hay que diferenciar las **shapes visibles** de las **shapes dinámicas**. Las *shapes* visibles son aquellas que representarán la forma visible de los elementos que tiene el modelo. Por otro lado, las *shapes* dinámicas serán aquellas que tengans asociados los cálculos dinámicos. Para este proyecto, se han insertado las piezas del modelo a través de objetos de extensión *.stl* [7]. Al insertar en V-REP archivos externos, estos se insertan como *random shapes*. Por lo tanto, no podemos realizar cálculos dinámicos para estos objetos. Para solventar el problema, lo ideal sería insertar un objeto o conjunto de objetos de tipo *pure shape* que tengan una forma parecida a la *random shape* (tal y

[7] Formato de un archivo de diseño asistido por computadora (CAD)

Figura 3.10: Ejemplo de jerarquía entre *convex shapes* y *random shapes*.

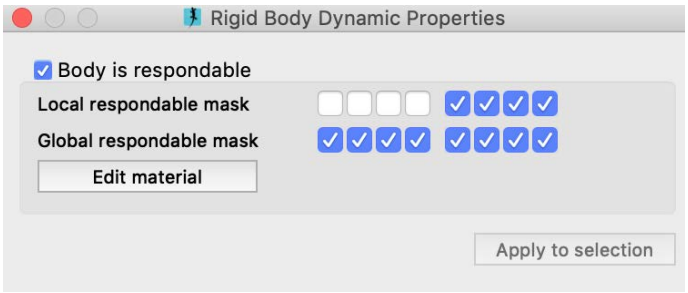


Figura 3.11: Ejemplo de configuración de las máscaras colisionables locales y globales

como se ha hecho para la base del robot de este proyecto). Sin embargo, en caso de no ser posible modelarlo a través de un objeto puro, habría que crear un objeto de tipo *convex shape* a partir del objeto *random shape* insertado, mediante la opción: *"Convex hull selection"*. A este nuevo objeto creado, al igual que a los objetos puros, también se le pueden asignar los parámetros oportunos para modelar correctamente las respuestas dinámicas. Aun así, para que la respuesta sea lo más rápida y precisa posible, es recomendable editar la *convex shape* creada y simplificarla lo máximo posible (por ejemplo, con el editor de triángulos incorporado dentro del simulador). Una vez creados todos los elementos necesarios, tanto los visuales como su objeto equivalente dinámico, se asociarán las *random shapes* a su *convex shape* correspondiente (como se puede observar en la Figura 3.10), y se ocultará la *convex shape*. De esta manera quedarán visibles las *random shapes*, pero serán las *convex shapes* ocultas las que realmente responderán de forma dinámica.

Por otro lado, es necesario configurar correctamente las **máscaras colisionables** de cada objeto. Para ello hay que tener en cuenta que para evitar colisiones entre elementos consecutivos, es necesario que en el primer elemento las primeras cuatro casillas de la máscara local se encuentren activadas y las otras cuatro estén desactivadas. Mientras que para el segundo elemento deben estar desactivadas las cuatro primeras y las otras cuatro tienen que estar activadas, y así consecutivamente (dentro del objeto, entrar en el menú *"Show dynamic properties dialog"* y configurar los parámetros como se observan en la Figura 3.11).

3.2.2. Configuración de los objetos de tipo *joint*

Dentro de V-REP, el objeto que representa las articulaciones que tiene el modelo es el *joint*. Puesto que nuestro robot hexápodo tendrá 6 articulaciones por cada pata, en total el modelo estará compuesto por 36 *joints*, todos de revolución. Los tres primeros *joints* de cada pata serán los encargados de posicionar correctamente la pata en el espacio. Por lo tanto, se corresponden con los ángulos q_1, q_2 y q_3, que se obtienen mediante la cinemática inversa (Sección 4.3.2). Se han establecido con el modo *torque/force*, puesto que es necesario poder controlarlos desde la API remota, tal y como se verá en el Capítulo 4 con mayor profundidad. Además, para tener una

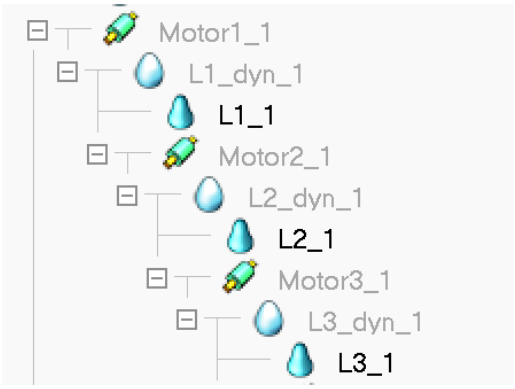


Figura 3.12: Jerarquía de los tres primeros grados de libertad de la pata 1

mejor respuesta, se ha incluido un **regulador PI** en cada uno de ellos, que permite reducir el error a la hora de alcanzar la posición deseada.

Por tanto, la jerarquía de los tres primeros grados de libertad se puede observar en la Figura 3.12.

Por otro lado, para asegurar el correcto funcionamiento de todos los motores y limitar su rango de operación, se han establecido unas posiciones mínimas y máximas de funcionamiento. Para el **primer motor** de cada pata, estas posiciones difieren debido a su disposición en el cuerpo del robot (Tabla 3.1).

Tabla 3.1: Posiciones límite Motor 1

Pata	**Pos. mín. (º)**	**Pos. máx. (º)**	**Rango (º)**
1	-60	60	120
2	-70	30	100
3	-100	50	150
4	-50	100	150
5	-30	70	100
6	-60	60	120

En el caso de los **motores 2 y 3**, todos comparten las mismas posiciones límite, las cuales se pueden observar en las tablas 3.2 y 3.3.

Tabla 3.2: Posiciones límite Motor 2

Pata	**Pos. mín. (º)**	**Pos. máx. (º)**	**Rango (º)**
1, 2, 3, 4, 5, 6	-90	110	200

Tabla 3.3: Posiciones límite Motor 3

Pata	**Pos. mín. (º)**	**Pos. máx. (º)**	**Rango (º)**
1, 2, 3, 4, 5, 6	-120	130	250

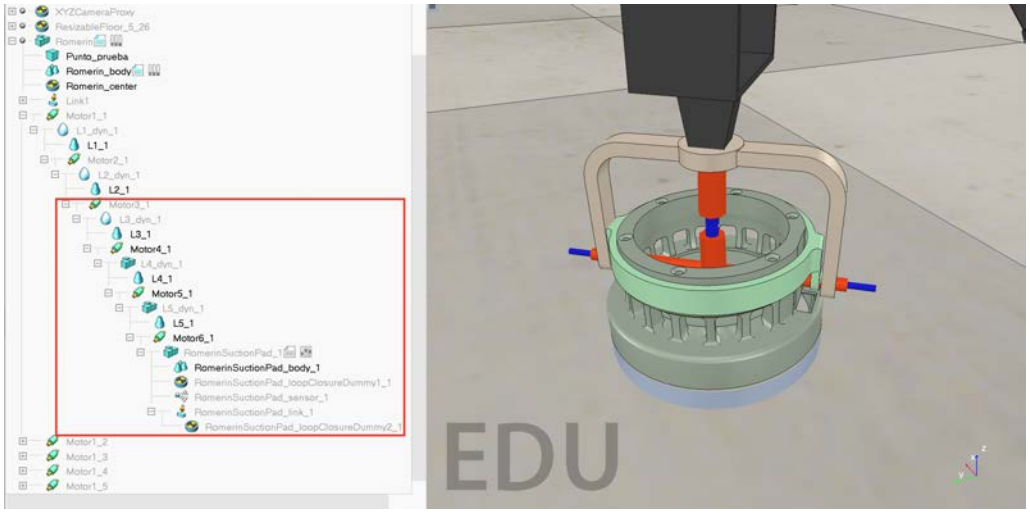


Figura 3.13: Jerarquía y montaje de una ventosa

3.2.3. Modelado de la ventosa

La ventosa que se ha modelado para que el hexápodo pueda adherirse y escalar por las paredes se puede observar con detalle en la Figura 3.13. Para ello, se han utilizado tres *joints* de revolución (aunque en el modelo final se oculta su visualización), que se encargan de unir los elementos y de modelar los tres grados de libertad libres de los que dispone la ventosa. Sin embargo, para evitar que los elementos de la ventosa se muevan con demasiada facilidad debido a la libertad de los *joints*, se ha modelado un pequeño **resorte** en cada una de las articulaciones mediante la incorporación de un **parámetro proporcional** de valor 0.1 (regulador P).

La implementación de la ventosa está basada en un componente llamado *Baxter-VacuumCup*, que se puede encontrar dentro de los modelos que tiene V-REP en su librería. Esta ventosa está provista de un sensor de proximidad, el cual se encarga de detectar objetos que se encuentren debajo de la ventosa. Cuando estos objetos se encuentran a una distancia menor que la distancia mínima establecida y se manda la orden de pegar la ventosa (mediante la función *sim.setIntegerSignal*), se realizará una unión física entre el primer *dummy* y el segundo, a través del sensor de fuerza. Cuando se envíe la orden de despegarla o, en su defecto, la fuerza contraria a la ejercida por la ventosa sea mayor que el parámetro de simulación que establece la fuerza de ruptura, la unión entre ambos *dummies* terminará y la ventosa quedará totalmente despegada del objeto o la superficie donde se encuentre.

Todo este modelado de comportamiento de la ventosa es posible gracias al *child script* que tiene asociada, el cual se puede observar en el fragmento de código 1.

Código 1: Child Script de la primera ventosa de ROMERIN en V-REP

```
1      -- Do following to activate/deactivate the vacuum cup from another location. For example:
2      --
3      -- vacuumCupName='RomerinSuctionPad#1' -- specify the full name. If the full name is
       ↪  "RomerinSuctionPad", specify "RomerinSuctionPad#"
4      -- sim.setIntegerSignal(vacuumCupName..'_active',1) -- activate
5      -- sim.setIntegerSignal(vacuumCupName..'_active',0) -- deactivate
6
7  function sysCall_init()
8      s=sim.getObjectHandle('RomerinSuctionPad_sensor_1')
```

```
    l=sim.getObjectHandle('RomerinSuctionPad_loopClosureDummy1_1')
    l2=sim.getObjectHandle('RomerinSuctionPad_loopClosureDummy2_1')
    b=sim.getObjectHandle('RomerinSuctionPad_1')
    objectName=sim.getObjectName(b)
    suctionPadLink=sim.getObjectHandle('RomerinSuctionPad_link_1')

    infiniteStrength=sim.getScriptSimulationParameter(sim.handle_self,'infiniteStrength')
    maxPullForce=sim.getScriptSimulationParameter(sim.handle_self,'maxPullForce')
    maxShearForce=sim.getScriptSimulationParameter(sim.handle_self,'maxShearForce')

    sim.setLinkDummy(l,-1)
    sim.setObjectParent(l,b,true)
    m=sim.getObjectMatrix(l2,-1)
    sim.setObjectMatrix(l,-1,m)

end


function sysCall_cleanup()
    sim.setLinkDummy(l,-1)
    sim.setObjectParent(l,b,true)
    m=sim.getObjectMatrix(l2,-1)
    sim.setObjectMatrix(l,-1,m)
end

function sysCall_sensing()
    parent=sim.getObjectParent(l)

    active=sim.getIntegerSignal(objectName..'_active')

    if (active~=1) then
        if (parent~=b) then
            sim.setLinkDummy(l,-1)
            sim.setObjectParent(l,b,true)
            m=sim.getObjectMatrix(l2,-1)
            sim.setObjectMatrix(l,-1,m)
        end
    else
        if (parent==b) then
            -- Here we want to detect a respondable shape, and then connect to it with a force sensor (via a loop closure dummy dummy link)
            -- However most respondable shapes are set to "non-detectable", so "sim.readProximitySensor" or similar will not work.
            -- But "sim.checkProximitySensor" or similar will work (they don't check the "detectable" flags), but we have to go through all shape objects!
            index=0
            while true do
                shape=sim.getObjects(index,sim.object_shape_type)
                if (shape==-1) then
                    break
                end
                if (shape~=b) and (sim.getObjectInt32Parameter(shape,sim.shapeintparam_respondable)~=0) and (sim.checkProximitySensor(s,shape)==1) then
                    -- Ok, we found a respondable shape that was detected
                    -- We connect to that shape:
                    -- Make sure the two dummies are initially coincident:
                    sim.setObjectParent(l,b,true)
                    m=sim.getObjectMatrix(l2,-1)
                    sim.setObjectMatrix(l,-1,m)
                    -- Do the connection:
                    sim.setObjectParent(l,shape,true)
                    sim.setLinkDummy(l,l2)
                    break
                end
                index=index+1
            end
        else
            -- Here we have an object attached
            if (infiniteStrength==false) then
```

```
                -- We might have to conditionally beak it apart!
                result,force,torque=sim.readForceSensor(suctionPadLink) -- Here we read the median
                ↪  value out of 5 values (check the force sensor prop. dialog)
                if (result>0) then
                    breakIt=false
                    if (force[3]>maxPullForce) then breakIt=true end
                    sf=math.sqrt(force[1]*force[1]+force[2]*force[2])
                    if (sf>maxShearForce) then breakIt=true end
                    if (breakIt) then
                        -- We break the link:
                        sim.setLinkDummy(l,-1)
                        sim.setObjectParent(l,b,true)
                        m=sim.getObjectMatrix(l2,-1)
                        sim.setObjectMatrix(l,-1,m)
                    end
                end
            end
        end
    end
end
```

Este *script* está compuesto por tres funciones:

- La función **sysCall_init** es la encargada de inicializar tanto los componentes que forman el modelo de la ventosa en V-REP, como los parámetros de simulación relacionados con la ventosa (la fuerza máxima de tracción o la fuerza máxima de cizallamiento).
- La función **sysCall_cleanup** se encarga de realizar la limpieza de los parámetros indicados para que el script se ejecute desde un estado inicial predeterminado.
- La función **sysCall_sensing** contiene toda la lógica de funcionamiento de la ventosa. En función de si la ventosa se encuentra activa o no (parámetro *active*), se recorren las diferentes condiciones donde se lee el parámetro del sensor de proximidad que indica si existe un objeto lo suficientemente cercano como para poder establecer un **link físico** con él. En caso de encontrarse ya pegada la ventosa, el script se irá ejecutando en bucle dentro del tiempo de simulación para comprobar que la fuerza que ejerce la ventosa no supera a la fuerza máxima de tracción o la fuerza de cizallamiento. En caso de hacerlo, el enlace físico se romperá, por lo que la ventosa se despegará del objeto al que se encuentre pegado en ese momento.

Por otro lado, para poder realizar la activación de la ventosa desde cualquier otro script de la simulación, es necesario utilizar la instrucción que se expone a continuación, donde *vacuumCupName* se corresponde con el nombre del elemento de la ventosa dentro del simulador (por ejemplo, *'RomerinSuctionPad_1'* para la primera ventosa de ROMERIN):

```
sim.setIntegerSignal(vacuumCupName..'_active',1) -- Activacion de la ventosa
```

A su vez, para poder desactivar correctamente la ventosa desde cualquier otro script, se utilizará la siguiente instrucción:

```
sim.setIntegerSignal(vacuumCupName..'_active',0) -- Desactivacion de la ventosa
```

3.2.4. Modelado de la Euclid y su soporte

Para que el robot hexápodo ROMERIN pueda realizar las tareas de navegación correctamente, es necesario dotar al sistema de un sistema de visión 3D que permita analizar el entorno en el que se encuentra el robot a medida que se desplaza y establecer los algoritmos de movimiento adecuados para cada tipo de superficie.

Por ello, se ha escogido el kit de desarrollo *Intel Euclid*, ya que proporciona un gran detalle para los diferentes tipos de visionado a un bajo coste. Además, otra de las principales razones para escoger este kit de desarrollo es su compatibilidad con *ROS*, brindando así la posibilidad de realizar modificaciones en los componentes de manera muy sencilla.

Figura 3.14: Intel Euclid Development Kit

Las principales especificaciones de la *Intel Euclid* [12] se enumeran a continuación:

- CPU: Intel Atom x7-z8700
- Memoria RAM: 4 GB LPDDR3-1600
- Almacenamiento: 32 GBeMMC MLC 5.0 y la posibilidad de expansión mediante tarjeta micro SD (hasta 128 GB)
- Puertos E/S: Micro USB 3.0, USB OTG, UART, micro HDMI
- Comunicación inalámbrica mediante Bluetooth y Wi-Fi
- GPS: GNS, GLONASS, Beidou, Galileo, QZSS, WAAS, EGNOS
- Sensores: Integrated Sensor Hub (ISH), acelerómetro, brújula digital, giroscopio, luz ambiental, proximidad, barómetro, altímetro, de humedad, de temperatura

Figura 3.15: Soporte de la *Intel Euclid* y de la *PCB*

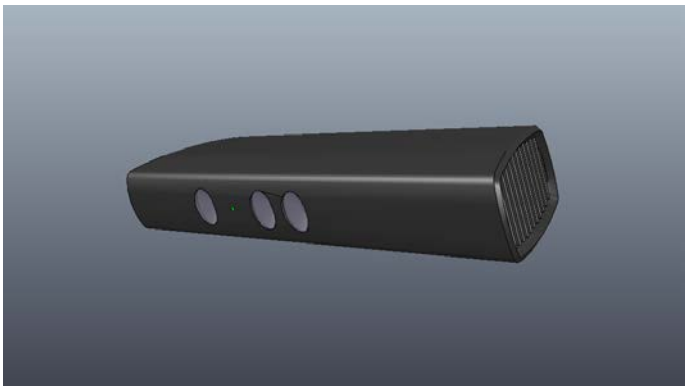


Figura 3.16: Adaptación de la *Intel Euclid* en V-REP

- Consumo energético: 5W nominal
- Batería: 3.8V 2000 mAh
- Sistema Operativo: Ubuntu 16.04

A pesar de que no existe un modelo implementado de la *Intel Euclid* dentro de V-REP, sí que existe una implementación de la cámara **Kinect** de *Microsoft*, que permite obtener una visión del entorno del robot en el simulador, así como obtener un mapa de profundidad. Por lo tanto, para los requisitos deseados para la simulación, se han realizado unas modificaciones para adaptar este modelo de la **Kinect** para que cumpla con las necesidades del proyecto.

En primer lugar, se ha importado un archivo de extensión *.stl* del soporte que se encargará de la sujeción de la *Intel Euclid* al robot (Figura 3.15), de manera que la cámara tenga una inclinación de **20 grados**, ya que esta inclinación es la que proporciona las lecturas más precisas. Además, este soporte también servirá para albergar la *PCB* que se encargará del control del hexápodo.

Por otro lado, se han realizado modificaciones en **tamaño**, **peso** y **geometría** para adaptar el modelo de la **Kinect** que se encuentra dentro de V-REP. El resultado puede observarse en la Figura 3.16.

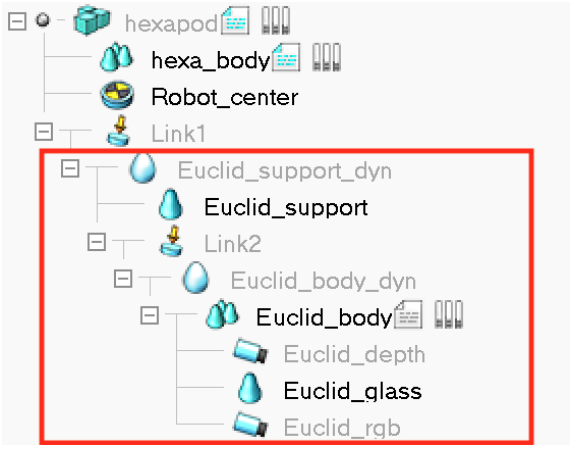


Figura 3.17: Jerarquía de montaje del soporte y la *Intel Euclid*

Figura 3.18: Soporte con la *Intel Euclid* acoplada

Para acoplar tanto el soporte como el modelo de la *Intel Euclid* al robot, es necesario tener en cuenta las consideraciones que se han visto en el Capítulo 3.2.1. Por tanto, la **jerarquía** de montaje sería la que se observa en la Figura 3.17.

Para realizar la unión física entre la base del robot y el soporte, se ha empleado un **sensor de fuerza** (*Link1*). El soporte está constituido por el **modelo dinámico** (*Euclid_support_dyn*) que contiene el peso del soporte y los parámetros necesarios para responder correctamente las respuestas dinámicas, y por el **modelo visual** (*Euclid_support*) que se trata de una *random shape* que sirve para dar la apariencia real al modelo.

A su vez, otro **sensor de fuerza** (*Link2*) es el encargado de realizar la unión entre el soporte y la *Intel Euclid*, constituida por el **modelo dinámico** (*Euclid_body _dyn*), el modelo visual (*Euclid_body*), los **sensores RGB y de profundidad** (*Euclid_rgb* y *Euclid_depth*, respectivamente), el **modelo visual** de las lentes (*Euclid_glass*) y un ***child script*** asociado, que es el encargado de obtener y registrar todos los datos que proporcionan los sensores.

Por lo tanto, el modelo resultante tras realizar todo el proceso de configuración y acoplamiento es el que se observa en la Figura 3.18.

Este montaje permite visualizar objetos que se encuentren por delante del robot

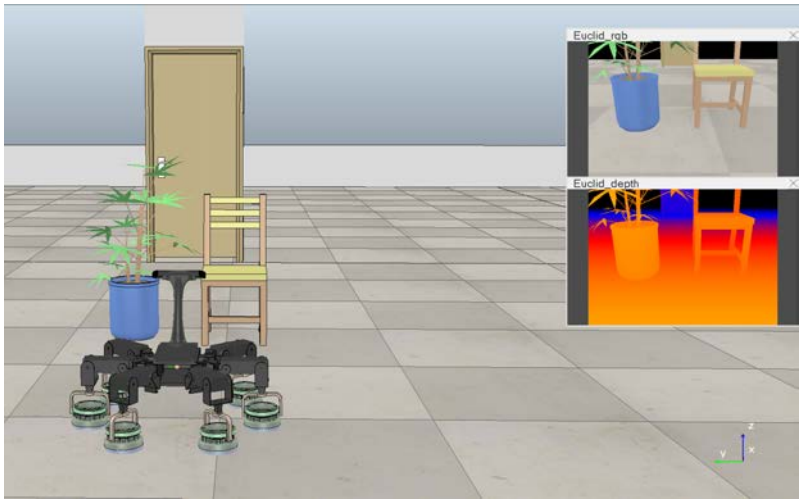


Figura 3.19: Captura de los sensores RGB y de profundidad de la *Intel Euclid* dentro de V-REP

(como se observa en la Figura 3.19), así como extraer la distancia a la que se encuentran, con el objetivo de poder extraer estos datos del simulador en un futuro e implementarlos en el control de ROMERIN.

3.2.5. Workspace de la pata

El *workspace* es el espacio donde el robot puede funcionar. La definición de *workspace* más comúnmente usada es la dada por [52]:

> "El workspace del manipulador de un robot se define como el conjunto de puntos que pueden ser alcanzados por su efector final."

Por el diseño que tiene el robot ROMERIN y con vista a futuro de crear un robot modular a partir de él, todas las patas tienen el mismo diseño y la misma programación, por lo que se ha obtenido el *workspace* de una de las patas (Figura 3.20), para poder observar qué puntos son alcanzables por el extremo de la pata y cuáles no.

Para poder obtener este *workspace*, se ha llevado a cabo una adaptación del *child script* que se encuentra dentro de una escena predefinida de V-REP, en la cual se calcula el *workspace* para el robot manipulador KUKA iiwa. El código del script utilizado se puede observar con más detalle en el código adjunto 9. Además, para que funcione correctamente, es necesario crear una *collection* de los objetos que componen el modelo (Figura 3.21).

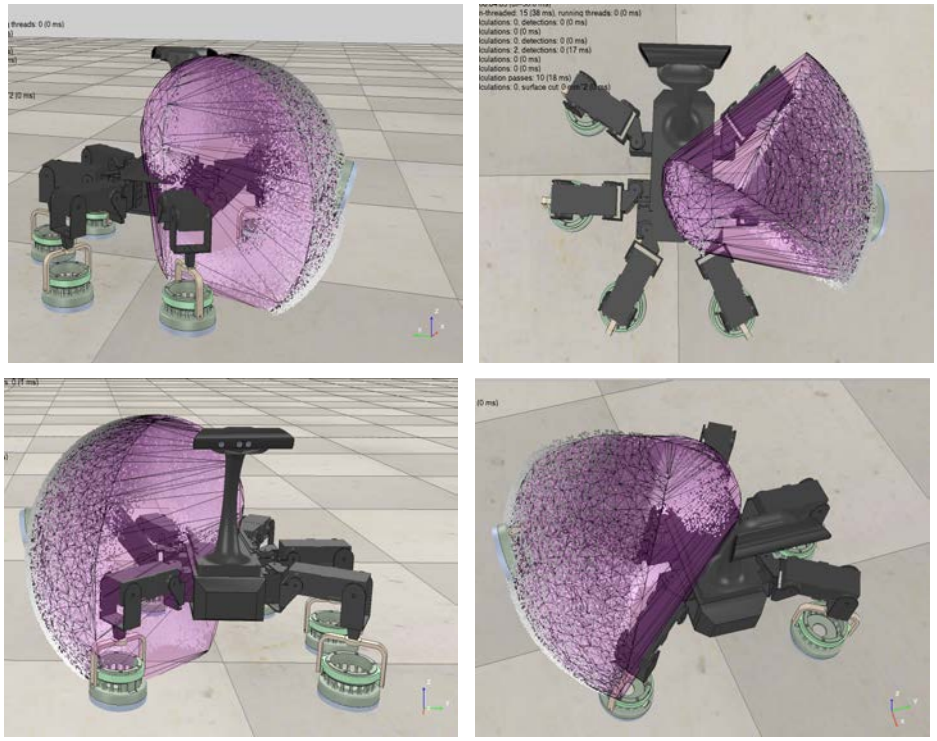


Figura 3.20: *Workspace* de la pata desde diferentes perspectivas

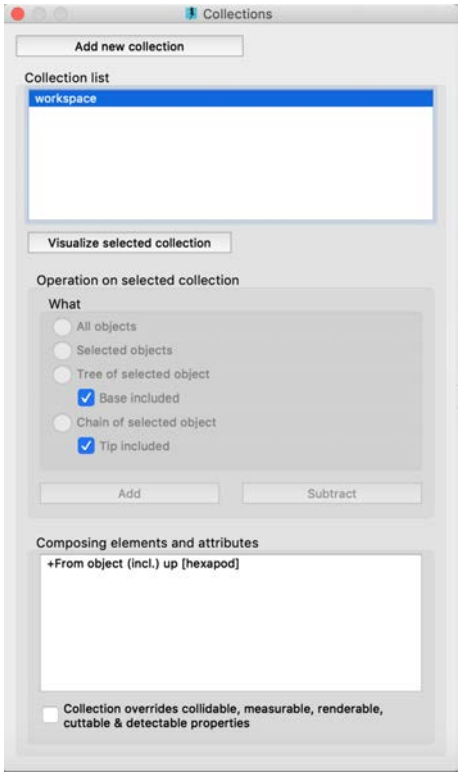


Figura 3.21: *Collection* de objetos para el cálculo del *workspace*

Capítulo 4

API Remota

La **API remota** es una de las funcionalidades más potentes que integra el simulador V-REP. Permite utilizar diversos lenguajes de programación para comunicar aplicaciones externas con los modelos que se encuentran dentro del simulador.

4.1. Motivos de utilización

Este proyecto se basa en la necesidad de implementar un modelo de simulación que sea lo más fiel posible al modelo real existente. Por ello, aunque dentro del simulador se pueden implementar módulos para calcular la cinemática directa o la cinemática inversa, o utilizar las funciones de la API regular para enviar la orden de pegar las ventosas, esto no es posible implementarlo dentro del robot real. Por tanto, los niveles de control existentes han de ser iguales tanto en el *firmware* del modelo real como en la simulación.

Una de las principales razones por las que se ha escogido el simulador V-REP frente a otros de sus competidores es esta, pues para nuestro caso resulta imprescindible el poder ejecutar un *script* externo donde se alojen todos los cálculos necesarios para el movimiento del robot.

4.2. Modo de funcionamiento

La **API remota** de V-REP [11] está formada por más de cien funciones específicas y una función genérica, las cuales pueden ser llamadas desde una aplicación externa en C/C++, Python, Java, Matlab/Octave o Lua. Esta API remota se comunica con V-REP mediante un *socket*, gracias a lo cual se consigue reducir el retardo que pueda existir en la comunicación entre la aplicación externa y el simulador. Además, esta API remota permite que varios programas externos interactúen con V-REP a la vez de manera síncrona o asíncrona.

La API remota se divide en dos entidades, las cuales se comunican entre ellas mediante un *socket*:

- El **cliente**, es decir, la aplicación externa: la API remota está disponible para diversos lenguajes de programación, como se ha indicado antes. Para poder configurar correctamente el cliente es necesario llevar a cabo una serie de pasos, los cuales se detallan en el Apéndice A.1.

- El **servidor**, es decir, V-REP: la API remota en el lado del servidor está implementada a través de un *plugin* de V-REP el cual es cargado por defecto por V-REP: *v_repExtRemoteApi.dll* (Windows), *libv_repExtRemoteApi.dylib* (Mac OS) o *libv_repExtRemoteApi.so* (Linux). Estos archivos serán necesarios incluirlos en el proyecto de la aplicación externa, como se explica con mayor detalle en el Apéndice A.2.

El modo de funcionamiento de la API remota [23] es similar al de la API regular, pero las diferencias residen en que la mayoría de las funciones de la API remota devuelven un valor similar (*return code*) y, además, todas las funciones de la API remota requieren de dos argumentos extra: un **modo de operación** y un **identificador de cliente** (clientID).

El modo de operación permite al usuario elegir la manera en que se ejecutará la tarea que se mande a través de la función remota. Es decir, la manera habitual de operar es enviar la tarea del cliente al servidor, y esperar a que el servidor procese la tarea y envíe una señal de vuelta al cliente. Sin embargo, este modo de operación puede implicar una gran carga de procesado y ralentizar la simulación, por lo que V-REP permite establecer los siguientes **modos de operación**:

- **Blocking function calls**: una llamada de función bloqueante se utiliza para los casos en los que no se puede estar esperando la respuesta por parte del servidor. En la Figura 4.1 se ilustra tal comportamiento.

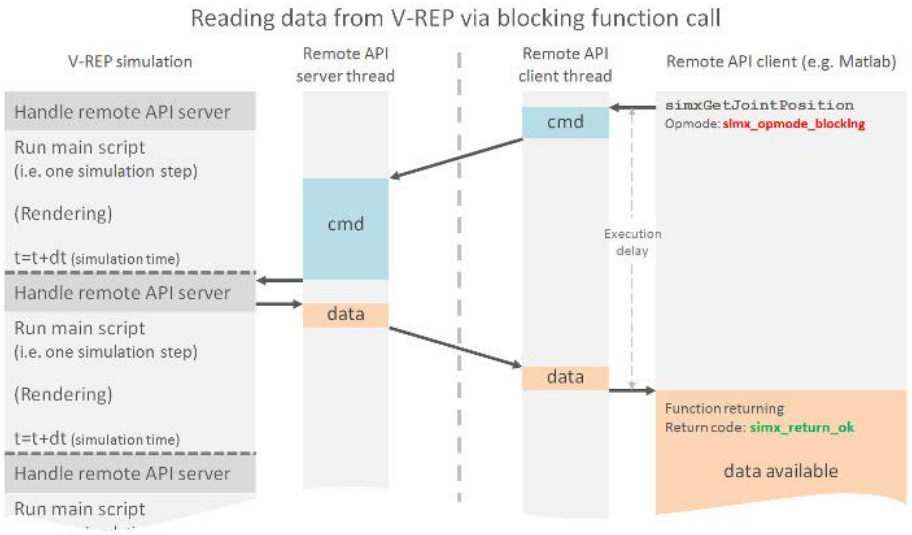


Figura 4.1: Modo de funcionamiento bloqueante (Fuente: [23])

- **Non-blocking function calls**: una llamada de función no bloqueante se utiliza cuando se necesita enviar información a V-REP sin la necesidad de una respuesta por parte del servidor, tal y como se puede observar en la figura 4.2.

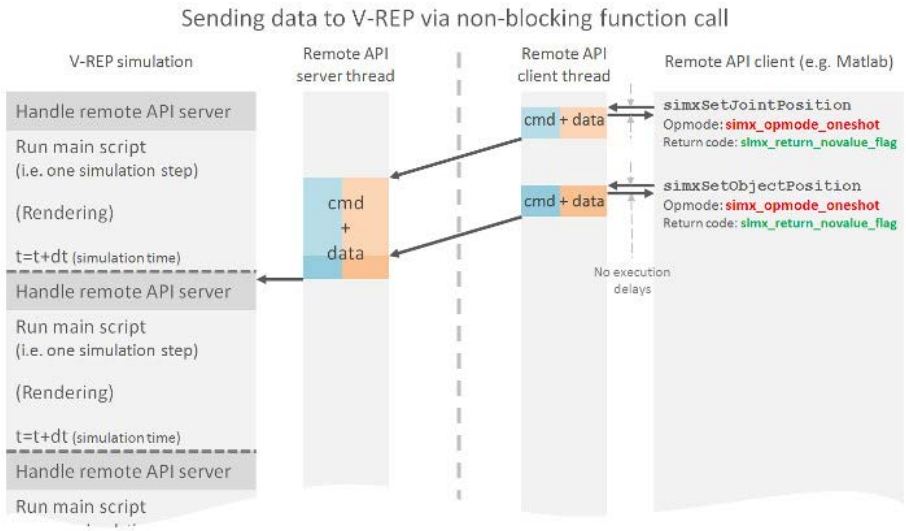


Figura 4.2: Modo de funcionamiento no bloqueante (Fuente: [23])

En ciertas ocasiones, el usuario necesita enviar varias órdenes y que todas se ejecuten dentro del simulador en el mismo instante. Esta situación se presenta, por ejemplo, cuando se quieren mover tres *joints* al mismo tiempo. Para ello, es necesario pausar momentáneamente la comunicación mediante la instrucción *simxPauseCommunication* [1], como refleja el fragmento de código 2.

Código 2: Establecer posicion de 3 joints a la vez (Fuente: [23])

```
1 simxPauseCommunication(clientID,1);
2 simxSetJointPosition(clientID,joint1Handle,joint1Value,simx_opmode_oneshot);
3 simxSetJointPosition(clientID,joint2Handle,joint2Value,simx_opmode_oneshot);
4 simxSetJointPosition(clientID,joint3Handle,joint3Value,simx_opmode_oneshot);
5 simxPauseCommunication(clientID,0);
```

Esta pausa en la comunicación se refleja en la Figura 4.3, donde se puede apreciar como el **thread del cliente** de la API Remota almacena todas las órdenes enviadas durante la pausa de la comunicación, para posteriormente enviar todas juntas al **servidor**.

[1] http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctions.htm#simxPauseCommunication

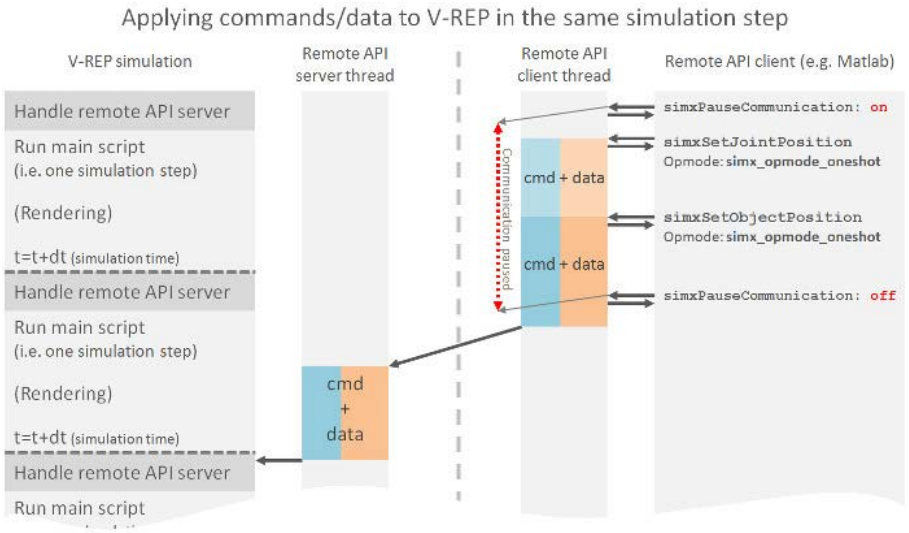


Figura 4.3: Envío de múltiples instrucciones a la vez desde el cliente (Fuente: [23])

- **Data streaming**: este modo de operación permite que el servidor pueda anticipar el tipo de datos que requiere el cliente. Para que esto sea posible, el cliente tiene que establecer un indicador al servidor si desea un modo de operación *"streaming"* o *"continuous"*. Esto quiere decir que la función se almacena en el servidor y se envía de manera regular al cliente, sin necesidad de que este tenga que realizar una petición cada vez que quiera recibir una respuesta.

  En la Figura 4.4 se puede observar un esquemático que ilustra este modo de operación:

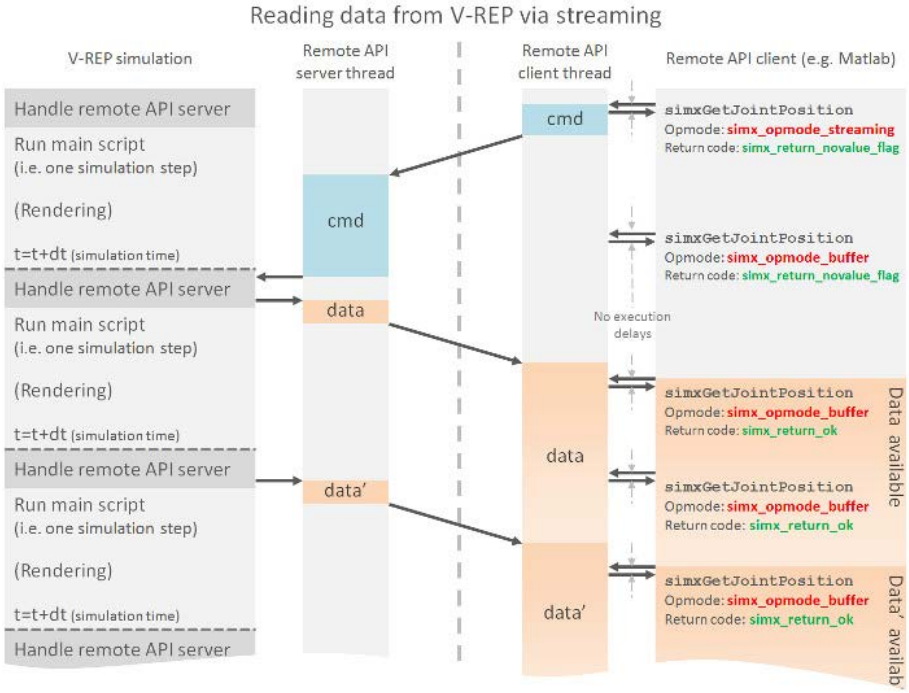


Figura 4.4: Modo de funcionamiento *data streaming* (Fuente: [23])

- **Synchronous operation**: a diferencia de los modos anteriores, donde la simu-

lación sigue ejecutándose sin esperar la respuesta por parte del cliente de que la orden enviada se ha ejecutado correctamente, este modo de funcionamiento permite establecer una **sincronización** entre el cliente y la simulación (servidor). De esta forma, habilitando al servidor de la API remota a operar en este modo, es posible conseguir que el cliente no continúe con su ejecución hasta que no reciba la respuesta del servidor, tal y como refleja la Figura 4.5.

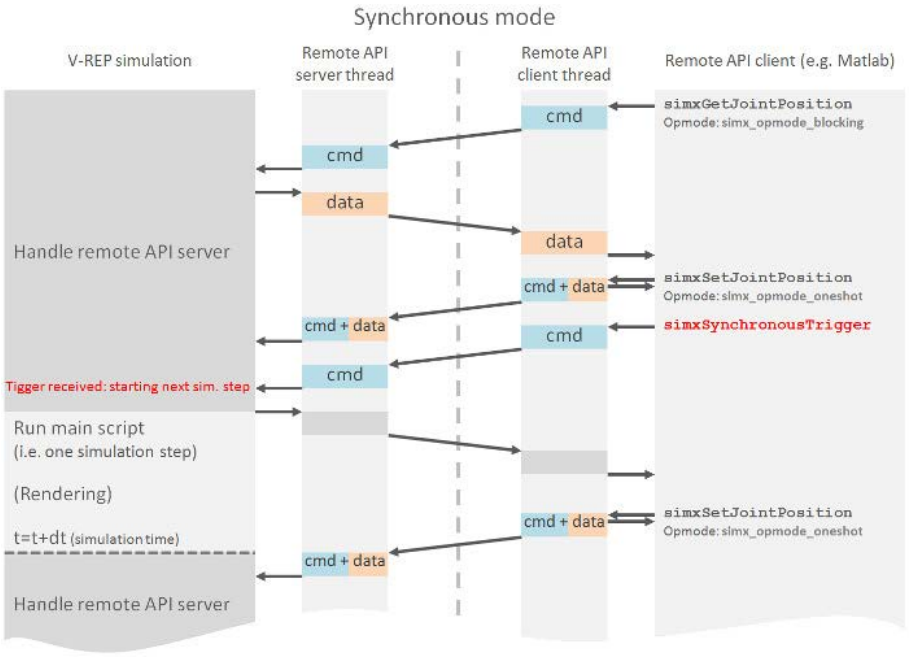


Figura 4.5: Modo de funcionamiento síncrono (Fuente: [23])

Al realizar la llamada a *simxSynchronousTrigger*, el siguiente paso de simulación empieza a ejecutarse, por lo que hay que tener precaución para leer los datos correctos y no los del paso de simulación anterior o posterior.

En este proyecto se ha empleado principalmente el modo **synchronous operation**, ya que es de vital importancia asegurar que el robot haya alcanzado la posición que se le ha enviado para poder continuar con las siguientes instrucciones.

4.3. Modelo cinemático

La **cinemática** de un robot constituye el estudio del movimiento del mismo con respecto a un sistema de referencia, sin tener en cuenta las fuerzas o pares que lo originan. Es decir, no se tienen en cuenta ni las *masas* ni las *inercias*. Gracias a la cinemática se puede obtener la relación entre el valor de las **coordenadas articulares** (q_1, ..., q_n) y la **posición y orientación del extremo** (Figura 4.6).

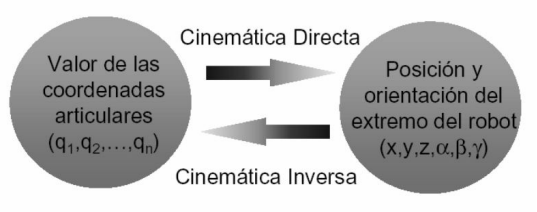


Figura 4.6: Relación entre la cinemática directa y la cinemática inversa

4.3.1. Cinemática directa

Para que el robot pueda alcanzar correctamente las posiciones deseadas, es necesario desarrollar la **cinemática** del mismo. Por un lado está la **cinemática directa**, que consiste en determinar la posición y la orientación del extremo del robot conociendo el valor de los ángulos de cada articulación y de la dimensiones geométricas de los elementos que forman la pata.

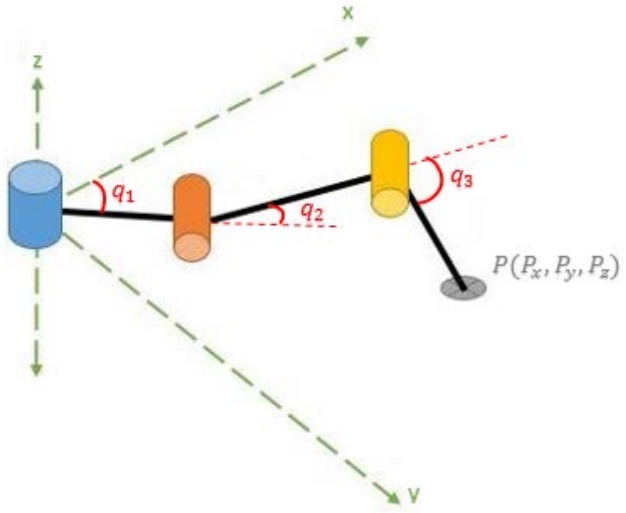


Figura 4.7: Esquema cinemática directa

Para el modelo desarrollado en este proyecto, se ha utilizado el método de Denavit-Hartenberg para hallar las ecuaciones que determinan la posición del extremo del robot a partir de las coordenadas articulares.

$$P_x = L_{C_{xy}} \cdot cos(Ang_C + q_1) + L_F \cdot cos(q_2) \cdot sin(q_1) + L_T \cdot cos(q_2 + q_3) \cdot sin(q_1) \quad (4.1)$$

$$P_y = L_{C_{xy}} \cdot sin(Ang_C + q_1) + L_F \cdot cos(q_2) \cdot cos(q_1) + L_T \cdot cos(q_2 + q_3) \cdot cos(q_1) \quad (4.2)$$

$$P_z = L_{C_z} + L_F \cdot sin(q_2) + L_T \cdot cos(q_2 + q_3) \cdot sin(q_2 + q_3) \quad (4.3)$$

En la función "*getFK*"que se puede observar dentro de la clase "*Leg*"del código adjunto 11 se implementan estas ecuaciones para poder calcular la posición del

extremo del robot, la cual se encuentra en la intersección entre el $joint_5$ y el $joint_6$ de cada pata.

4.3.2. Cinemática inversa

Por otro lado se encuentra la **cinemática inversa**, que es la técnica que permite obtener el valor que tienen las coordenadas articulares del robot para una determinada posición y orientación del extremo.

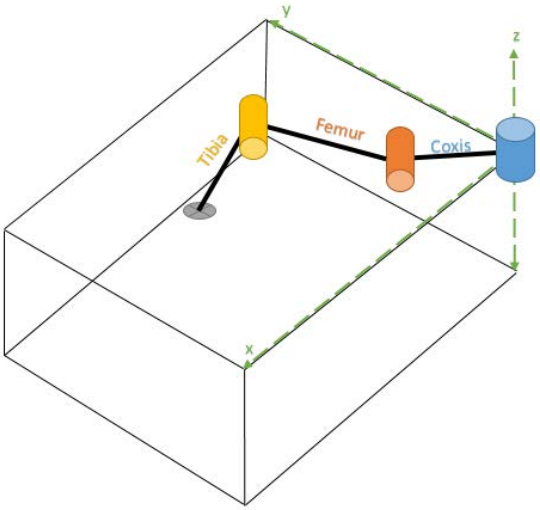


Figura 4.8: Esquema cinemática inversa

Para su resolución se han empleado una serie de **métodos geométricos** que se van a exponer a continuación.

En primer lugar, se calcula el ángulo de la primera articulación (q_1). Para ello, es necesario visualizar la pata del robot en un plano XY, como se ve en la Figura 4.9.

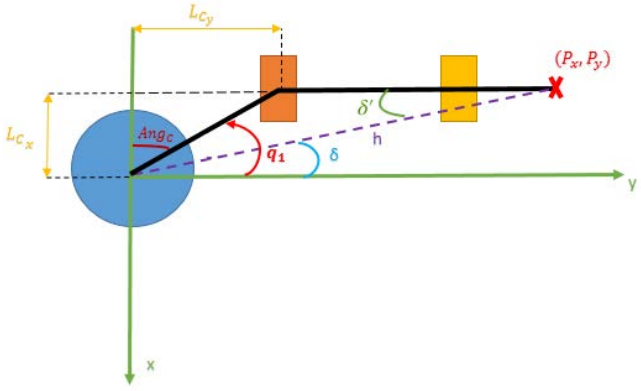


Figura 4.9: Perspectiva de la pata en el plano XY

Donde:

$$L_{C_x} = L_C \cdot cos(Ang_C) \tag{4.4}$$

$$L_{C_y} = L_C \cdot sin(Ang_C) \tag{4.5}$$

De la Figura 4.9, se puede extraer el triángulo de la Figura 4.10, donde:

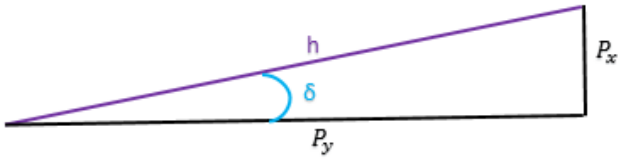


Figura 4.10: Primer triángulo obtenido de la Figura 4.9

$$h = \sqrt{P_x^2 + P_y^2} \tag{4.6}$$

$$\delta = tg^{-1}\left(\frac{-P_x}{P_y}\right) \tag{4.7}$$

Por otro lado, también se puede extraer de la Figura 4.9 el triángulo de la Figura 4.11 y obtener las siguientes ecuaciones:

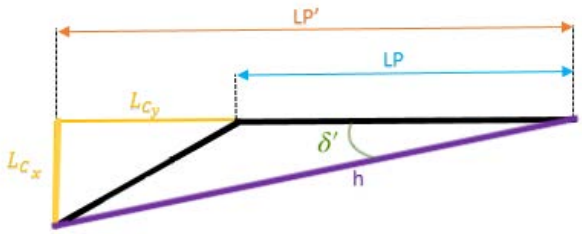


Figura 4.11: Segundo triángulo obtenido de la Figura 4.9

$$LP' = \sqrt{h^2 - L_{C_x}^2} \tag{4.8}$$

$$LP = LP' - L_{C_y} \tag{4.9}$$

$$\delta' = tg^{-1}\left(\frac{L_{C_x}}{LP'}\right) \tag{4.10}$$

De esta manera, se obtiene el ángulo de la primera articulación:

$$q_1 = \delta + \delta' \tag{4.11}$$

Una vez hallado el primer ángulo, se procede a observar la pata del robot desde una perspectiva frontal, como refleja la Figura 4.12. De aquí se puede obtener la distancia HF:

$$HF = \sqrt{LP^2 + (P_z + L_{C_z})^2} \tag{4.12}$$

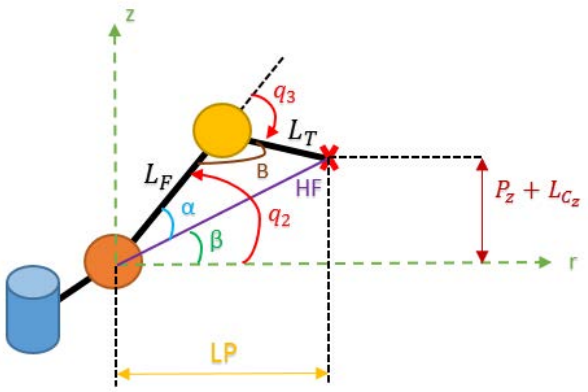


Figura 4.12: Perspectiva frontal de la pata en el plano ZR

Aplicando el teorema del coseno:

$$HF^2 = L_F^2 + L_T^2 - 2 \cdot L_F \cdot L_T \cdot cosB \tag{4.13}$$

Realizando un cambio al ángulo complementario, se obtendría:

$$HF^2 = L_F^2 + L_T^2 + 2 \cdot L_F \cdot L_T \cdot cos(q_3) \tag{4.14}$$

$$cos(q_3) = \frac{HF^2 - L_F^2 - L_T^2}{2 \cdot L_F \cdot L_T} \tag{4.15}$$

Por otro lado, se va a obtener la expresión del *seno* del ángulo de la tercera articulación (q_3), ya que en cuanto a carga computacional es mejor utilizar la *arcotangente* antes que el *arcocoseno*.

$$cos(q_3)^2 + sin(q_3)^2 = 1 \rightarrow sin(q_3) = \pm\sqrt{1 - cos(q_3)^2} \tag{4.16}$$

En la expresión 4.16, el **signo positivo** se corresponde con la configuración **codo abajo**, y el **signo negativo** con la configuración **codo arriba**.

Por tanto, para obtener el ángulo de la tercera articulación, se puede utilizar la expresión de la *arcotangente*:

$$q_3 = tg^{-1}\left[\frac{sin(q_3)}{cos(q_3)}\right] \tag{4.17}$$

Por último, para hallar el ángulo de la segunda articulación:

$$\beta = tg^{-1}\left(\frac{P_z + L_{C_z}}{LP}\right) \tag{4.18}$$

$$\alpha = tg^{-1}\left(\frac{L_T \cdot sin(q_3)}{L_F + L_T \cdot cos(q_3)}\right) \tag{4.19}$$

$$q_2 = \beta - \alpha \tag{4.20}$$

4.4. Matrices de transformación homogéneas

Puesto que los cálculos que se realicen para cada una de las patas deben estar referidos al centro del robot, es necesario realizar una serie de transformaciones para que el **punto en el espacio** se encuentre orientado correctamente al **centro del robot**.

La relación existente entre las coordenadas del punto en el espacio y las referidas a la pata se puede observar en la ecuación 4.21.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}_{MUNDO} = MTH_N \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}_{PATA} \tag{4.21}$$

MTH_N es la **matriz de transformación homogénea** del elemento N. Por lo tanto, para poder obtener las coordenadas referidas a la pata, las cuales son las que utiliza la cinemática inversa 4.3.2, es necesario despejar la ecuación 4.21.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}_{PATA} = MTH_N^{-1} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}_{MUNDO} \tag{4.22}$$

Esta matriz posee una dimensión de 4x4 y permite realizar la transformación de un vector de coordenadas homogéneas de un sistema de coordenadas a otro [21]. La matriz de transformación se compone de las submatrices que se pueden observar en la ecuación 4.23. La submatriz de la Perspectiva (f_{1x3}) se considera nula y la submatriz de Escalado (w_{1x1}) es la unidad.

$$MTH = \begin{bmatrix} R_{3x3} & p_{3x1} \\ f_{1x3} & w_{1x1} \end{bmatrix} = \begin{bmatrix} Rotacion & Traslacion \\ Perspectiva & Escalado \end{bmatrix} \tag{4.23}$$

Por tanto, las submatrices que influyen en la posición del robot son la de **rotación** y la de **traslación**.

La **matriz de rotación** tiene la forma que se observa en la ecuación 4.24, donde α es el ángulo de rotación del eje de coordenadas de la pata respecto al eje del centro del robot. Para calcular este ángulo de rotación es necesario tener en cuenta la orientación de los ejes de referencia de cada pata (Figura 4.13). De esta manera se obtiene, por ejemplo, que la Pata 1 tiene una rotación de 180º.

$$R_{3x3} = \begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.24}$$

Por otro lado, para obtener la **matriz de traslación** (ecuación 4.25) es necesario conocer las distancias existentes entre los ejes de la primera articulación de cada pata respecto al centro del robot. Estas distancias se pueden observar en la Figura 4.14.

$$p_{3x1} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \tag{4.25}$$

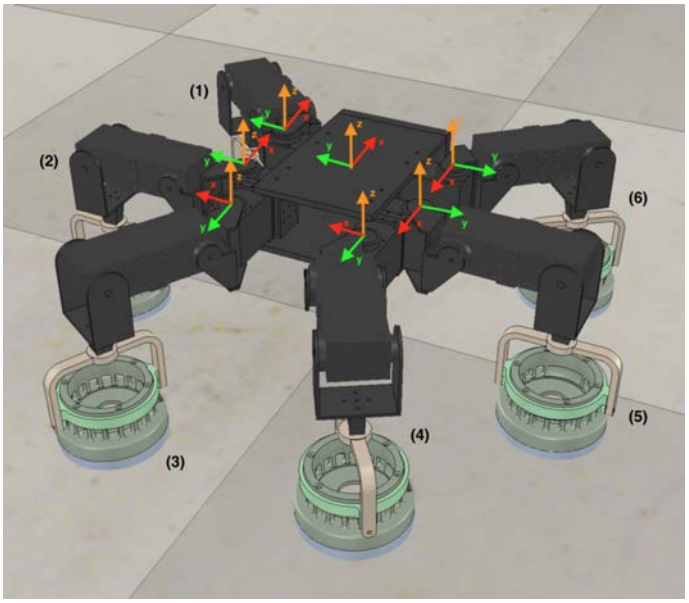


Figura 4.13: Ejes de coordenadas de cada pata y del centro del robot

Por lo tanto, teniendo en cuenta estas consideraciones, se pueden obtener las **matrices de transformación homogénea de todas las patas**.

$$MTH_{Pata_1} = \begin{bmatrix} -1 & 0 & 0 & 28{,}962 \\ 0 & -1 & 0 & -75{,}060 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.26}$$

$$MTH_{Pata_2} = \begin{bmatrix} 1 & 0 & 0 & 28{,}962 \\ 0 & 1 & 0 & 75{,}060 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.27}$$

$$MTH_{Pata_3} = \begin{bmatrix} -1 & 0 & 0 & -40{,}220 \\ 0 & -1 & 0 & -75{,}060 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.28}$$

$$MTH_{Pata_4} = \begin{bmatrix} 1 & 0 & 0 & -40{,}220 \\ 0 & 1 & 0 & 75{,}060 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.29}$$

$$MTH_{Pata_5} = \begin{bmatrix} 0 & -1 & 0 & -97{,}347 \\ 1 & 0 & 0 & -54{,}413 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.30}$$

$$MTH_{Pata_6} = \begin{bmatrix} 0 & -1 & 0 & -97{,}347 \\ 1 & 0 & 0 & 54{,}413 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.31}$$

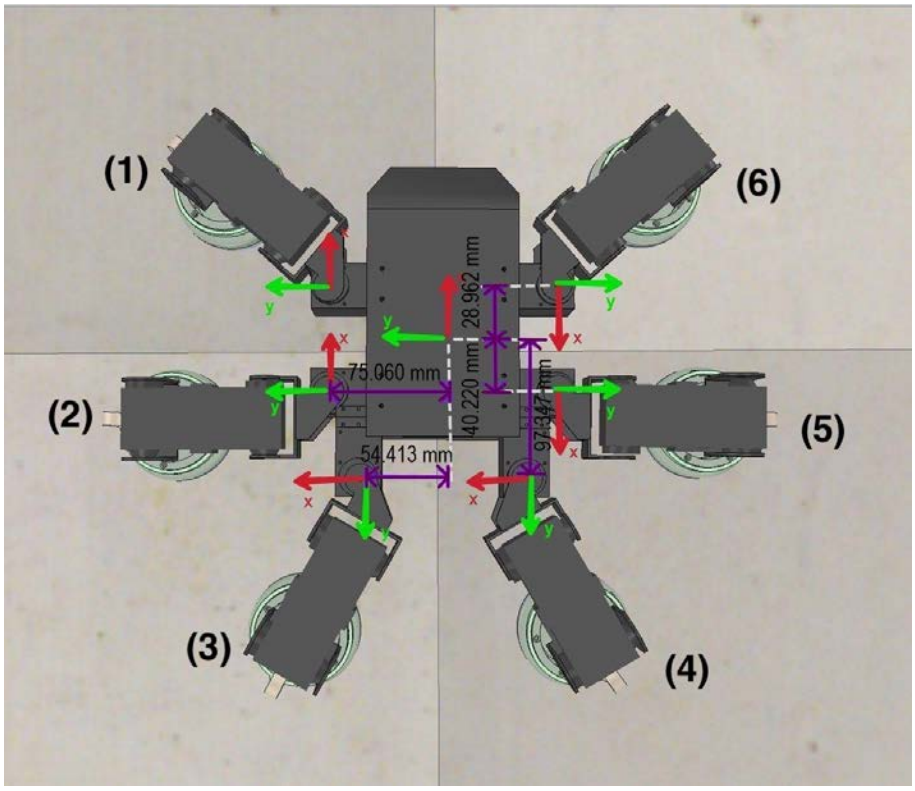


Figura 4.14: Distancia entre la primera articulación de cada pata y el centro del robot

4.5. Descripción de la API de ROMERIN

Con el propósito de tener una interfaz simplificada y común al control firmware, se ha desarrollado la API de ROMERIN. Esta **clase** se ha definido como la de mayor nivel, es decir, es el nivel de control más alto desde donde se realizarán las llamadas a los diferentes métodos de las clases **Leg** y **Joint**.

La clase **ROMERIN_API** está compuesta por las siguientes funciones:

Código 3: Cabecera constructor de ROMERIN_API

```
1 def _init_(self, motor11, motor21, motor31, ventosa1, motor12, motor22, motor32, ventosa2, motor13,
  ↪  motor23, motor33, ventosa3, motor14, motor24, motor34, ventosa4, motor15, motor25, motor35,
  ↪  ventosa5, motor16, motor26, motor36, ventosa6, clientID):
```

El constructor de la API de ROMERIN recibe como argumentos todos los elementos de V-REP que se necesitan para poder realizar correctamente el movimiento del robot. Por lo tanto, esta clase se construye con los 18 motores (tres por pata), y las 6 ventosas, cuyo parámetro representará la señal de activación o desactivación de las mismas en el simulador.

Código 4: Cabecera moveLeg de ROMERIN_API

```
1 def moveLeg (self, pata, vectorPos):
```

La función **moveLeg** recibe como argumentos las coordenadas referidas al eje de

coordenadas del mundo y la pata que se quiere mover para alcanzar dichas coordenadas. De esta manera, se realiza la transformación de las coordenadas del mundo en las relativas a la pata utilizando la ecuación 4.22. Tras esto, se llamará a la función *setPosition* de la clase **Leg** pasando como argumentos las coordenadas referidas a la pata. Esta función, en última instancia y pasando primero por la cinemática inversa, realizará la llamada a la función *setJointPosition* que establecerá el ángulo de cada motor para que la pata objetivo alcance la posición en el mundo deseada.

Código 5: Cabecera getPosition de ROMERIN_API

```
1 def getPosition (self, pata):
```

Esta función recibe como argumento la pata de la cual se quiere obtener la posición de su extremo. Por ende, esta función realizará la llamada a *getPosition* de la clase **Leg**, que a su vez llamará a la función *getJointPosition* de la clase **Joint** para obtener el ángulo de cada motor y, tras realizar los cálculos de la cinemática directa, devolverá la posición del extremo referida al eje de coordenadas de la pata. De este modo, la función *getPosition* de la **ROMERIN_API** realizará la transformación a las coordenadas relativas al mundo (Ecuación 4.21) las cuales constituirán el parámetro de retorno de la función.

Código 6: Cabecera setGripper de ROMERIN_API

```
1 def setGripper (self, pata, active):
```

La función **setGripper** se encarga de enviar la señal de activación (valor 1) o desactivación (valor 0) de la ventosa. Este parámetro se recoge en el argumento *active*, el cual se enviará a la función *setGripper* de la clase **Leg**, que se encarga de enviar la órden de activación o desactivación.

Código 7: Cabecera getGripper de ROMERIN_API

```
1 def getGripper (self, pata):
```

Esta función retorna el estado de la señal que controla el funcionamiento de la ventosa, es decir, si se encuentra activada (valor 1) o desactivada (valor 0).

Código 8: Cabecera setPosition de ROMERIN_API

```
1 def setAngles (self, pata, angles):
```

La función **setAngles** sirve para mover directamente los motores de la pata indicada a los ángulos que se envían como argumentos. Por tanto, desde esta función se hará la llamada a *setAngles* de la clase **Leg**, que llamará a su vez a *setJointPosition* de la clase **Joint** para establecer los ángulos deseados.

Capítulo 5

Resultados

Para la validación del modelo construido en V-REP, se han llevado a cabo una serie de pruebas dentro del simulador para poder extraer una serie de resultados, los cuales se desarrollan con detalle a lo largo de este capítulo.

5.1. Resultados de la cinemática

A lo largo de esta sección se detallan las pruebas que se han realizado sobre los motores de las patas, así como los resultados obtenidos, con el objetivo de verificar el correcto funcionamiento de la cinemática del robot. Para ello, se han desarrollado dos programas en Python que se comunican con el modelo de V-REP a través de la API Remota, mediante el uso de dos clientes que se ejecutan al mismo tiempo. El primero se encarga de mover los motores de la pata seleccionada a los ángulos que sitúan el extremo del robot en el punto deseado. El segundo cliente, durante el tiempo estipulado, lee continuamente la **posición real** que tiene el motor y la **posición ideal** que debería tener para, posteriormente, representar estos valores en una serie de gráficas. El código de este programa se puede observar con más detalle en el fragmento 13.

Se han seleccionado una serie de puntos a alcanzar con las posibles **configuraciones de codo** que puede tener cada pata (codo arriba o codo abajo). Las coordenadas de estos puntos y de los puntos finales se pueden observar en la Tabla 5.1.

Tabla 5.1: Coordenadas de los puntos de prueba y de los puntos finales

Pata	**Config. codo**	**Pos. inicial (mm)**	**Pos. final ideal (mm)**	**Pos. final real (mm)**
2	Codo arriba	[-56.91, 196.51, -96.29]	[25, 250, -40]	[23.85, 248.14, -50.42]
	Codo abajo			[15.59, 249.94, -45.908]
4	Codo arriba	[-210.91, -100.66, -96.29]	[-275, -40, 0]	[-274.27, -39.53, -12.71]
	Codo abajo			[-275.02, -48.80, -6.65]
6	Codo arriba	[119.69, -157.41, -96.29]	[25, -225, 68]	[25.63, 224.39, 55.65]
	Codo abajo			[32.28, -224.61, 58.14]

5.1.1. Análisis sobre la pata 2

La posición inicial de la pata 2 se puede observar en la Figura 5.1.

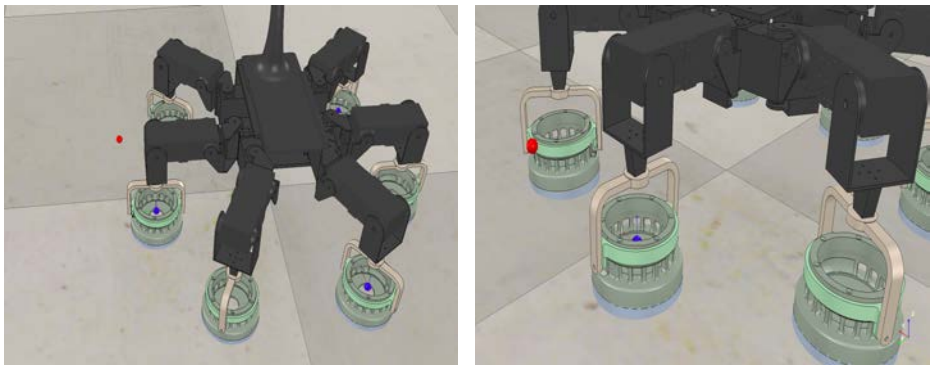


Figura 5.1: Posición de reposo de la pata 2 desde diferentes perspectivas

A partir de esta posición de reposo, se envía la orden de que el extremo de la pata (esfera de color azul) se sitúe en el punto objetivo (esfera de color rojo).

En primer lugar, se lleva a cabo la prueba con la configuración de **codo arriba**, dando como resultado el observado en la Figura 5.2. Este resultado es sorprendentemente bueno ya que, a pesar de presentar un pequeño error de unos 10 mm en el eje Z, el extremo de la pata consigue alcanzar la posición objetivo con gran exactitud.

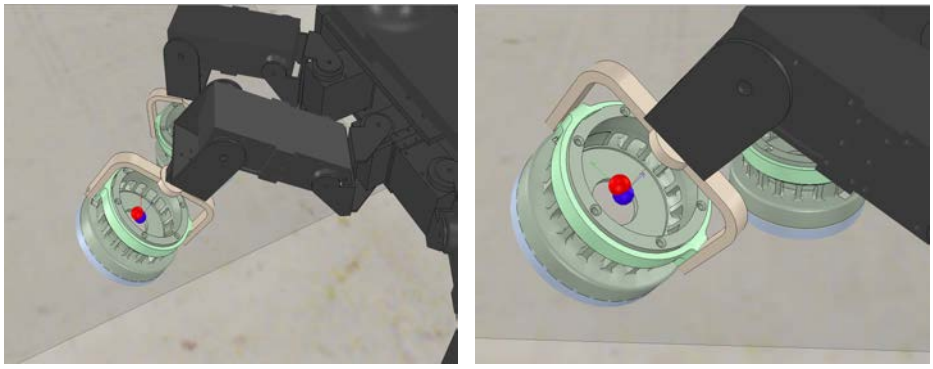


Figura 5.2: Posición objetivo de la pata 2 desde diferentes perspectivas (Codo arriba)

A través del programa en Python desarrollado para representar gráficas de la posición de los motores, se obtienen los resultados de los ángulos de los motores de la pata 2 (Figura 5.3).

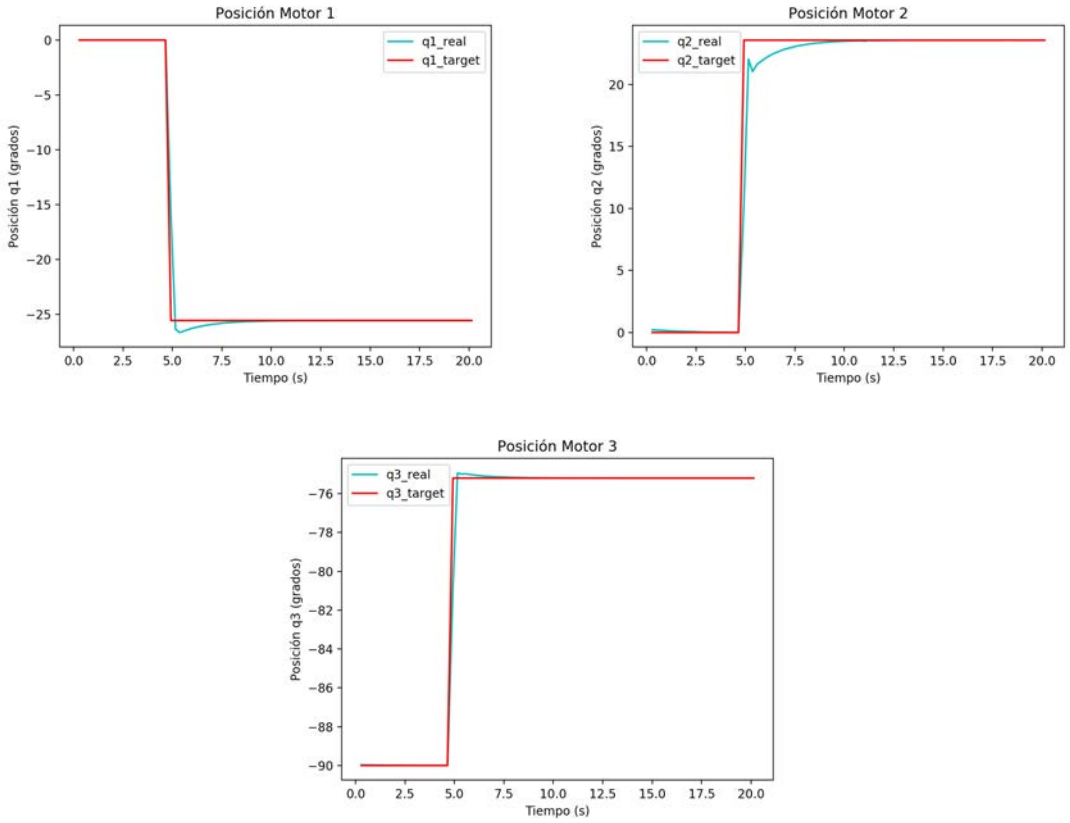


Figura 5.3: Gráficas de los motores de la pata 2 (Codo arriba)

Tal y como se puede observar en las gráficas, el motor 2 es el que más error presenta. Sin embargo, gracias al controlador de posición implementado en los motores, este error se consigue corregir y al cabo de unos pocos segundos el ángulo real alcanza el ángulo objetivo.

En segundo lugar, se realiza la misma prueba pero utilizando la configuración de **codo abajo**, como se puede observar en la Figura 5.4.

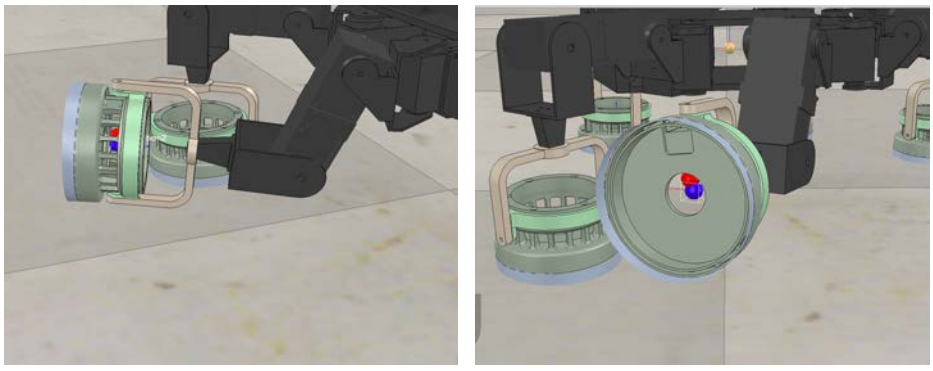


Figura 5.4: Posición objetivo de la pata 2 desde diferentes perspectivas (Codo abajo)

Al igual que para el caso anterior, se obtienen las gráficas de los motores de la pata 2 con la configuración de codo abajo (Figura 5.5).

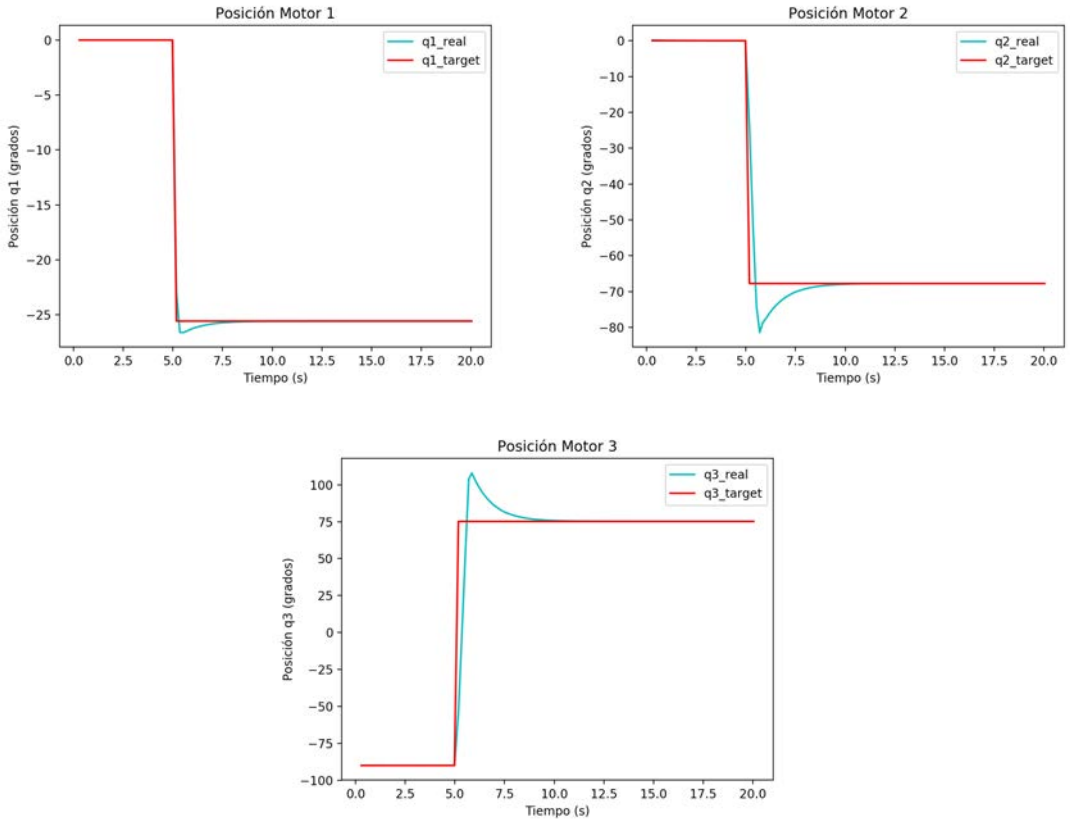


Figura 5.5: Gráficas de los motores de la pata 2 (Codo abajo)

En este caso son los motores 2 y 3 los que presentan más error, de nuevo corregido por el regulador PI.

5.1.2. Análisis sobre la pata 4

Para la pata 4, se lleva a cabo la misma prueba, pero modificando el punto objetivo. En este caso, se ha escogido un punto que se encuentra por debajo del origen de coordenadas del robot, con el objetivo de verificar que la cinemática funciona correctamente a la hora de alcanzar puntos con una componente Z negativa.

La posición de reposo de la pata 4 se puede observar en la Figura 5.6.

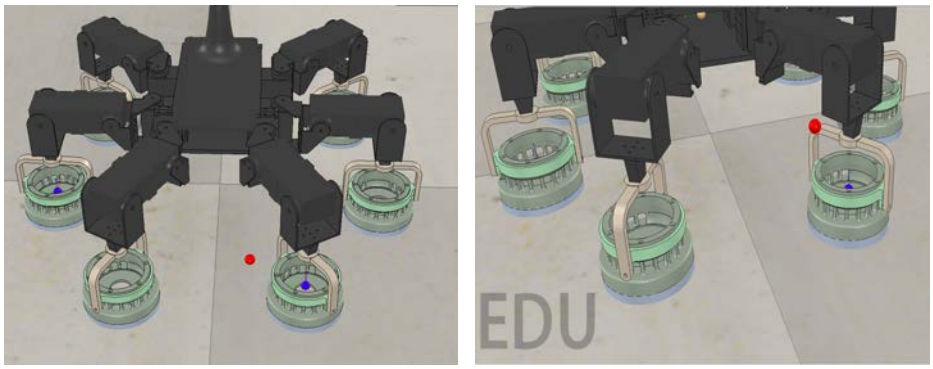


Figura 5.6: Posición de reposo de la pata 4 desde diferentes perspectivas

El resultado de la configuración del **codo arriba** se puede observar en la Figura 5.7. Para esta configuración, se obtienen las gráficas de los motores de la pata 4 (Figura 5.8).

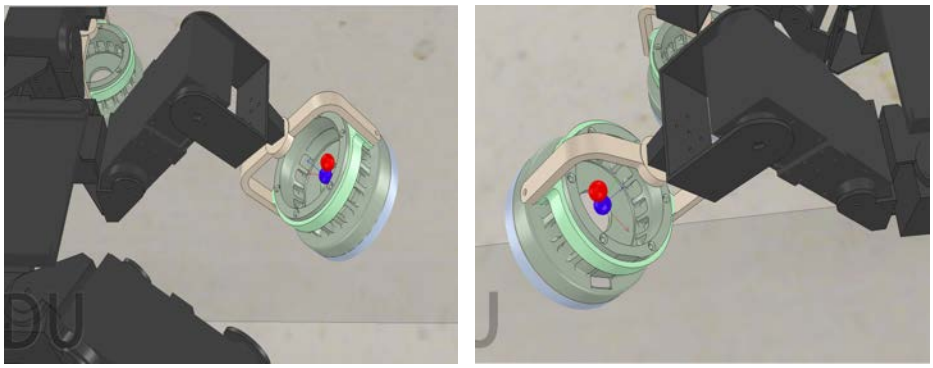


Figura 5.7: Posición objetivo de la pata 4 desde diferentes perspectivas (Codo arriba)

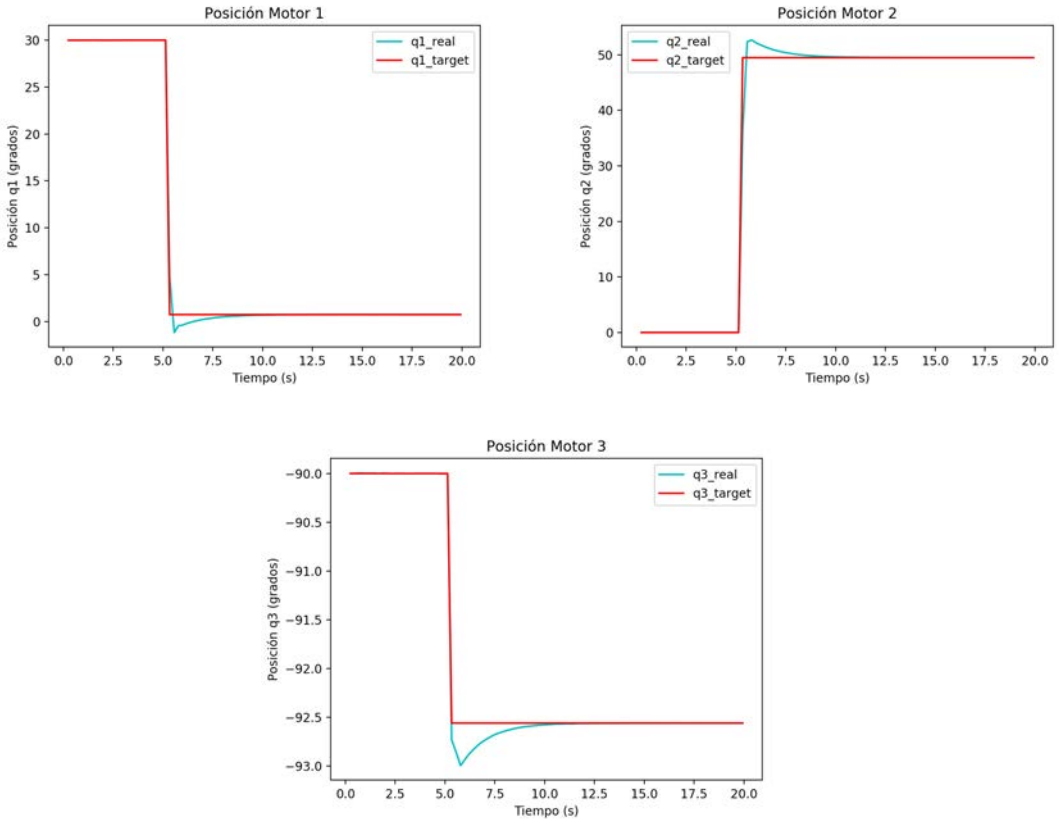


Figura 5.8: Gráficas de los motores de la pata 4 (Codo arriba)

Por otro lado, para la configuración de **codo abajo** (Figura 5.9), se obtiene el conjunto de gráficas que se puede apreciar en la Figura 5.10.

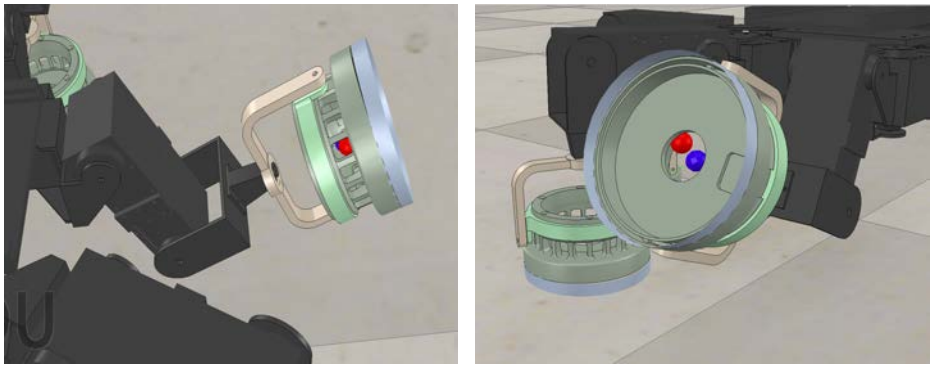


Figura 5.9: Posición objetivo de la pata 4 desde diferentes perspectivas (Codo abajo)

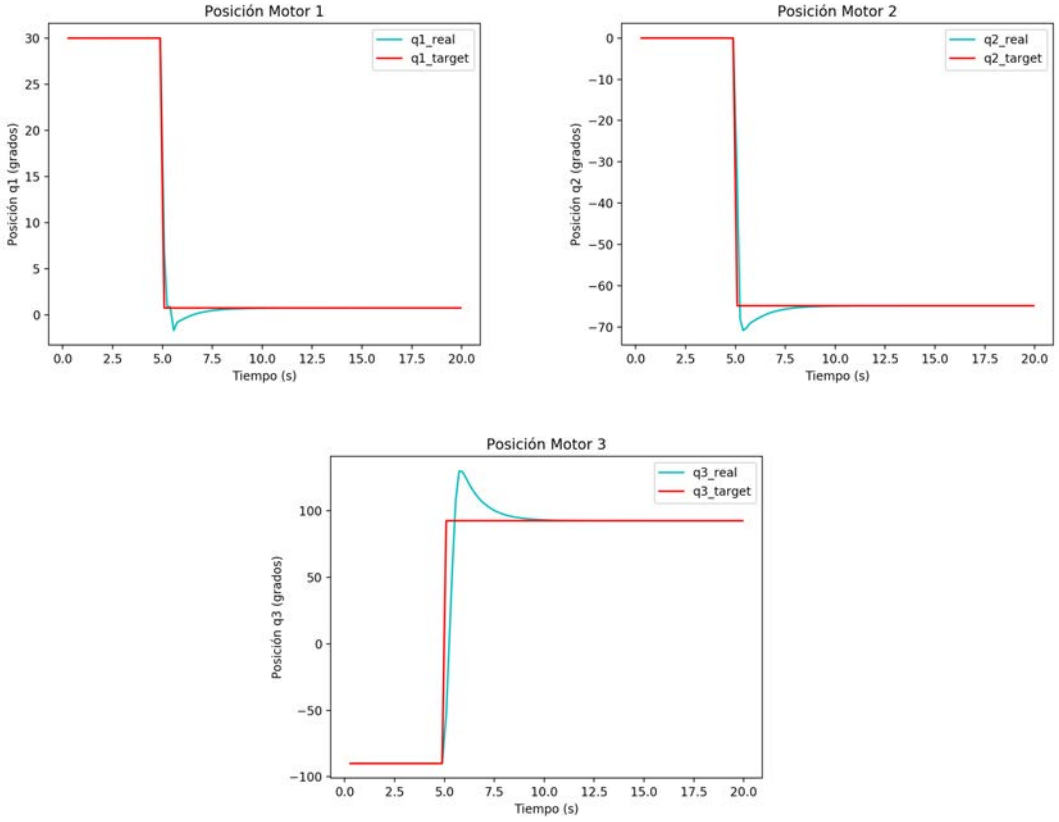


Figura 5.10: Gráficas de los motores de la pata 4 (Codo abajo)

Como se puede observar tanto en las gráficas de codo arriba como de codo abajo, los resultados son similares a los obtenidos en la pata 2. La posición objetivo se alcanza al cabo de unos segundos, comprobando así que el funcionamiento del regulador PI y los cálculos realizados por la cinemática inversa son los adecuados.

5.1.3. Análisis sobre la pata 6

En último lugar, para finalizar la batería de pruebas que se han llevado a cabo sobre la cinemática de ROMERIN, se pone a prueba el funcionamiento de la pata 6. En este caso, se ha escogido un punto objetivo que tiene una componente en Z positiva, con el propósito de comprobar el funcionamiento de la cinemática a la hora de tener que alcanzar un punto que se encuentra notablemente por encima del centro del robot.

Para ello, la posición inicial para la pata 6 se observa en la Figura 5.11.

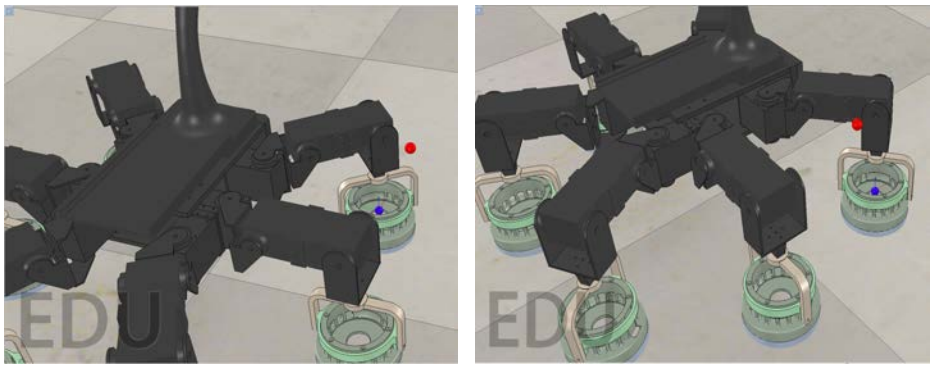


Figura 5.11: Posición de reposo de la pata 6 desde diferentes perspectivas

Para la configuración de **codo arriba** (Figura 5.12), se representan las gráficas de los motores de la Figura 5.13. En ellas se puede observar que, a pesar de tratarse de una posición algo más complicada de mantener debido a la componente Z, los motores de la pata alcanzan de manera muy precisa los ángulos objetivos, validando de nuevo los cálculos de la cinemática inversa.

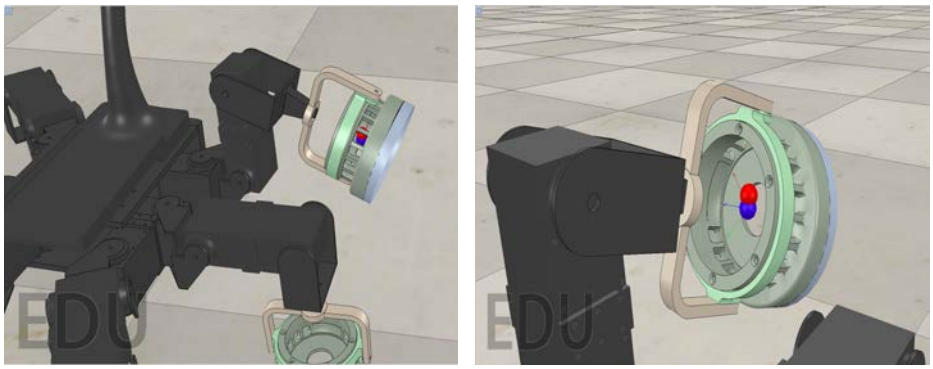


Figura 5.12: Posición objetivo de la pata 6 desde diferentes perspectivas (Codo arriba)

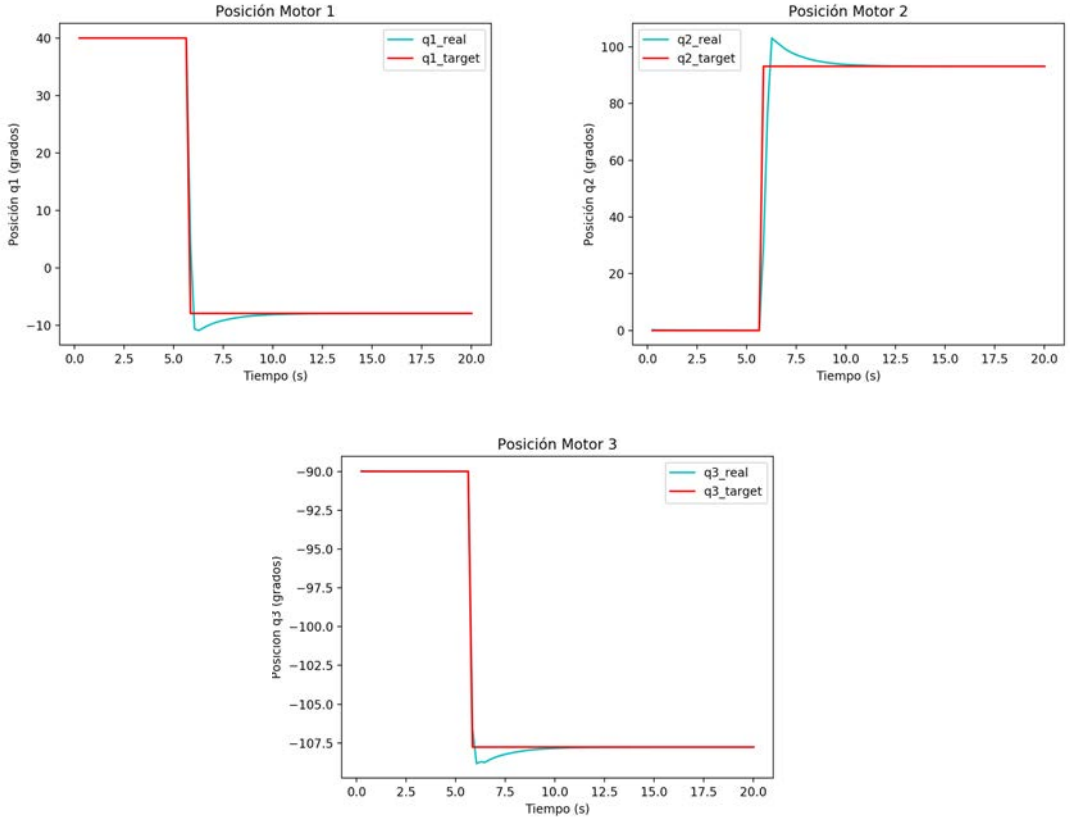


Figura 5.13: Gráficas de los motores de la pata 6 (Codo arriba)

Por último, se lleva a cabo el mismo procedimiento, pero esta vez para la configuración de **codo abajo** (Figura 5.14), obteniendo las gráficas de los motores que se pueden observar en la Figura 5.15.

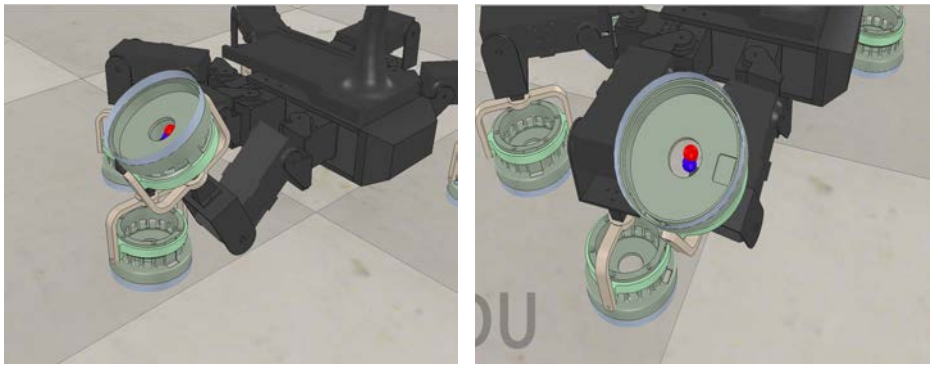


Figura 5.14: Posición objetivo de la pata 6 desde diferentes perspectivas (Codo abajo)

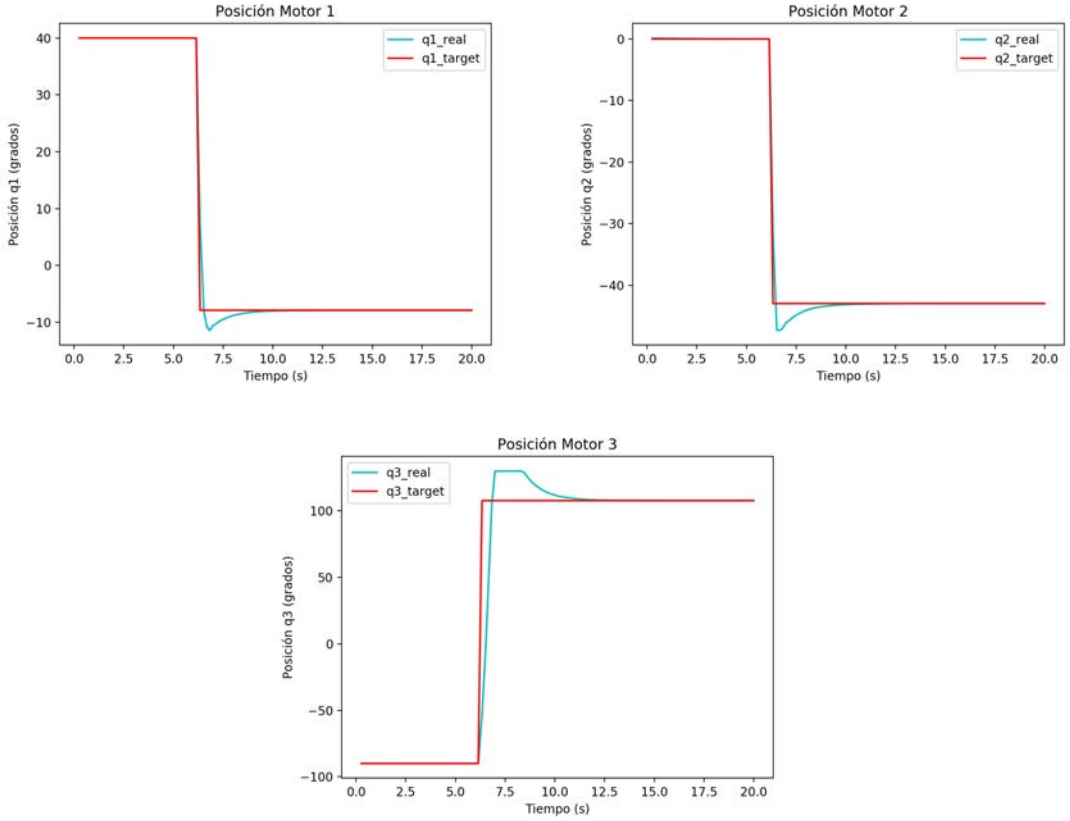


Figura 5.15: Gráficas de los motores de la pata 6 (Codo abajo)

En lás gráficas obtenidas en esta pata se puede apreciar como el comportamiento es muy similar al del resto de patas. Sin embargo, en la gráfica del motor 3 de la Figura 5.15 se observa como tarda más tiempo de lo normal en estabilizarse la posición real. Sin embargo, el controlador de posición del motor consigue finalmente minimizar el error y establecer la posición real en línea con la posición objetivo.

5.2. Resultados del algoritmo de movimiento

Para validar la correcta construcción del modelo y simular un modo de funcionamiento real, se ha desarrollado un **algoritmo básico de movimiento** que permite al robot desplazarse hacia delante a lo largo de una superficie plana. Debido a que se trata de un primer desarrollo, no se ha dotado de un control de trayectoria, por lo que simplemente se le envía a cada pata la posición que tiene que alcanzar para que se produzca el avance correspondiente.

Este algoritmo consta de los pasos que se detallan en el diagrama de la Figura 5.16. Estos pasos se repiten en bucle, primero para las patas pares y luego para las impares, constituyendo así un movimiento de tres en tres (*tripod gait*, Figura 5.17).

Para poder observar las posiciones que alcanzan los motores, se ha empleado el mismo programa en Python utilizado en la sección 5.1, que permite graficar la respuesta real del motor y superponer la gráfica con la posición objetivo. De esta manera, se obtienen las gráficas referentes a los motores de la pata 1 (Figura 5.18) a lo largo de una muestra de 20 segundos, durante los cuales el robot se encuentra

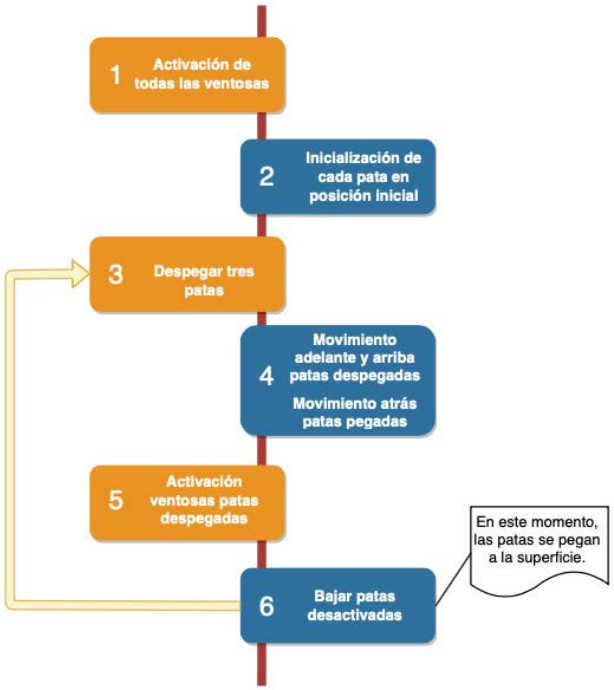


Figura 5.16: Algoritmo de movimiento por el suelo de ROMERIN

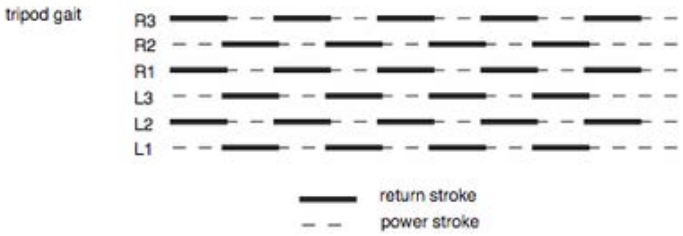


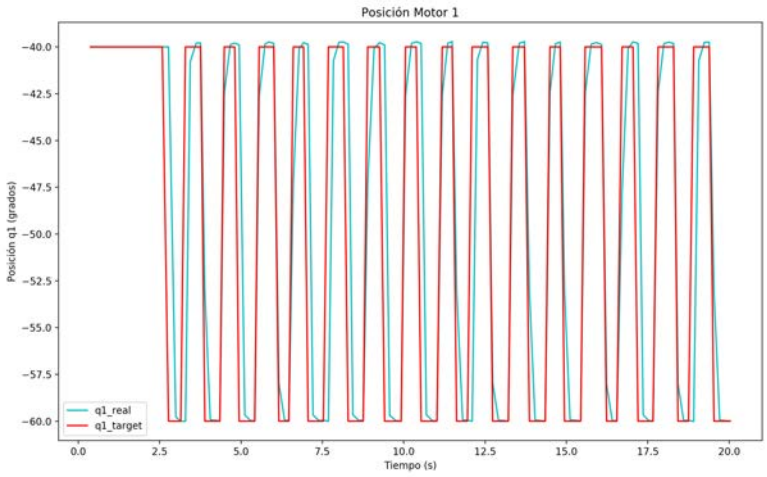
Figura 5.17: Algoritmo de movimiento *tripod gait*

andando por el suelo.

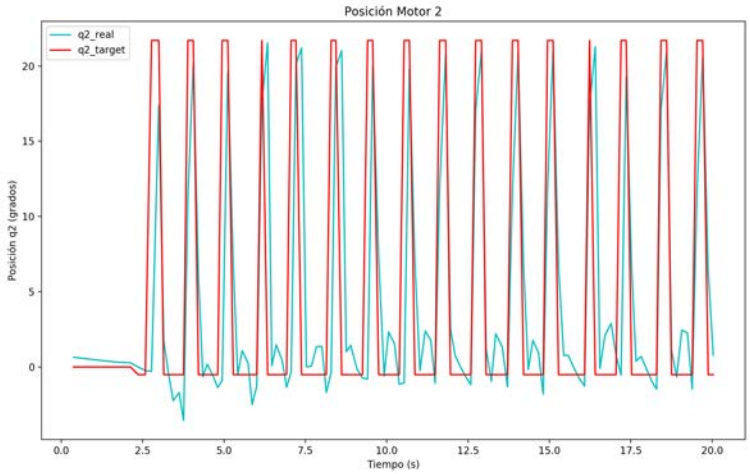
Estas gráficas nos permiten sacar algunas conclusiones. Como se puede observar, existen mayores desviaciones respecto a las gráficas obtenidas en la sección 5.1, especialmente en el motor 2 y el motor 3.

En la mayoría de ocasiones, los motores no alcanzan la posición objetivo, debido a que antes de que el motor pueda alcanzar la posición enviada y el controlador de posición del motor pueda minimizar el error, se envía la orden de volver a mover ese motor, por lo que comienza el movimiento hacia la siguiente posición, sin haber alcanzado correctamente la anterior.

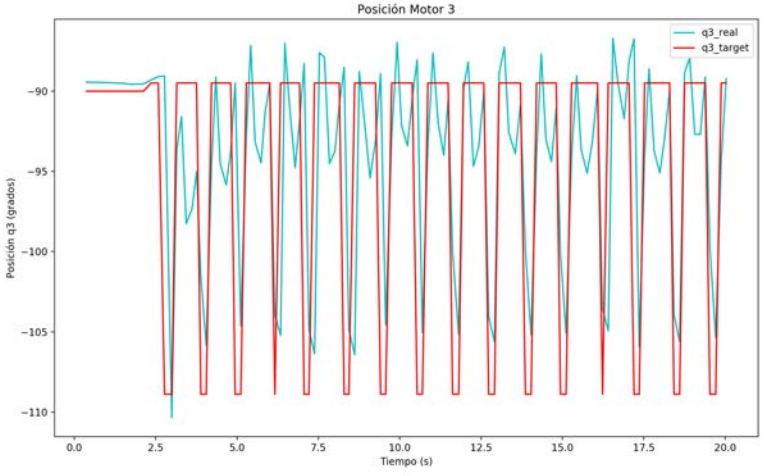
Por otro lado, también se observa que la posición real tiene una pequeña oscilación alrededor de la posición objetivo, lo cual puede ser consecuencia de que el sensor de proximidad de la pata activa la ventosa cuando detecta un objeto a menos de 7 milímetros de distancia, por lo que al establecerse la unión entre el suelo y la ventosa se produce una variación en la posición de los motores, especialmente en el motor 3.



(a) Motor 1



(b) Motor 2



(c) Motor 3

Figura 5.18: Gráficas de los motores de la pata 1 utilizando el algoritmo de movimiento

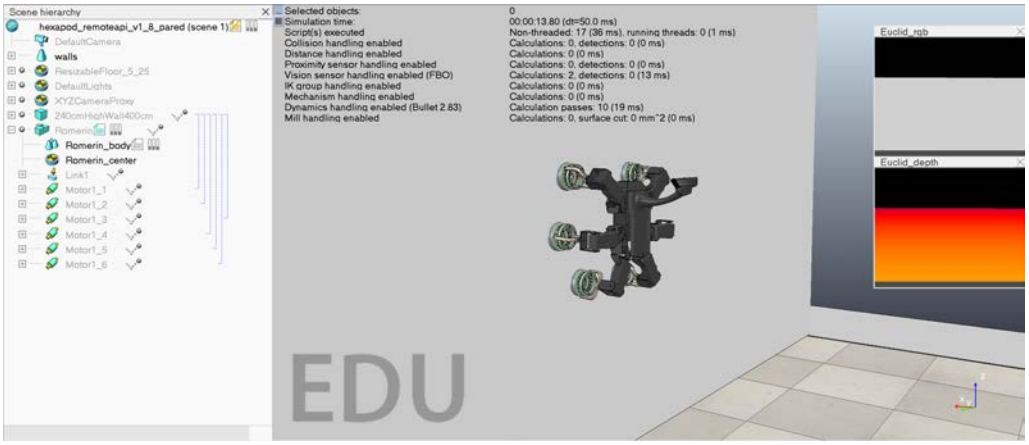


Figura 5.19: Robot ROMERIN pegado a la pared

5.3. Resultados de las ventosas

En última instancia, se han llevado a cabo una serie de pruebas para validar que la construcción de la ventosa dentro del simulador se ha realizado de manera correcta y que su respuesta dinámica es la esperada.

Para cada una de las ventosas se ha establecido una **fuerza de tracción máxima de 15 N** y una **fuerza de cizallamiento máxima de 30 N**, valores obtenidos de la ventosa real. Por tanto, cuando la ventosa se encuentra sometida a una fuerza de tracción o a una fuerza de cizallamiento mayor que los máximos establecidos, el enlace creado con la superficie ha de romperse.

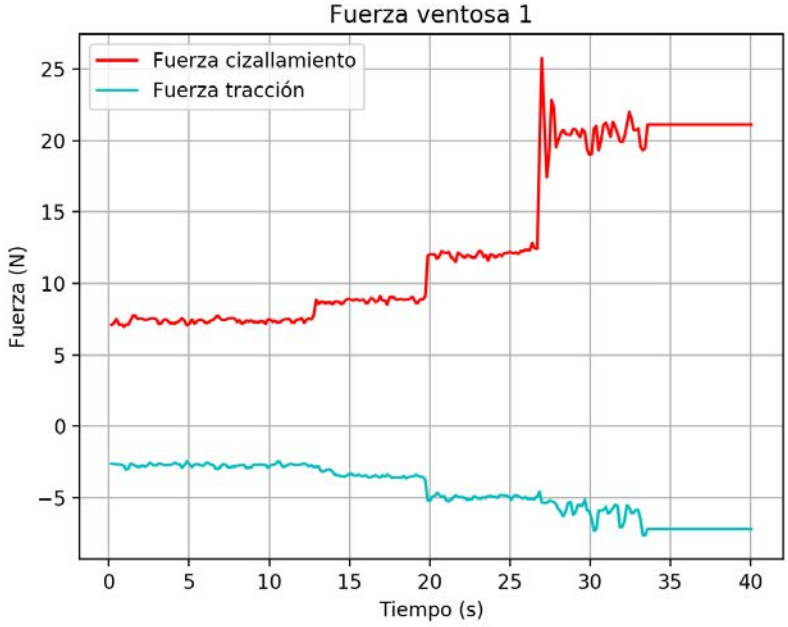
Con el objetivo de validar este comportamiento, se parte de una situación inicial (Figura 5.19) donde las seis ventosas del robot se encuentran activadas y pegadas a la pared, sosteniendo así todo el peso del cuerpo.

Tras unos 13 segundos en la posición inicial, se procede a despegar progresivamente las ventosas hasta que se rompa el enlace con la pared y el robot caiga al suelo.

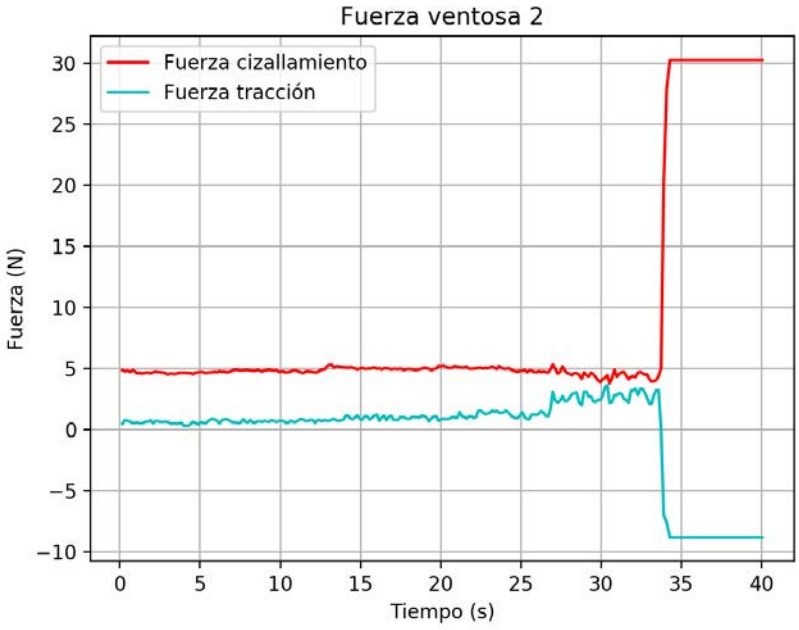
1. En primer lugar, se despega la ventosa de la pata 4.
2. Transcurridos 7 segundos, se procede a desactivar la ventosa de la pata 6.
3. Tras otros 7 segundos, se desactiva la ventosa de la pata 5.
4. Por último, pasados 7 segundos más, se desactiva la ventosa de la pata 1. En este momento, las ventosas de las patas 2 y 3 se encuentran sometidas a unas fuerzas mayores a las máximas establecidas, por lo que se despegan de la pared y el robot cae al suelo.

Para recoger todos los datos de las fuerzas de cada una de las ventosas y, posteriormente, poder obtener gráficas para analizar los resultados, se ha desarrollado un programa en Python, el cual se puede observar con más detalle en el fragmento de código 14.

De este modo, se obtienen las gráficas que se pueden observar en la Figura 5.19 donde, en función del tiempo, se representa la fuerza de tracción y la fuerza de cizallamiento que sufre cada una de las ventosas.

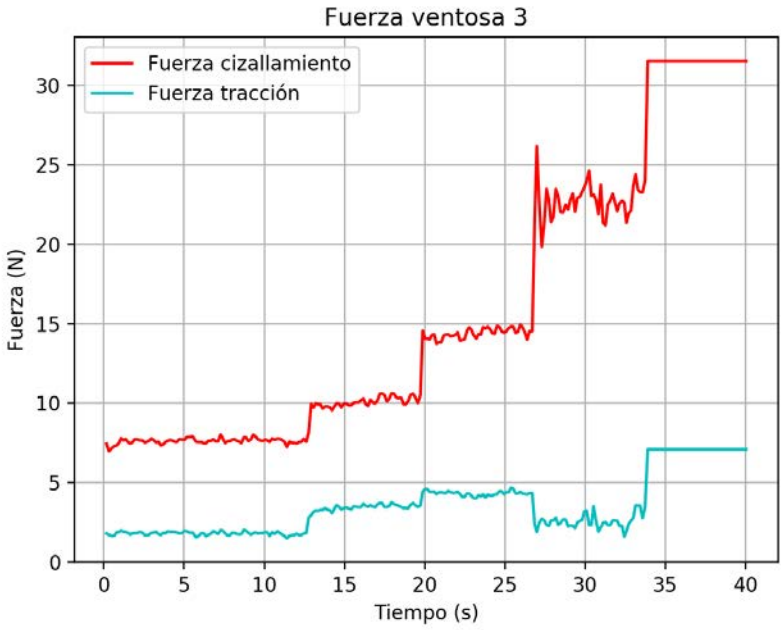


(a) Ventosa 1

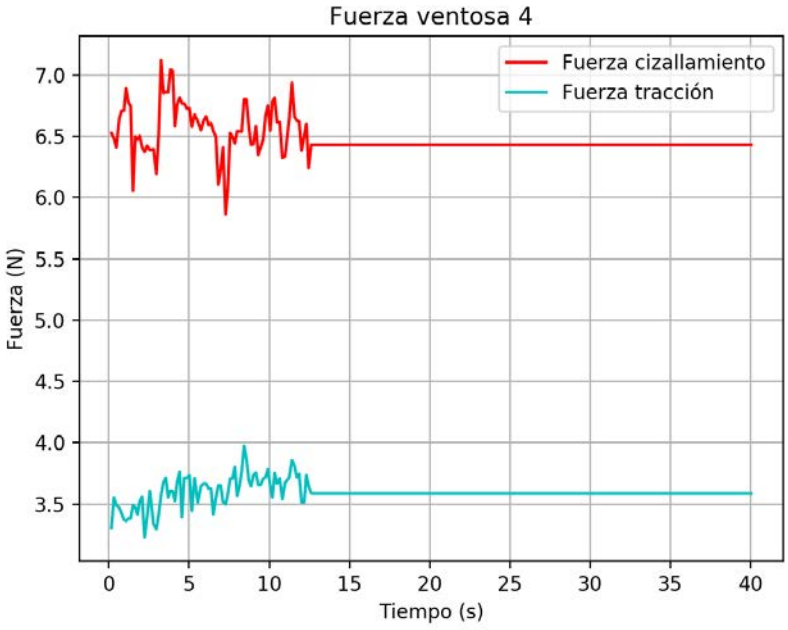


(b) Ventosa 2

Figura 5.19: Gráficas de las fuerzas de cada una de las ventosas de ROMERIN

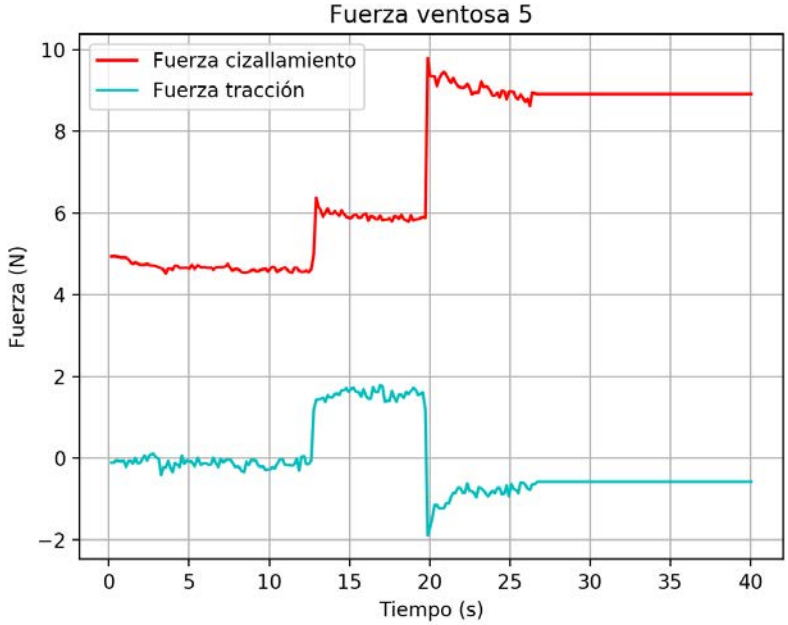


(c) Ventosa 3

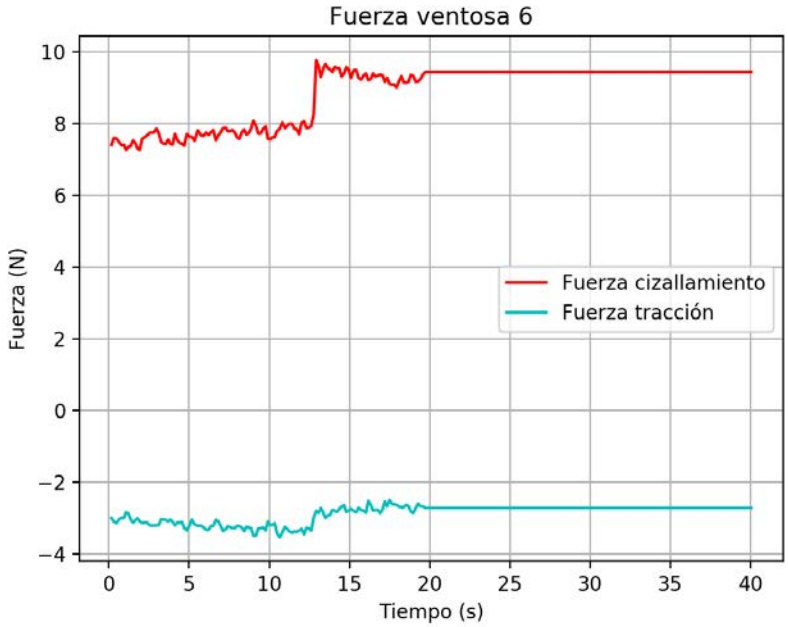


(d) Ventosa 4

Figura 5.19: Gráficas de las fuerzas de cada una de las ventosas de ROMERIN



(e) Ventosa 5



(f) Ventosa 6

Figura 5.19: Gráficas de las fuerzas de cada una de las ventosas de ROMERIN

De estas gráficas se pueden obtener una serie de resultados. En primer lugar, se aprecia como en los instantes de tiempo en los que se desactiva una ventosa, las restantes sufren un **aumento** en la **fuerza de cizallamiento**. Este comportamiento es totalmente lógico, ya que al perder un punto de sujección, el resto de ventosas tienen que aguantar un mayor porcentaje del peso del cuerpo.

Como se puede observar, las variaciones bruscas en las fuerzas de cada ventosa se

producen en los instantes donde se despega una ventosa, es decir, a los 13 segundos (ventosa 4), a los 20 segundos (ventosa 6), a los 27 segundos (ventosa 5) y, por último, a los 34 segundos (ventosa 1).

En el momento en el que se desactiva la ventosa 1, el robot se encuentra pegado a la pared únicamente por las **ventosas 2 y 3**. Sin embargo, como muestran las gráficas, en ese instante la **fuerza de cizallamiento** que se ejerce sobre estas ventosas es **mayor que 30 N**, por lo que las ventosas rompen el enlace y el robot se despega completamente de la pared.

Por lo tanto, se puede afirmar que las pruebas realizadas han resultado satisfactorias, pues se ha podido comprobar que el comportamiento dinámico de las ventosas es el esperado y se corresponde con el que tiene en la realidad.

Capítulo 6

Conclusiones

Se presentan a continuación las conclusiones del proyecto y los posibles desarrollos futuros que se pueden realizar sobre el mismo.

6.1. Conclusión

Una vez finalizado el proyecto se pueden analizar los resultados obtenidos a lo largo de las pruebas realizadas y obtener una serie de conclusiones.

En primer lugar, se ha conseguido **construir** con éxito el modelo del robot real dentro del simulador V-REP, respetando la estructura de los componentes y modelando su respuesta dinámica de manera muy próxima a la realidad.

Por otro lado, se han desarrollado los cálculos matemáticos relacionados con la **cinemática** del robot, así como su validación a través de una serie de pruebas donde se ha podido comprobar que el extremo del robot se mueve a la posición objetivo con un alto grado de precisión.

Otro objetivo fundamental que se ha cumplido es la implementación de la **API** del robot ROMERIN, de manera que se ha podido externalizar todos los cálculos referentes al robot en programas desarrollados en **Python**, los cuales se comunican con el simulador a través de la **API Remota**. Gracias a esto, se ha podido crear una interfaz común que permitirá controlar el robot de manera externa.

Por otra parte, gracias a las pruebas que se han llevado a cabo en el capítulo 5.3, se ha podido validar el **correcto funcionamiento de las ventosas**, de manera que se asegura que el comportamiento que tienen dentro del simulador cumple con los requisitos que posee la ventosa real.

Por último, para corroborar que el robot modelado en V-REP es capaz de moverse a lo largo de una superficie, se ha programado un algoritmo básico de movimiento el cual permite a ROMERIN moverse por el suelo en una trayectoria curvilínea y pegarse a él mediante el uso de las ventosas.

6.2. Desarrollos futuros

Un posible desarrollo que se puede realizar sobre este proyecto es corregir la trayectoria del robot cuando se mueve a lo largo de una superficie. Para esto, es necesario implementar un **sistema de control** que se realimente a partir de la posición actual de cada una de las patas, para así poder corregir las desviaciones en

la trayectoria. Además, este sistema de control debería ser capaz de detectar cuándo una ventosa está sometida a una fuerza de tracción o de cizallamiento cercana a los máximos establecidos, con el objetivo de poder asegurar que, en caso de despegarse, el resto de ventosas que se encuentran pegadas serán capaces de soportar todo el peso del robot.

Otra importante mejora que se puede realizar sobre este proyecto es la de obtener la nube de puntos que proporciona la *Intel Euclid*, la cual se ha modelado dentro de la escena de simulación. De esta manera, el robot podría ser capaz de evitar los obstáculos que se encuentre en su camino o de detectar una superficie con diferente grado de inclinación, para así poder realizar la transición de una superficie a otra.

Por último, otro desarrollo futuro sería la mejora y optimización de los componentes utilizados dentro del simulador, ya que muchos de ellos están compuestos por un gran número de triángulos y de vértices, lo que provoca que la escena consuma muchos recursos del ordenador y la simulación, en ocasiones, se ralentice.

Apéndice A

Configuración de la API Remota

A lo largo de este anexo se va a explicar los pasos que hay que realizar para poder utilizar correctamente la API Remota. Para ello, es necesario llevar a cabo dos configuraciones: una en el lado del **cliente** y otra en el lado del **servidor**.

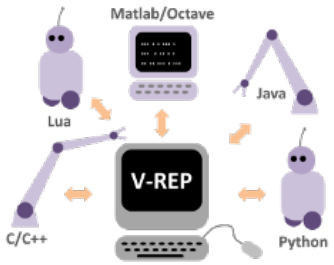


Figura A.1: Esquema API Remota

A.1. Configuración del cliente

En primer lugar, es necesario incluir algunos archivos en nuestro proyecto externo (en este caso, en Python) para poder llevar a cabo la comunicación con V-REP de manera satisfactoria.

Para que el proyecto en desarrollado Python funcione correctamente, es necesario los archivos que se indican a continuación, los cuales se encuentran dentro del directorio de instalación de V-REP, en la ruta *programming/remoteApiBindings/python*:

- vrep.py
- vrepConst.py
- remoteApi.dll, remoteApi.dylib or remoteApi.so (dependiendo de si la plataforma donde se utiliza es Windows, Mac OS o Linux, respectivamente).

Por otro lado, es posible que haya que definir la librería de la API Remota como compartida (haciendo uso de *remoteApiSharedLib.vcproj*), por lo que sería necesario definir como directivas de preprocesador **NON_MATLAB_PARSING** y **MAX_EXT_API_CONNECTIONS=255**.

Una vez realizada esta configuración, es necesario incluir la librería de V-REP (*import vrep*) en el *script* de Python donde se quiera realizar la comunicación con el simulador. A su vez, para permitir que funcione en el cliente, es necesario llamar a la funcion *vrep.simxStart* [1]. En [13] se encuentran todas las funciones compatibles con Python, fácilmente reconocibles por su prefijo *simx*.

En [9] se pueden encontrar los pasos necesarios para configurar el cliente de la API Remota en caso de utilizar otro lenguaje de programación diferente a Python (C/C++, Java, Matlab, Octave o Lua).

A.2. Configuración del servidor

En el otro extremo está situado el **servidor**, el cual se encuentra implementado en V-REP a través de un *plugin*. Este *plugin* de la API Remota puede ejecutar tantos servicios de servidores como se desee, de forma que cada uno de ellos se abrirá en un puerto diferente. Estos servicios pueden ser ejecutados de dos maneras:

- Durante el arraque de V-REP (servicio de la API Remota **continuo**). Con este método las funciones de la API Remota se estarán ejecutando siempre del lado del servidor, aunque la simulación no se esté ejecutando.
- Desde un *script* (servicio **temporal** de la API Remota). Este es el método que se utiliza a lo largo de este proyecto, pues permite al usuario establecer el comienzo y el final de la comunicación entre el programa externo y V-REP. Para poder **iniciar** o **terminar** un servicio de la API Remota es necesario utilizar una de las siguientes funciones de Lua (para más detalle sobre los argumentos de cada una de las funciones, se puede consultar [10]):
  - ***simRemoteApi.start***: función que comienza la ejecución de un servicio temporal de la API Remota en el puerto especificado.
  - ***simRemoteApi.stop***: se utiliza para terminar la ejecución de un servicio temporal de la API Remota que se está ejecutando en un determinado puerto.

[1] `http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctions.htm#simxPauseCommunication`

Apéndice B

Configuración de la escena de simulación

A lo largo de este anexo se va a detallar la configuración que hay que seguir para poder ejecutar correctamente tanto la **escena de V-REP**, como los **programas de Python** que vayan a comunicarse con el simulador.

En primer lugar, para poder establecer la comunicación entre el cliente (Python) y el servidor (V-REP), es necesario llevar a cabo la configuración que se explica en el Anexo A.

Tras esto, es necesario tener en cuenta las siguientes consideraciones:

- Para permitir la correcta comunicación entre la escena de V-REP y el cliente, es necesario incluir en el **script principal** del modelo de V-REP la instrucción ***simRemoteApi.start (clientID)***, donde *clientID* se corresponde con el identificador asociado al cliente.
- A su vez, también es necesario incluir la siguiente instrucción para que el cliente se pueda conectar correctamente al servidor.

```
1 clientID = vrep.simxStart('127.0.0.1', clientID, True, True, 5000, 5)
```

Teniendo en cuenta estas consideraciones, ejecutar el modelo en V-REP y comunicarse a través de la API remota resulta una tarea trivial. De manera breve se detallan a continuación los pasos habituales a seguir:

1. Abrir la escena en V-REP.
2. Abrir el proyecto en Python que contenga los archivos necesarios para poder ejecutar la API remota.
3. Teniendo en cuenta las consideraciones anteriormente mencionadas, comenzar la simulación dentro de V-REP.
4. Por último, ejecutar el *script* correspondiente desde un IDE (para este proyecto se ha utilizado PyCharm), tras lo cual el modelo comenzará a recibir las órdenes que se envíen desde ese *script* de Python.

Apéndice C

Código software

C.1. Código en Lua

Código 9: Child Script del Workspace de una pata de ROMERIN

```
function sysCall_init()

    -- Put your initialization code here
    dlg=sim.displayDialog('Workspace','Computing reachable points.&&nThis may take a few
    ↪  seconds...',
    sim.dlgstyle_message,false,nil)
    a=0

end
doCalculation=function()
    local divisions=5 -- the divisions used for each joint. The total nb of pts becomes
    ↪  (divisions+1)^7
    local checkCollision=true
    local usePointCloud=true -- if false, 3D spheres will be used
    local extractConvexHull=true
    local generateRandom=true -- Use random configurations instead of joint divisions
    local randomSteps=46000 -- Number of random configs to test


    local jointHandles={}
    local jointLimitLows={}
    local jointLimitRanges={}
    local initialJointPositions={}
    for i=1,3,1 do
        jointHandles[1]=sim.getObjectHandle('Motor1_3')
        jointHandles[2]=sim.getObjectHandle('Motor2_3')
        jointHandles[3]=sim.getObjectHandle('Motor3_3')
        local cyclic,interv=sim.getJointInterval(jointHandles[i])
        if cyclic then
            jointLimitLows[i]=-180*math.pi/180
            jointLimitRanges[i]=360*math.pi/180
        else
            jointLimitLows[i]=interv[1]
            jointLimitRanges[i]=interv[2]
        end
        initialJointPositions[i]=sim.getJointPosition(jointHandles[i])
    end
    local tip=sim.getObjectHandle('Punto_extremo3')
    local base=sim.getObjectHandle('hexapod')
    local robotCollection=sim.getCollectionHandle('workspace')
    local points={}
    local normals={}

    if not usePointCloud then
        pointContainer=sim.addDrawingObject(sim.drawing_spherepoints,0.01,0.01,base,99999,{1,1,1})
```

```
    end

    if generateRandom then
        for cnt=1,randomSteps,1 do
            local p1=jointLimitLows[1]+math.random()*jointLimitRanges[1]
            sim.setJointPosition(jointHandles[1],p1)
            local p2=jointLimitLows[2]+math.random()*jointLimitRanges[2]
            sim.setJointPosition(jointHandles[2],p2)
            local p3=jointLimitLows[3]+math.random()*jointLimitRanges[3]
            sim.setJointPosition(jointHandles[3],p3)

            local colliding=false
            local matrix=sim.getObjectMatrix(tip,-1)
            -- local pos=sim.getObjectPosition(tip,-1)
            if checkCollision then
                if sim.checkCollision(robotCollection,sim.handle_all)~=0 then
                    colliding=true
                end
            end
            if not colliding then
                points[#points+1]=matrix[4]
                points[#points+1]=matrix[8]
                points[#points+1]=matrix[12]
                if usePointCloud then
                    normals[#normals+1]=matrix[3]
                    normals[#normals+1]=matrix[7]
                    normals[#normals+1]=matrix[11]
                else
                    sim.addDrawingObjectItem(pointContainer,{matrix[4],matrix[8],matrix[12]})
                end
            end
        end
    else
        for i1=1,divisions+1,1 do
            local p1=jointLimitLows[1]+(i1-1)*jointLimitRanges[1]/divisions
            sim.setJointPosition(jointHandles[1],p1)
            for i2=1,divisions+1,1 do
                local p2=jointLimitLows[2]+(i2-1)*jointLimitRanges[2]/divisions
                sim.setJointPosition(jointHandles[2],p2)
                for i3=1,divisions+1,1 do
                    local p3=jointLimitLows[3]+(i3-1)*jointLimitRanges[3]/divisions
                    sim.setJointPosition(jointHandles[3],p3)


                    local colliding=false
                    local matrix=sim.getObjectMatrix(tip,-1)
                    -- local pos=sim.getObjectPosition(tip,-1)
                    if checkCollision then
                    if sim.checkCollision(robotCollection,sim.handle_all)~=0 then
                          colliding=true
                    end
                    end
                    if not colliding then
                       points[#points+1]=matrix[4]
                       points[#points+1]=matrix[8]
                       points[#points+1]=matrix[12]
                    if usePointCloud then
                       normals[#normals+1]=matrix[3]
                       normals[#normals+1]=matrix[7]
                       normals[#normals+1]=matrix[11]
                    else
                       sim.addDrawingObjectItem(pointContainer,{matrix[4],matrix[8],matrix[12]})
                    end
                    end
                end
            end
        end
    end


```

```
    for i=1,3,1 do
        sim.setJointPosition(jointHandles[i],initialJointPositions[i])
    end

    if usePointCloud then
        sim.addPointCloud(0,255,base,0,5,points,{255,255,255,0,0,0,64,64,64,0,0,0},nil,normals)
    end

    if extractConvexHull then
        local vertices,indices=sim.getQHull(points)
        local shape=sim.createMeshShape(3,0,vertices,indices)
        sim.reorientShapeBoundingBox(shape,-1)
        sim.setShapeColor(shape,nil,0,{1,0,1})
        sim.setShapeColor(shape,nil,4,{0.2})
        sim.setObjectName(shape,'ConvexHull')
    end
end



function sysCall_actuation()
    -- Put your main ACTUATION code here
    a=a+1
    if a==2 then
        doCalculation()
        sim.endDialog(dlg)
    end
end


function sysCall_sensing()
    -- Put your main SENSING code here
end


function sysCall_cleanup()
    -- Put some restoration code here
end
```

C.2. Código en Python

Código 10: Clase RomerinAPI

```
import numpy as np
import Leg

class RomerinAPI:

    def __init__(self, motor11, motor21, motor31, ventosa1, motor12, motor22, motor32, ventosa2,
    ↪  motor13, motor23, motor33, ventosa3, motor14, motor24, motor34, ventosa4, motor15, motor25,
    ↪  motor35, ventosa5, motor16, motor26, motor36, ventosa6, clientID):

        self.leg1 = Leg.Pata(motor11, motor21, motor31, ventosa1, Leg.PosCux.CuxHaciaX, -40,
        ↪  clientID)
        self.leg2 = Leg.Pata(motor12, motor22, motor32, ventosa2, Leg.PosCux.CuxOpX, 0, clientID)
        self.leg3 = Leg.Pata(motor13, motor23, motor33, ventosa3, Leg.PosCux.CuxOpX, -30, clientID)
        self.leg4 = Leg.Pata(motor14, motor24, motor34, ventosa4, Leg.PosCux.CuxHaciaX, 30,
        ↪  clientID)
        self.leg5 = Leg.Pata(motor15, motor25, motor35, ventosa5, Leg.PosCux.CuxHaciaX, 0,
        ↪  clientID)
        self.leg6 = Leg.Pata(motor16, motor26, motor36, ventosa6, Leg.PosCux.CuxOpX, 40, clientID)

        # Homogeneous transformation matrix

```

```
        self.MTH1 = [[1, 0, 0, 28.962], [0, 1, 0, 75.060], [0, 0, 1, 0], [0, 0, 0, 1]]
        self.MTH2 = [[1, 0, 0, -40.220], [0, 1, 0, 75.060], [0, 0, 1, 0], [0, 0, 0, 1]]
        self.MTH3 = [[0, -1, 0, -97.347], [1, 0, 0, 54.413], [0, 0, 1, 0], [0, 0, 0, 1]]
        self.MTH4 = [[0, -1, 0, -97.347], [1, 0, 0, -54.413], [0, 0, 1, 0], [0, 0, 0, 1]]
        self.MTH5 = [[-1, 0, 0, -40.220], [0, -1, 0, -75.060], [0, 0, 1, 0], [0, 0, 0, 1]]
        self.MTH6 = [[-1, 0, 0, 28.962], [0, -1, 0, -75.060], [0, 0, 1, 0], [0, 0, 0, 1]]


    # This function sets world coordinates for each leg. If OK, returns 1. If not, returns -1.

    def moveLeg (self, pata, Pxy, Pz):

        posWorld = np.transpose(np.array([Pxy[0], Pxy[1], Pz, 1])) #Translation vector

        if pata == 1:
            posLeg = np.dot(np.linalg.inv(self.MTH1), posWorld)
            self.leg1.setPosition(posLeg)

        elif pata == 2:
            posLeg = np.dot(np.linalg.inv(self.MTH2), posWorld)
            self.leg2.setPosition(posLeg)

        elif pata == 3:
            posLeg = np.dot(np.linalg.inv(self.MTH3), posWorld)
            self.leg3.setPosition(posLeg)

        elif pata == 4:
            posLeg = np.dot(np.linalg.inv(self.MTH4), posWorld)
            self.leg4.setPosition(posLeg)

        elif pata == 5:
            posLeg = np.dot(np.linalg.inv(self.MTH5), posWorld)
            self.leg5.setPosition(posLeg)

        elif pata == 6:
            posLeg = np.dot(np.linalg.inv(self.MTH6), posWorld)
            self.leg6.setPosition(posLeg)


    # This function gets world coordinates of each leg

    def getPosition (self, pata):

        if pata == 1:
            posLeg = self.leg1.getPosition()
            posLeg.append(1)
            posWorld = np.dot(self.MTH1, posLeg)

        elif pata == 2:
            posLeg = self.leg2.getPosition()
            posLeg.append(1)
            posWorld = np.dot(self.MTH2, posLeg)

        elif pata == 3:
            posLeg = self.leg3.getPosition()
            posLeg.append(1)
            posWorld = np.dot(self.MTH3, posLeg)

        elif pata == 4:
            posLeg = self.leg4.getPosition()
            posLeg.append(1)
            posWorld = np.dot(self.MTH4, posLeg)

        elif pata == 5:
            posLeg = self.leg5.getPosition()
            posLeg.append(1)
            posWorld = np.dot(self.MTH5, posLeg)

        elif pata == 6:
            posLeg = self.leg6.getPosition()
```

```
87              posLeg.append(1)
88              posWorld = np.dot(self.MTH6, posLeg)
89
90          return posWorld
91
92
93      # This function enables o disables the suction pads
94
95      def setGripper (self, pata, active):
96
97          if pata == 1:
98              self.leg1.setGripper(active)
99
100         elif pata == 2:
101             self.leg2.setGripper(active)
102
103         elif pata == 3:
104             self.leg3.setGripper(active)
105
106         elif pata == 4:
107             self.leg4.setGripper(active)
108
109         elif pata == 5:
110             self.leg5.setGripper(active)
111
112         elif pata == 6:
113             self.leg6.setGripper(active)
114
115
116     # This function gets the state of the suction pad (If it is enables, suction_pad = 1. If not,
    ↪  suction_pad = 0)
117
118     def getGripper (self, pata):
119
120         if pata == 1:
121             suction_pad = self.leg1.getGripper()
122
123         elif pata == 2:
124             suction_pad = self.leg2.getGripper()
125
126         elif pata == 3:
127             suction_pad = self.leg3.getGripper()
128
129         elif pata == 4:
130             suction_pad = self.leg4.getGripper()
131
132         elif pata == 5:
133             suction_pad = self.leg5.getGripper()
134
135         elif pata == 6:
136             suction_pad = self.leg6.getGripper()
137
138         return suction_pad
139
140
141     # This function sets the joint angles of the selected leg
142
143     def setAngles (self, pata, angles):
144
145         if pata == 1:
146             self.leg1.setAngles(angles)
147
148         if pata == 2:
149             self.leg2.setAngles(angles)
150
151         if pata == 3:
152             self.leg3.setAngles(angles)
153
154         if pata == 4:
155             self.leg4.setAngles(angles)
```

```
156
157         if pata == 5:
158             self.leg5.setAngles(angles)
159
160         if pata == 6:
161             self.leg6.setAngles(angles)
```

Código 11: Clase Leg

```
1   import math
2   import vrep
3   import Joint as motor
4   from enum import Enum
5
6   class PosCux(Enum):
7       CuxHaciaX = 1
8       CuxOpX = 2
9
10
11  class ElbowPos(Enum):
12     ElbUp = 1
13     ElbDown = 2
14
15  class Pata:
16
17      # Constants common to all legs (mm)
18
19      LC = 45.718
20      AngHaciaX = 1.19677
21      AngOpX = 1.94482
22      LCz = -18.208
23      LF = 78.89
24      LT = 67.788 + 46.71
25
26      # Function which set motors handle for each leg
27
28      def __init__(self, name0, name1, name2, ventosa, Cux, m0pos, clientID):
29
30          self.clientID = clientID
31
32          self.m0 = motor.Joint()
33          self.m1 = motor.Joint()
34          self.m2 = motor.Joint()
35          self.m3 = motor.Joint()
36
37          self.m0.setJointHandle(name0)
38          self.m1.setJointHandle(name1)
39          self.m2.setJointHandle(name2)
40
41
42          self.m0.setClientID(clientID)
43          self.m1.setClientID(clientID)
44          self.m2.setClientID(clientID)
45
46
47          self.m0.position = m0pos
48          self.m1.position = 0
49          self.m2.position = -90
50
51          self.m0.setJointPosition(self.m0.position)
52
53          self.vectorAngles = []
54
55          self.vectorAngles.append(self.m0.position)
56          self.vectorAngles.append(self.m1.position)
57          self.vectorAngles.append(self.m2.position)
58
```

```
59          self.vectorPosition = []
60
61          self.ventosa = ventosa
62
63          self.ElbowDefaultPos = ElbowPos.ElbUp
64
65          if Cux == PosCux.CuxHaciaX:
66              self.AngCx = Pata.AngHaciaX
67          if Cux == PosCux.CuxOpX:
68              self.AngCx = Pata.AngOpX
69
70
71      # FORWARD KINEMATICS
72
73      def getFK (self, angles):
74
75          q1 = abs(angles[0]) * math.pi/180
76          q2 = abs(angles[1]) * math.pi/180
77          q3 = abs(angles[2]) * math.pi/180
78
79          x = Pata.LC * math.cos(self.AngCx + q1) + Pata.LF * math.cos(q2) * math.sin(q1) + Pata.LT *
            ↪  math.cos(q2 + q3) * math.sin(q1)
80          y = Pata.LC * math.sin(self.AngCx + q1) + Pata.LF * math.cos(q2) * math.cos(q1) + Pata.LT *
            ↪  math.cos(q2 + q3) * math.cos(q1)
81          z = Pata.LCz + Pata.LF * math.sin(q2) + Pata.LT * math.sin(q2 + q3)
82
83          self.vectorPosition = [x, y, z]
84
85      # INVERSE KINEMATICS
86
87      def getIK_angles (self, position, ElbPos):
88          x = position[0]
89          y = position[1]
90          z = position[2]
91
92          LCx = Pata.LC * math.cos(self.AngCx)
93          LCy = Pata.LC * math.sin(self.AngCx)
94
95          h = math.sqrt(x ** 2 + y ** 2)
96
97          LP_aux = math.sqrt(h ** 2 - LCx ** 2)
98          LP = LP_aux - LCy
99          HF = math.sqrt(LP ** 2 + (z + Pata.LCz) ** 2)
100
101         # q1 calculations
102         q1_aux1 = math.atan2(-x, y)
103         q1_aux2 = math.atan2(LCx, LP_aux)
104         q1 = math.degrees(q1_aux1 + q1_aux2)
105
106         # q3 calculations
107         cosq3 = (HF ** 2 - Pata.LF ** 2 - Pata.LT ** 2) / (2 * Pata.LF * Pata.LT) #Complementary
            ↪  cosB = cosq3
108
109         if math.fabs(cosq3) > 1:
110             print("This point is unachievable")
111             return False # Unachievable
112
113         if ElbPos == ElbowPos.ElbUp:
114             senq3 = -math.sqrt(1 - (cosq3 ** 2))
115
116         elif ElbPos == ElbowPos.ElbDown:
117             senq3 = math.sqrt(1 - (cosq3 ** 2))
118
119         q3_aux = math.atan2(senq3, cosq3)
120         q3 = math.degrees(q3_aux)
121
122         # q2 calculations
123         alpha = math.atan2((Pata.LT * math.sin(q3_aux)), (Pata.LF + Pata.LT * math.cos(q3_aux)))
124         beta = math.atan2(z + Pata.LCz, LP)
125         q2 = math.degrees(beta - alpha)
```

```
126
127         self.vectorAngles = [q1, q2, q3]
128
129
130     # This function sets the position of the leg according the given point
131
132     def setPosition (self, posLeg):
133         self.vectorPosition = posLeg
134         self.getIK_angles(self.vectorPosition, self.ElbowDefaultPos)
135
136         self.m0.setJointPosition(self.vectorAngles[0])
137         self.m2.setJointPosition(self.vectorAngles[2])
138         self.m1.setJointPosition(self.vectorAngles[1])
139
140     # This function gets position of the 3 joints of a leg
141
142     def getPosition (self):
143         self.m0.getJointPosition()
144         self.m1.getJointPosition()
145         self.m2.getJointPosition()
146
147         self.vectorAngles = [self.m0.position, self.m1.position, self.m2.position]
148
149         self.getFK(self.vectorAngles)
150
151         return self.vectorPosition
152
153     # This function sets the position of the leg according the given angles
154
155     def setAngles (self, angles):
156         self.vectorAngles = angles
157
158         self.m0.setJointPosition(self.vectorAngles[0])
159         self.m1.setJointPosition(self.vectorAngles[1])
160         self.m2.setJointPosition(self.vectorAngles[2])
161
162     # This function sets the velocity of each motor
163
164     def setVelocity (self, v0, v1, v2):
165         if v0 != -1:
166             self.m0.setJointVelocity(v0)
167         if v1 != -1:
168             self.m1.setJointVelocity(v1)
169         if v2 != -1:
170             self.m2.setJointVelocity(v2)
171
172     # This function enables or disables the suction pads
173
174     def setGripper (self, active):
175         vrep.simxSetIntegerSignal(self.clientID, self.ventosa, active, vrep.simx_opmode_oneshot)
176
177     # This function gets the state of the suction pad (If it is enables or not)
178
179     def getGripper(self):
180         err, gripperState = vrep.simxGetIntegerSignal(self.clientID, self.ventosa,
            ↪  vrep.simx_opmode_oneshot)
181         return gripperState
```

Código 12: Clase Joint

```
1  import math
2  import vrep
3
4  class Joint:
5
6      # Attributes
7      handle = 0
```

```
    position = 0
    velocity = 0
    clientID = 0

    def setClientID(self, clientID):
        self.clientID = clientID

    def setJointHandle(self, name):
        r, self.handle = vrep.simxGetObjectHandle(self.clientID, name,
        ↪  vrep.simx_opmode_oneshot_wait)

    def setJointPosition(self, pos):
        self.position = pos
        vrep.simxSetJointTargetPosition(self.clientID, self.handle, pos * math.pi / 180,
        ↪  vrep.simx_opmode_oneshot)

    def getJointPosition (self):
        ret, position = vrep.simxGetJointPosition(self.clientID, self.handle,
        ↪  vrep.simx_opmode_streaming)
        self.position = math.degrees(position)

    def setJointVelocity(self, vel):
        self.velocity = vel
        ret = vrep.simxSetObjectFloatParameter(self.clientID, self.handle,
        ↪  vrep.sim_jointfloatparam_upper_limit, self.velocity,
                                               vrep.simx_opmode_oneshot)
```

Código 13: Graficar posiciones de los motores

```
import vrep
import math
import time
import matplotlib.pyplot as python_plot
import sys

vrep.simxFinish(-1)
clientID = vrep.simxStart('127.0.0.1', 20000, True, True, 5000, 5)
vrep.simxSynchronous(clientID, True)

if clientID != -1:
    print('Connected with VREP')
    print("")

    pos1 = []
    target1 = []
    pos2 = []
    target2 = []
    pos3 = []
    target3 = []
    t = []

    t0 = time.time()
    while (vrep.simxGetConnectionId(clientID)) == 1:

        res1, retInts1, info, retStrings1, retBuffer1 = vrep.simxCallScriptFunction(clientID,
        ↪  "Romerin_body", vrep.sim_scripttype_childscript, "info_patas", [], [], [], bytearray(),
        ↪  vrep.simx_opmode_blocking)

        if (info == [] or (time.time()-t0) > 20):
            break
        vrep.simxSynchronousTrigger(clientID)


        target1.append(info[0] * 180 / math.pi)
        target2.append(info[1] * 180 / math.pi)
        target3.append(info[2] * 180 / math.pi)
        pos1.append(info[3] *180/math.pi)
```

```
        pos2.append(info[4] * 180 / math.pi)
        pos3.append(info[5] * 180 / math.pi)
        vrep.simxSynchronousTrigger(clientID)
        t.append(time.time()-t0)


    # Plot the data q1
    python_plot.figure(1)

    python_plot.plot(t, pos1, 'c-', label='q1_real')
    python_plot.plot(t, target1, 'r-', label='q1_target')

    python_plot.legend()
    python_plot.xlabel('Tiempo (s)')
    python_plot.ylabel('Posicion q1 (grados)')
    python_plot.title('Posicion Motor 1')


    # Show the plot
    python_plot.show()



    # Plot the data q2
    python_plot.figure(1)

    python_plot.plot(t, pos2, 'c-', label='q2_real')
    python_plot.plot(t, target2, 'r-', label='q2_target')

    python_plot.legend()
    python_plot.xlabel('Tiempo (s)')
    python_plot.ylabel('Posicion q2 (grados)')
    python_plot.title('Posicion Motor 2')

    # Show the plot
    python_plot.show()



    # Plot the data q3
    python_plot.figure(1)

    python_plot.plot(t, pos3, 'c-', label='q3_real')
    python_plot.plot(t, target3, 'r-', label='q3_target')

    python_plot.legend()
    python_plot.xlabel('Tiempo (s)')
    python_plot.ylabel('Posicion q3 (grados)')
    python_plot.title('Posicion Motor 3')

    # Show the plot
    python_plot.show()

    vrep.simxFinish(-1)

    f = open('punto1.txt', 'w+')
    f.write("target1:" +'\n' + " ".join(str(x) for x in target1) +'\n')
    f.write("target2:" +'\n' + " ".join(str(x) for x in target2) +'\n')
    f.write("target3:" +'\n' + " ".join(str(x) for x in target3) +'\n')
    f.write("pos1:" +'\n' + " ".join(str(x) for x in pos1) +'\n')
    f.write("pos2:" +'\n' + " ".join(str(x) for x in pos2) +'\n')
    f.write("pos3:" +'\n' + " ".join(str(x) for x in pos3) +'\n')
    f.close()

else:
    sys.exit('Error: unable to connect')
    vrep.simxFinish(clientID)
```

Código 14: Graficar fuerzas de las ventosas

```
import vrep
import time
import matplotlib.pyplot as python_plot
import sys
import math

vrep.simxFinish(-1)
clientID = vrep.simxStart('127.0.0.1', 60000, True, True, 5000, 5)
vrep.simxSynchronous(clientID, True)

if clientID != -1:

    print('Connected with VREP')
    print("")

    FuerzaX1 = []
    FuerzaY1 = []
    FuerzaZ1 = []
    FuerzaXY1 = []

    FuerzaX2 = []
    FuerzaY2 = []
    FuerzaZ2 = []
    FuerzaXY2 = []

    FuerzaX3 = []
    FuerzaY3 = []
    FuerzaZ3 = []
    FuerzaXY3 = []

    FuerzaX4 = []
    FuerzaY4 = []
    FuerzaZ4 = []
    FuerzaXY4 = []

    FuerzaX5 = []
    FuerzaY5 = []
    FuerzaZ5 = []
    FuerzaXY5 = []

    FuerzaX6 = []
    FuerzaY6 = []
    FuerzaZ6 = []
    FuerzaXY6 = []

    t = []
    t0 = time.time()

    while (vrep.simxGetConnectionId(clientID)) == 1:

        res1, retInts1, infovent, retStrings1, retBuffer1 = vrep.simxCallScriptFunction(clientID,
        ↪  "Romerin_body", vrep.sim_scripttype_childscript, "info_ventosas", [], [], [],
        ↪  bytearray(), vrep.simx_opmode_blocking)
        if (infovent == [] or (time.time()-t0) > 40):
            break
        vrep.simxSynchronousTrigger(clientID)

        # Suction pad 1
        FuerzaX1.append(infovent[0])
        FuerzaY1.append(infovent[1])
        FuerzaZ1.append(infovent[2])
        FuerzaXY1.append(math.sqrt(infovent[0] ** 2 + infovent[1] ** 2))

        # Suction pad 2
        FuerzaX2.append(infovent[3])
        FuerzaY2.append(infovent[4])
        FuerzaZ2.append(infovent[5])
        FuerzaXY2.append(math.sqrt(infovent[3] ** 2 + infovent[4] ** 2))
```

```
67
68          # Suction pad 3
69          FuerzaX3.append(infovent[6])
70          FuerzaY3.append(infovent[7])
71          FuerzaZ3.append(infovent[8])
72          FuerzaXY3.append(math.sqrt(infovent[6] ** 2 + infovent[7] ** 2))
73
74          # Suction pad 4
75          FuerzaX4.append(infovent[9])
76          FuerzaY4.append(infovent[10])
77          FuerzaZ4.append(infovent[11])
78          FuerzaXY4.append(math.sqrt(infovent[9] ** 2 + infovent[10] ** 2))
79
80          # Suction pad 5
81          FuerzaX5.append(infovent[12])
82          FuerzaY5.append(infovent[13])
83          FuerzaZ5.append(infovent[14])
84          FuerzaXY5.append(math.sqrt(infovent[12] ** 2 + infovent[13] ** 2))
85
86          # Suction pad 6
87          FuerzaX6.append(infovent[15])
88          FuerzaY6.append(infovent[16])
89          FuerzaZ6.append(infovent[17])
90          FuerzaXY6.append(math.sqrt(infovent[15] ** 2 + infovent[16] ** 2))
91
92          vrep.simxSynchronousTrigger(clientID)
93          t.append(time.time()-t0)
94
95
96      # Plot the data suction pad 1
97      python_plot.figure(1)
98
99      python_plot.plot(t, FuerzaXY1, 'r-', label='Fuerza cizallamiento')
100     python_plot.plot(t, FuerzaZ1, 'c-', label='Fuerza tracción')
101
102     python_plot.legend()
103     python_plot.xlabel('Tiempo (s)')
104     python_plot.ylabel('Fuerza (N)')
105     python_plot.title('Fuerza ventosa 1')
106
107     python_plot.grid(True)
108     python_plot.show()
109
110     # Plot the data suction pad 2
111     python_plot.figure(1)
112
113     python_plot.plot(t, FuerzaXY2, 'r-', label='Fuerza cizallamiento')
114     python_plot.plot(t, FuerzaZ2, 'c-', label='Fuerza tracción')
115
116     python_plot.legend()
117     python_plot.xlabel('Tiempo (s)')
118     python_plot.ylabel('Fuerza (N)')
119     python_plot.title('Fuerza ventosa 2')
120
121     python_plot.grid(True)
122     python_plot.show()
123
124     # Plot the data suction pad 3
125     python_plot.figure(1)
126
127     python_plot.plot(t, FuerzaXY3, 'r-', label='Fuerza cizallamiento')
128     python_plot.plot(t, FuerzaZ3, 'c-', label='Fuerza tracción')
129
130     python_plot.legend()
131     python_plot.xlabel('Tiempo (s)')
132     python_plot.ylabel('Fuerza (N)')
133     python_plot.title('Fuerza ventosa 3')
134
135     python_plot.grid(True)
136     python_plot.show()
```

```
137
138     # Plot the data suction pad 4
139     python_plot.figure(1)
140
141     python_plot.plot(t, FuerzaXY4, 'r-', label='Fuerza cizallamiento')
142     python_plot.plot(t, FuerzaZ4, 'c-', label='Fuerza tracción')
143
144     python_plot.legend()
145     python_plot.xlabel('Tiempo (s)')
146     python_plot.ylabel('Fuerza (N)')
147     python_plot.title('Fuerza ventosa 4')
148
149     python_plot.grid(True)
150     python_plot.show()
151
152     # Plot the data suction pad 5
153     python_plot.figure(1)
154
155     python_plot.plot(t, FuerzaXY5, 'r-', label='Fuerza cizallamiento')
156     python_plot.plot(t, FuerzaZ5, 'c-', label='Fuerza tracción')
157
158     python_plot.legend()
159     python_plot.xlabel('Tiempo (s)')
160     python_plot.ylabel('Fuerza (N)')
161     python_plot.title('Fuerza ventosa 5')
162
163     python_plot.grid(True)
164     python_plot.show()
165
166     # Plot the data suction pad 6
167     python_plot.figure(1)
168
169     python_plot.plot(t, FuerzaXY6, 'r-', label='Fuerza cizallamiento')
170     python_plot.plot(t, FuerzaZ6, 'c-', label='Fuerza tracción')
171
172     python_plot.legend()
173     python_plot.xlabel('Tiempo (s)')
174     python_plot.ylabel('Fuerza (N)')
175     python_plot.title('Fuerza ventosa 6')
176
177     python_plot.grid(True)
178     python_plot.show()
179
180     vrep.simxFinish(-1)
181
182     f = open('suctionPad_Forces.txt', 'w+')
183     f.write("FuerzaX1:" + '\n' + ", ".join(str(x) for x in FuerzaX1) + '\n')
184     f.write("FuerzaY1:" + '\n' + ", ".join(str(x) for x in FuerzaY1) + '\n')
185     f.write("FuerzaZ1:" + '\n' + ", ".join(str(x) for x in FuerzaZ1) + '\n')
186     f.write("FuerzaXY1:" + '\n' + ", ".join(str(x) for x in FuerzaXY1) + '\n')
187     f.write("FuerzaX2:" + '\n' + ", ".join(str(x) for x in FuerzaX2) + '\n')
188     f.write("FuerzaY2:" + '\n' + ", ".join(str(x) for x in FuerzaY2) + '\n')
189     f.write("FuerzaZ2:" + '\n' + ", ".join(str(x) for x in FuerzaZ2) + '\n')
190     f.write("FuerzaXY2:" + '\n' + ", ".join(str(x) for x in FuerzaXY2) + '\n')
191     f.write("FuerzaX3:" + '\n' + ", ".join(str(x) for x in FuerzaX3) + '\n')
192     f.write("FuerzaY3:" + '\n' + ", ".join(str(x) for x in FuerzaY3) + '\n')
193     f.write("FuerzaZ3:" + '\n' + ", ".join(str(x) for x in FuerzaZ3) + '\n')
194     f.write("FuerzaXY3:" + '\n' + ", ".join(str(x) for x in FuerzaXY3) + '\n')
195     f.write("FuerzaX4:" + '\n' + ", ".join(str(x) for x in FuerzaX4) + '\n')
196     f.write("FuerzaY4:" + '\n' + ", ".join(str(x) for x in FuerzaY4) + '\n')
197     f.write("FuerzaZ4:" + '\n' + ", ".join(str(x) for x in FuerzaZ4) + '\n')
198     f.write("FuerzaXY4:" + '\n' + ", ".join(str(x) for x in FuerzaXY4) + '\n')
199     f.write("FuerzaX5:" + '\n' + ", ".join(str(x) for x in FuerzaX5) + '\n')
200     f.write("FuerzaY5:" + '\n' + ", ".join(str(x) for x in FuerzaY5) + '\n')
201     f.write("FuerzaZ5:" + '\n' + ", ".join(str(x) for x in FuerzaZ5) + '\n')
202     f.write("FuerzaXY5:" + '\n' + ", ".join(str(x) for x in FuerzaXY5) + '\n')
203     f.write("FuerzaX6:" + '\n' + ", ".join(str(x) for x in FuerzaX6) + '\n')
204     f.write("FuerzaY6:" + '\n' + ", ".join(str(x) for x in FuerzaY6) + '\n')
205     f.write("FuerzaZ6:" + '\n' + ", ".join(str(x) for x in FuerzaZ6) + '\n')
206     f.write("FuerzaXY6:" + '\n' + ", ".join(str(x) for x in FuerzaXY6) + '\n')
```

```
207         f.write("tiempo:" + '\n' + ", ".join(str(x) for x in t) + '\n')
208         f.close()
209 
210 else:
211         sys.exit('Error: unable to connect')
212         vrep.simxFinish(clientID)
```

Bibliografía

[1] Add-ons. url: http://www.coppeliarobotics.com/helpFiles/en/addOns.htm (visitada el 30/11/2018).

[2] Anykode. url: http://www.anykode.com/index.php (visitada el 30/11/2018).

[3] Aristosim. url: https://mtabindia.com/srobotics.htm (visitada el 3/12/2018).

[4] Autonomous navigation virtual environment laboratory (anvel). url: https://anvelsim.com/what-is-anvel (visitada el 3/12/2018).

[5] Autonomous robots go swarming (argos). url: https://www.argos-sim.info/ (visitada el 3/12/2018).

[6] Blender. url: https://www.blender.org (visitada el 30/11/2018).

[7] Bullet physics library. url: https://pybullet.org/wordpress/ (visitada el 10/01/2019).

[8] Child scripts. url: http://www.coppeliarobotics.com/helpFiles/en/childScripts.htm (visitada el 18/11/2018).

[9] Configuracion de la api remota en el cliente. url: http://www.coppeliarobotics.com/helpFiles/en/remoteApiClientSide.htm (visitada el 11/12/2018).

[10] Configuracion de la api remota en el servidor. url: http://www.coppeliarobotics.com/helpFiles/en/remoteApiServerSide.htm (visitada el 11/12/2018).

[11] Descripcion general de la api remota. url: http://www.coppeliarobotics.com/helpFiles/en/remoteApiOverview.htm (visitada el 4/12/2018).

[12] Especificaciones de la intel euclid development kit. url: https://click.intel.com/intelr-euclidtm-development-kit.html (visitada el 7/12/2018).

[13] Funciones de python para la api remota. url: http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm (visitada el 2/12/2018).

[14] Gazebo. url: http://gazebosim.org (visitada el 3/12/2018).

[15] Interfaz de ros en v-rep. url: http://www.coppeliarobotics.com/helpFiles/en/rosInterfaces.htm (visitada el 15/11/2018).

[16] Joints en v-rep. url: http://www.coppeliarobotics.com/helpFiles/en/jointDescription.htm (visitada el 30/11/2018).

[17] Librerias dinamicas compatibles con v-rep. url: http://www.coppeliarobotics.com/helpFiles/en/dynamicsModule.htm (visitada el 20/11/2018).

[18] Linear complementary problem. url: https://en.wikipedia.org/wiki/Linear_complementarity_problem (visitada el 10/01/2019).

[19] Listado de simuladores disponibles. url: https://www.intorobotics.com/robotics-simulation-softwares-with-3d-modeling-and-programming-support/ (visitada el 30/11/2018).

[20] Lpzrobots. url: http://robot.informatik.uni-leipzig.de/software/ (visitada el 3/12/2018).

[21] Matriz de transformacion homogenea. url: http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro39/321_transformacin_de_matrices_homogneas.html (visitada el 11/12/2018).

[22] Microsoft robotics developer studio. url: https://www.microsoft.com/en-us/download/details.aspx?id=29081 (visitada el 30/11/2018).

[23] Modos de funcionamiento de la api remota. url: http://www.coppeliarobotics.com/helpFiles/en/remoteApiModusOperandi.htm (visitada el 3/12/2018).

[24] Motosim. url: https://www.motoman.com/products/software/default (visitada el 30/11/2018).

[25] Newton dynamics. url: http://newtondynamics.com/forum/newton.php (visitada el 10/01/2019).

[26] Nodos bluezero. url: https://github.com/blueworkforce/bluezero (visitada el 10/01/2019).

[27] Objetos de escena dentro de v-rep. url: http://www.coppeliarobotics.com/helpFiles/en/objects.htm (visitada el 20/12/2018).

[28] Open dynamics engine. url: http://www.ode.org/ (visitada el 10/01/2019).

[29] Pitching moment. url: https://en.wikipedia.org/wiki/Pitching_moment (visitada el 10/01/2019).

[30] Plugins. url: http://www.coppeliarobotics.com/helpFiles/en/plugins.htm (visitada el 18/11/2018).

[31] Principales caracteriticas v-rep. url: http://www.coppeliarobotics.com/helpFiles/en/vrepFeatures.htm (visitada el 30/11/2018).

[32] Proyecto ascens. url: http://ascens-ist.eu/ (visitada el 10/01/2019).

[33] Proyecto swarmanoid. url: http://www.swarmanoid.org/ (visitada el 10/01/2019).

[34] Robologix. url: https://www.robologix.com/ (visitada el 30/11/2018).

[35] Robotexpert. url: `https://www.plm.automation.siemens.com/media/store/es_es/Siemens-PLM-Tecnomatix-RobotExpert-fs_tcm1023-190476_tcm44-1981.pdf` (visitada el 30/11/2018).

[36] Robotstudio. url: `https://new.abb.com/products/robotics/es/robotstudio` (visitada el 30/11/2018).

[37] Roboworks. url: `http://www.newtonium.com/` (visitada el 30/11/2018).

[38] The ros plugin skeleton. url: `https://github.com/CoppeliaRobotics/vrep_plugin_skeleton` (visitada el 10/01/2019).

[39] Rosinterface. url: `http://www.coppeliarobotics.com/helpFiles/en/rosInterf.htm` (visitada el 10/01/2019).

[40] Shapes en v-rep. url: `http://www.coppeliarobotics.com/helpFiles/en/shapes.htm` (visitada el 3/12/2018).

[41] Simbad. url: `http://simbad.sourceforge.net/` (visitada el 3/12/2018).

[42] Tutorial sobre como construir un modelo limpio de cero en v-rep. url: `http://www.coppeliarobotics.com/helpFiles/en/buildingAModelTutorial.htm` (visitada el 27/11/2018).

[43] Virtual robot experimentation platform (v-rep). url: `http://www.coppeliarobotics.com/index.html` (visitada el 3/12/2018).

[44] Vortex dynamics. url: `https://www.cm-labs.com/` (visitada el 10/01/2019).

[45] Voxel. url: `https://es.wikipedia.org/wiki/Voxel` (visitada el 10/01/2019).

[46] Webots. url: `https://www.cyberbotics.com/` (visitada el 30/11/2018).

[47] Workspace. url: `http://www.workspace5.com/` (visitada el 30/11/2018).

[48] Writing code in and around v-rep. url: `http://www.coppeliarobotics.com/helpFiles/en/writingCode.htm#sixMethods` (visitada el 18/11/2018).

[49] Fr-W Bach, M Rachkov, J Seevers, and M Hahn. High tractive power wall-climbing robot. *Automation in construction*, 4(3):213–224, 1995.

[50] C Balaguer, JM Pastor, A Giménez, VM Padrón, and M Abderrahim. Roma: Multifunctional autonomous self-supported climbing robot for inspection applications. *IFAC Proceedings Volumes*, 31(3):563–568, 1998.

[51] Alberto Brunete, Miguel Hernando, and Ernesto Gambao. Modular multiconfigurable architecture for low diameter pipe inspection microrobots. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 490–495. IEEE, 2005.

[52] Yi Cao, Ke Lu, Xiujuan Li, and Yi Zang. Accurate numerical methods for computing 2d and 3d robot workspace. *International Journal of Advanced Robotic Systems*, 8(6):76, 2011.

[53] Sigurd A Fjerdingen, Pål Liljebäck, and Aksel A Transeth. A snake-like robot for internal inspection of complex pipe structures (piko). In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 5665–5671. IEEE, 2009.

[54] Da Guan, Lei Yan, Yibo Yang, and Wenfu Xu. A small climbing robot for the intelligent inspection of nuclear power plants. In *Information Science and Technology (ICIST), 2014 4th IEEE International Conference on*, pages 484–487. IEEE, 2014.

[55] Serena Ivaldi, Vincent Padois, and Francesco Nori. Tools for dynamics simulation of robots: a survey based on user feedback. *arXiv preprint arXiv:1402.7050*, 2014.

[56] T Miyake and H Ishihara. Wall walker: proposal of locomotion mechanism cleaning even at the corner. In *Climbing and Walking Robots*, pages 341–348. Springer, 2005.

[57] Lucas Nogueira. Comparative analysis between gazebo and v-rep robotic simulators. *Seminario Interno de Cognicao Artificial-SICA*, 2014:5, 2014.

[58] Kishan Panchal, Chirag Vyas, and Dhaval Patel. Developing the prototype of wall climbing robot. *International Journal of Advance Engineering and Research Development*, 1(3), 2014.

[59] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, et al. Argos: a modular, multi-engine simulator for heterogeneous swarm robotics. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 5027–5034. IEEE, 2011.

[60] Lenka Pitonakova, Manuel Giuliani, Anthony Pipe, and Alan Winfield. Feature and performance comparison of the v-rep, gazebo and argos robot simulators. In *Annual Conference Towards Autonomous Robotic Systems*, pages 357–368. Springer, 2018.

[61] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1321–1326. IEEE, 2013.

[62] Matthew J Spenko, G Clark Haynes, JA Saunders, Mark R Cutkosky, Alfred A Rizzi, Robert J Full, and Daniel E Koditschek. Biologically inspired climbing with a hexapedal robot. *Journal of field robotics*, 25(4-5):223–242, 2008.