Trabajo Fin de Máster
Máster Universitario en Ingeniería Industrial

Desarrollo y modelado 3D de planta termosolar

Autor: Eduardo Mayoral Briz
Tutores: Antonio J. Gallego Len
Ramón A. García Rodríguez

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

Sevilla, 2021

Trabajo Fin de Máster
Máster Universitario en Ingeniería Industrial

Desarrollo y modelado 3D de planta termosolar

Autor:
Eduardo Mayoral Briz

Tutor:
Antonio J. Gallego Len
Profesor

Ramón A. García Rodríguez
Profesor

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Proyecto Fin de Carrera: Desarrollo y modelado 3D de planta termosolar

Autor: Eduardo Mayoral Briz

Tutores: Antonio J. Gallego Len
Ramón A. García Rodríguez

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El secretario del Tribunal

A mi familia, mi pareja y mis amigos

A mis maestros

Agradecimientos

Este trabajo no hubiera conocido nunca un final si no hubiera sido por el apoyo incondicional de mi familia, de María y de todos mis amigos, y representa un precioso broche final a toda esta etapa de estudios universitarios en la Escuela de la que, a pesar de las dificultades, guardaré un bonito recuerdo.

Gracias también a Antonio y Ramón por toda la ayuda, el aprendizaje, y por entender con paciencia en todo momento mi situación personal a lo largo del desarrollo del trabajo.

No habría logrado este éxito si cualquiera de estas piezas del puzle no hubiera estado en su sitio.

Gracias a todos.

Eduardo Mayoral Briz

Sevilla, 2021

Resumen

En un mundo donde la demanda energética es cada vez mayor, a la par que cada día se intenta apostar cada vez más por la sostenibilidad ambiental, no parece viable dejar de apostar por la generación eléctrica a partir de energías renovables. La obtención de ella a partir del Sol en plantas termosolares es una rama más de todo el árbol de opciones disponibles.

En paralelo a lo anterior, y en concreto en los últimos años, el crecimiento de las nuevas tecnologías aplicadas a la robotización y automatización de tareas en la industria han abierto un nuevo paradigma. Las plantas de generación no se quedan atrás, y la incorporación de estos nuevos métodos es algo que, al igual que en muchos otros campos, puede proporcionar muchas ventajas.

A lo largo de este proyecto se desarrolla un entorno de simulación de una planta termosolar real, con el fin de dotarla de capacidad de interacción con futuros modelos que se incorporen a ella. Esto permite disponer de todo un campo de pruebas de cara a ensayos y pruebas de cualquier tipo de robot enfocado a la inspección, soporte o vigilancia de la planta, entre otros.

Para ello, y usando como base ROS y Gazebo, se diseña y programa una simulación de planta termosolar a partir de una real, intentando ser lo más fiel posible a esta. En gran parte, esto es posible gracias a que se crean los modelos existentes en la planta desde cero, usando Blender, manteniendo así el mayor nivel de detalle posible siempre que el rendimiento de la simulación no se vea afectado. Una vez se incorporan estos modelos a la simulación, es posible realizar diferentes análisis dinámicos de cara a obtener el mejor método para el movimiento general de la planta.

Por otro lado, también se realiza una gestión y automatización de los elementos de la planta mediante XML y Python. Esto implica adaptar el sistema de comunicación de ROS a las diferentes tareas que surgen, sobre todo dado el elevado número de modelos con el que se trabaja, así como los diferentes casos puntuales que deban ser tratados con distinción respecto al resto.

Por último, se incorpora a la simulación un robot de inspección básico, el cual es controlado manualmente por teclado y permite mostrar cual es el comportamiento general de un elemento externo a la planta cuando es introducido en la simulación. Esto sienta las bases a futuros proyectos complementarios y de ampliación de diversos enfoques.

Abstract

In a world where the demand for energy is ever increasing, at the same time as there is a growing commitment to environmental sustainability, it does not seem viable to stop betting on electricity generation from renewable energies. Obtaining it from the sun in solar thermal plants is just one more branch of the tree of available options.

In parallel to the above, and specifically in recent years, the growth of new technologies applied to the robotisation and automation of tasks in industry has opened a new paradigm. Power plants are not lagging, and the incorporation of these new methods is something that, as in many other fields, can provide many advantages.

Throughout this project, a simulation environment of a real solar thermal plant is developed, to provide it with the capacity to interact with future models that are incorporated into it. This makes it possible to have a whole test field available for the testing and testing of any type of robot focused on the inspection, support, or surveillance of the plant, among others.

To do this, and using ROS and Gazebo as a base, a simulation of a thermosolar plant is designed and programmed based on a real one, trying to be as faithful as possible to it. To a large extent, this is possible thanks to the fact that the existing models of the plant are created from scratch, using Blender, thus maintaining the highest possible level of detail if the performance of the simulation is not affected. Once these models are incorporated into the simulation, it is possible to perform different dynamic analyses in order to obtain the best method for the general movement of the plant.

On the other hand, management and automation of the plant elements is also carried out using XML and Python. This involves adapting the ROS communication system to the different tasks that arise, especially given the large number of models with which it works, as well as the different specific cases that need to be treated differently from the rest.

Finally, a basic inspection robot is incorporated into the simulation, which is manually controlled by keyboard and makes it possible to show the general behaviour of an element outside the plant when it is introduced into the simulation. This lays the foundations for future complementary projects and the extension of various approaches.

Índice

Agradecimientos **ix**
Resumen **xi**
Abstract **xiii**
Índice **xv**
Índice de Figuras **xvii**

1 Introducción **1**
1.1 Planta termosolar de colectores cilindro parabólicos *2*
1.2 Objetivos *3*

2 ROS **5**
2.1 Conceptos básicos *6*
2.1.1 Nodos 6
2.1.2 Master 7
2.1.3 Servidor de parámetros 7
2.1.4 Topics 7
2.1.5 Mensajes 8
2.1.6 Servicios 8
2.1.7 Bags 9
2.2 Estructura y jerarquía de archivos en ROS *9*
2.3 Comandos principales *11*
2.3.1 Comandos de navegación 11
2.3.2 Comandos para obtención de información 11
2.3.3 Comandos de ejecución 12
2.3.4 Comandos de herramientas gráficas 12
2.3.5 Otros comandos 12
2.4 Instalación y configuración de ROS *13*
2.4.1 Instalación 13
2.4.2 Configuración del entorno de trabajo 13

3 Gazebo **15**
3.1 Instalación de Gazebo *15*
3.2 Simulaciones conectando Gazebo y ROS *16*
3.3 Creación de mundos *18*
3.3.1 Modificación manual 19
3.3.2 Modificación en *scripts* SDF 19

4 Modelado en 3D con Blender **25**
4.1 Introducción *25*
4.1.1 Operadores 27
4.1.2 Modificadores 30
4.2 Modelado de un módulo *32*
4.2.1 Soporte 33
4.2.2 Parábola 34
4.3 Modelado de un módulo simplificado *38*

4.4 Modelado de un colector *38*
4.5 Modelado de un lazo *39*

5 Dinámica de la simulación **43**
5.1 Movimiento de un colector *43*
5.1.1 Idea inicial: uso de *topics* de Gazebo 44
5.1.2 Creación de *plugins* propios como primera alternativa 45
5.1.3 Idea final: control en bucle cerrado 45
5.2 Movimiento de un lazo *50*
5.3 Gestión de varios lazos *51*
5.4 Scripts de automatización del movimiento *54*
5.4.1 Detección de *topics* activos 54
5.4.2 Escucha en posición de reposo 54
5.4.3 Movimiento general 55

6 Implementación del robot de inspección **57**
6.1 Modelado del robot *57*
6.2 Control manual del robot *58*

7 Conclusiones y líneas futuras **61**

Anexo A. Código implementado **63**
A.1 Lanzamiento de la simulación: multi_lazo_main.launch *63*
A.2 Generación de un modelo de lazo: multi_lazo_main.launch *66*
A.3 Mundo de la simulación (SDF): simple.world *67*
A.4 Modelo de lazo (URDF): lazo.urdf *72*
A.5 Script para sensor en modo espera: home_listener.py *75*
A.6 Script para generación de topics: topic_pub_generator.py *76*
A.7 Script para automatización de movimiento: move_auto.py *77*

Referencias **81**

Glosario **83**

ÍNDICE DE FIGURAS

Figura 1-1. Módulo captador (Protermosolar, 2020). 2

Figura 1-2. Esquema simplificado de la planta. 2

Figura 2-1. Distribuciones ROS en los últimos cinco años (ROS Wiki, 2018). 5

Figura 2-2. Ejemplo de la función del ROS Master, de izquierda a derecha (ROS Wiki, 2018). 7

Figura 2-3. Ejemplo básico del *Computation graph* de ROS (The Robotics Back-End, 2021). 8

Figura 2-4. Modificación implementada en *.bashrc*. 9

Figura 2-5. Ejemplo de la jerarquía de archivos dentro del *workspace* (Develop Paper, 2020). 10

Figura 2-6. Organización del paquete *ptcplant*. 10

Figura 2-7. Inicialización y compilación del *workspace*. 14

Figura 3-1. Interfaz de Gazebo tras la primera ejecución. 16

Figura 3-2. Ejemplo de archivo *.launch* básico. 17

Figura 3-3. Resultado de *rqt_graph* tras ejecutar el archivo *.launch* básico (nodos en elipses, *topics* en rectángulos). 17

Figura 3-4. Apartado de las físicas en el archivo *.world*. 19

Figura 3-5. Apartado de la escena en el archivo *.world*. 20

Figura 3-6. Apartado de la iluminación en el archivo *.world*. 20

Figura 3-7. Material implementado para el suelo en la simulación. 22

Figura 4-1. A la izquierda, modo edición. A la derecha, modo objeto. 26

Figura 4-2. De izquierda a derecha: *autosmooth* aplicado con ángulos de 0º, 15º, 30º y 90º. 27

Figura 4-3. Aplicación de *extrude*: en el centro, por defecto; a la derecha, *extrude individual*. 28

Figura 4-4. Aplicación de *inset faces* en la cara superior de un cubo. 28

Figura 4-5. Aplicación de *bevel* en la cara superior de un cubo. 28

Figura 4-6. Aplicación de *loop cut* en un cilindro para posterior extrusión. 29

Figura 4-7. Aplicación de *spin* en un cuadrado, usando 12 pasos y 360º de giro. 29

Figura 4-8. Uso de *array* con rotación para crear una cadena a partir del modelo de un eslabón. 31

Figura 4-9. De izquierda a derecha, uso de *boolean* con parámetros *union, diference* e *intersect*. 31

Figura 4-10. Uso de *mirror* sobre el eje en verde, y posterior uso de *array*. 31

Figura 4-11. Aplicación de *solidify* en superficie cilíndrica. 32

Figura 4-12. De izquierda a derecha: modelo original; *subdivision surface* aplicado una vez; *subdivision surface* aplicado tres veces. 32

Figura 4-13. Base y eje de las patas del módulo. 33

Figura 4-14. Modelo final del soporte del módulo. 34
Figura 4-15. Mallado de la parábola creada y de los soportes del tubo portador del líquido. 35
Figura 4-16. Mallado del eje y la estructura que soportan la parábola. 35
Figura 4-17. Modelo final de la parábola. 36
Figura 4-18. Renderizado frontal del módulo completo. 37
Figura 4-19. Renderizado trasero del módulo completo. 37
Figura 4-20. Modelo de un módulo simplificado. 38
Figura 4-21. Modelo de colector a partir del modelo de módulo simple. 39
Figura 4-22. Modelo de lazo a partir del modelo de colector. 39
Figura 4-23. Uniones entre colectores que comparten eje de giro. 40
Figura 4-24. Unión entre colectores, perpendicular al eje de giro. 41
Figura 4-25. Unión entre colectores y distribución general. 42
Figura 5-1. Bloque de transmisión usado para el control de un colector. 46
Figura 5-2. Archivo de configuración YAML para controlar un colector. 46
Figura 5-3. En rojo, mensaje de error obtenido al iniciar la primera simulación con *ROS Control*. 47
Figura 5-4. Nodo generador de modelos. 47
Figura 5-5. Nodo de carga de controladores. 48
Figura 5-6. Nodo publicador del estado del robot. 48
Figura 5-7. Uso de *rostopic* para mover un modelo a la posición angular 1,2 radianes. 48
Figura 5-8. *Message Publisher* de *rqt*. 49
Figura 5-9. Visualizador gráfico de *rqt*. 49
Figura 5-10. *Dynamic Reconfigure* de *rqt*. 50
Figura 5-11. Archivo *lazo_control.yaml*. 50
Figura 5-12. Nodo de carga de controladores para un lazo. 51
Figura 5-13. Esquema de lanzamiento de la simulación. 52
Figura 5-14. Fragmento del nuevo archivo de lanzamiento de modelos usando *namespaces*. 52
Figura 5-15. Organización de los lazos en la simulación. 53
Figura 5-16. Clase *Lazo* creada para registrar las características de cada eje. 55
Figura 6-1. Plugin usado para el movimiento del robot y argumentos de entrada. 57
Figura 6-2. Aspecto visual del robot de inspección. 58
Figura 6-3. Parámetros y nodo de generación del robot de inspección. 58
Figura 6-4. Control manual por teclado del robot de inspección. 59

1 INTRODUCCIÓN

Son numerosas las diferentes tecnologías usadas en la actualidad para la generación de energía eléctrica. Dentro de las consideradas como renovables, una de las más extendidas y con mayor potencial es la que aprovecha la radiación solar para ello, que a su vez se divide en energía solar fotovoltaica y térmica. Si bien la primera es más conocida en términos generales, habiendo llegado incluso a nuestros propios hogares para desarrollar autoconsumo eléctrico, a lo largo del presente trabajo la idea central de generación será basada en la segunda.

La energía solar térmica se basa en el aprovechamiento de la energía solar para producir calor. Obtenido dicho calor, podemos emplearlo de diferentes formas: desde labores domésticas como la cocina, la calefacción en el hogar o la obtención de agua caliente, hasta grandes procesos industriales con otros fines. Dependiendo del sector en el que nos encontremos, los rangos de temperatura en los que nos movamos serán muy diferentes, y por tanto los materiales necesarios para los procesos (La Energía Solar, 2019).

El caso que se tratará a lo largo de este trabajo corresponde al de una planta termosolar de colectores cilindro parabólicos, donde se hace un aprovechamiento como el mencionado con anterioridad a nivel industrial. Como en la mayoría de las plantas de dicho tipo, el fin es la obtención de energía eléctrica a partir del calor obtenido mediante diferentes elementos alojados en una zona de potencia para la conversión.

Es posible encontrar diferentes tipos de plantas termosolares. Los más comunes son los siguientes:

- Central termosolar de torre. Se compone de una serie de heliostatos y de un receptor. Los primeros, que se encontrarán en una situación ordenada en el terreno y con una orientación concreta para cada uno, reflejan la luz del sol, con el fin de que los rayos sean recibidos por el receptor. Este último, situado generalmente en una torre, recibirá toda la luz, alcanzará la mayor densidad de potencia, y con ello un aumento de la temperatura.

- Central de colectores cilindro parabólicos. Difiere con la anterior en la forma de llevar el calor a la zona de potencia. En este caso, se compone de numerosos colectores de forma cilindro parabólica, por cuyos ejes focales se hace pasar un ciclo cerrado de un determinado fluido. De esta forma, todos los rayos que reflejan en la parábola aterrizan en el tubo que contiene dicho fluido, elevando considerablemente la temperatura de este. Gracias al estado en el que vuelve el líquido a la zona de potencia, obtenemos el calor necesario para la transformación.

- Central de colectores Fresnel. La idea de funcionamiento es similar a la de colectores cilindro parabólicos. Algunas de las diferencias principales son una mayor distancia focal, el uso de elementos reflectores planos y la necesidad de un menor espacio. De esta forma, y en líneas generales, este método resulta más sencillo de construir y económico, aunque es de menor eficiencia.

1.1 Planta termosolar de colectores cilindro parabólicos

Como se ha comentado anteriormente, en el presente proyecto se trabajará en base a una central de colectores cilindro parabólicos. El principio fundamental de esta es la forma geométrica parabólica de los captadores. A lo largo de ellos, y recorriendo el eje focal de la parábola, discurre el fluido que va acumulando el calor obtenido por la reflexión de los rayos del Sol. Por otro lado, estos captadores tienen un movimiento rotatorio, y modifican su orientación de forma que su exposición solar, y con ello la transferencia de calor con el fluido, sea la máxima posible.

Figura 1-1. Módulo captador (Protermosolar, 2020).

La estructura más básica de la planta es un módulo, como el que podemos ver en la Figura 1-1. El encadenamiento en línea recta de un determinado número de módulos da lugar a un colector. El número de módulos en un colector, aunque es variable según el diseño realizado, está en torno a los doce, quedando una longitud final de colector de aproximadamente 150 metros.

En cada uno de los bucles que realiza el fluido para calentarse, recorre un total de cuatro colectores. Estos están situados en parejas de dos, de forma que cada par se sitúa en un mismo eje de giro. De esta forma, el líquido fluye por los dos primeros colectores en un sentido, y por los otros dos en el sentido contrario. Este bucle completo formado por los cuatro colectores se denomina lazo.

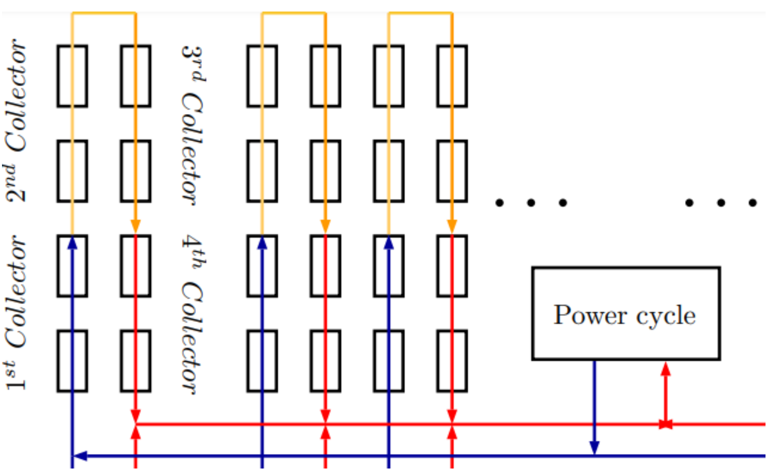


Figura 1-2. Esquema simplificado de la planta.

Así, la estructura general de la planta quedaría como se ve en la Figura 1-2: un par de tuberías generales, de gran tamaño, hacen las funciones de entrada y salida de la zona de potencia; de la tubería con líquido frío se ramifican otras más pequeñas, que recorrerán cada uno de los lazos. Finalmente, se realiza un recorrido inverso, y todas las pequeñas ramificaciones (ya con el líquido a temperatura máxima) concurren en la tubería general de líquido caliente, que llegará hasta la zona de potencia.

1.2 Objetivos

Tal y como se desarrollará en los próximos capítulos, se elaborará un modelo en 3D de una planta termosolar, el cual se incorporará posteriormente a una simulación dinámica utilizando Gazebo y ROS. El objetivo final es obtener un modelo de planta que desarrolle un movimiento lo más fiel a la realidad posible. Para ello, se usarán controladores PID para los ejes de giro de cada uno de los colectores. De esta forma, resultará un sistema en el cual el movimiento de cada colector es independiente.

Por norma general, realizarán un movimiento de seguimiento al Sol para maximizar el número de rayos de luz incidentes. Sin embargo, habrá casos en los que alguno deba mantenerse en la posición de reposo para, por ejemplo, recibir un mantenimiento programado o una reparación.

Así, obtendremos un modelo del complejo en el cual se podrá hacer una simulación dinámica en tiempo real, y en el que existen una serie de físicas y colisiones que permitirán la interacción con otros futuros elementos introducidos en la simulación. Finalmente, se incorporará un sencillo robot de inspección controlado manualmente capaz de interactuar con los elementos presentes en el modelo, que sienta las bases a futuras ideas de continuación o ampliación del presente trabajo.

El objetivo del presente proyecto es disponer de una simulación dinámica de una planta termosolar al completo, en la cual puedan incorporarse diferentes modelos adicionales de cara a la realización de pruebas y mejoras en la planta. Lo modelos se replicarán, tanto visual como físicamente, gracias al motor de físicas y colisiones en la simulación. De esta forma, se dispondría de un entorno ideal de pruebas, debido a que el comportamiento dinámico de la planta será lo más parecido posible a la realidad.

Para llegar a dicho objetivo final, es necesario ir alcanzando otros tantos intermedios. En primer lugar, será prioritario configurar correctamente el entorno de trabajo donde se desarrollará la simulación. Dicho entorno estará formado en conjunto por ROS y Gazebo. Esto implica que, más allá de configurar ambos softwares por separado, será necesario configurar todas las dependencias y relaciones entre ellos, de cara a que el funcionamiento del conjunto sea el oportuno. En paralelo a ello, será recomendable realizar pruebas con modelos muy básicos para comprobar el funcionamiento y posteriormente extrapolarlo a los modelos de la planta, para organizar la depuración de errores.

El siguiente paso es desarrollar desde cero los diferentes modelos 3D en Blender. Se comenzará con el diseño de un módulo, pasando por el de un colector, y llegando al de un lazo. Estos modelos serán originales para este proyecto, y se crearán tomando como base diferentes referencias en plantas reales. En cada uno de los pasos se deberá comprobar que los modelos se cargan correctamente en ROS, y que Gazebo es capaz de dar visualización a estos sin ningún problema.

También será fundamental desarrollar el código que describa cada uno de los modelos creados, así como de todos los elementos partícipes en la simulación: divisiones de los modelos, articulaciones y relaciones entre ellos; características físicas como masas, inercias y rozamientos; y aspectos visuales, tanto de la escena como de los propios modelos.

Una vez definido lo anterior, uno de los objetivos claves será programar toda la dinámica, desde un simple módulo hasta la totalidad de la planta. En un comienzo, se estudiarán las diversas alternativas para capacitar de movimiento a un módulo: integración de la gestión del movimiento en Gazebo/ROS, tipo de control aplicado, elementos mecánicos y controladores necesarios.

Tras conseguir aplicar esto al elemento más pequeño de la simulación, se continuará probando en colectores y lazos. Una vez que el control manual sea posible, se procederá a la automatización del movimiento de toda la planta. Será necesario programar en Python o C++ las órdenes del movimiento de cada uno de los módulos, así como de las diferentes excepciones que se presenten en la planta.

Además, también será obligatorio programar todos los sensores y actuadores adicionales, así como los *scripts* necesarios para gestionar el intercambio de información por mensajes entre los diferentes módulos de Gazebo y ROS. Esto alcanzará cierta complejidad cuando el número de elementos a controlar en paralelo sea más numeroso, ya que entrará en juego la capacidad de computación del ordenador, y será necesario un análisis de las físicas de la simulación para su optimización.

Finalmente, se modelará un sencillo robot de inspección para introducirlo en la simulación. Se controlará manualmente por teclado gracias a un plugin de la librería ROS, el cual tendrá que ser readaptado según las características del modelo creado. Este será capaz de interactuar con los elementos presentes en el modelo general de la planta.

Esto sienta las bases a futuras ideas de continuación o ampliación del presente trabajo, en el cual se podrá disponer de todo un entorno de pruebas para integrar nuevos modelos en la planta y realizar un estudio dinámico de ellos y las colisiones con la planta termosolar.

2 ROS

En la actualidad, existen multitud de plataformas orientadas a la programación de robots. De entre todas ellas, ROS (Robot Operating System) destaca por ser una plataforma de código abierto que provee servicios que esperaríamos obtener de un sistema operativo: abstracción de hardware, paso de mensajes entre procesos, capacidad de generar y correr código, y control de dispositivos de bajo nivel, entre otros.

ROS nace en el año 2007 bajo el nombre de *Switchyard*, como un proyecto de la Universidad de Stanford. Un año después, sería la start-up *Willow Garage* la encargada del desarrollo, donde se dio la mayor parte de este. En 2013, se funda la *Open Source Robotics Foundation* (OSRF), organización que en la actualidad se encarga del mantenimiento activo (Joseph, 2017).

El principal objetivo de ROS no reside en ser la plataforma de software robótico con más características. Realmente, ROS busca la modularidad y la reutilización: que cada uno de los procesos desarrollados sea fácilmente acoplable a otros. Además, busca ser un conjunto liviano, independiente del lenguaje usado y escalable.

Hoy en día, ROS se ha desarrollado para ser ejecutado en plataformas basadas en Unix, siendo el software de pruebas habitual tanto Ubuntu como Mac OS X. Sin embargo, la comunidad colabora y contribuye dando soporte para otras plataformas Linux, así como Windows mediante virtualización (ROS Wiki, 2018).

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (**Recommended**)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017

Figura 2-1. Distribuciones ROS en los últimos cinco años (ROS Wiki, 2018).

ROS se presenta en diversas distribuciones con el paso de los años, de un modo similar a como lo hace Ubuntu (Figura 2-1). De hecho, aunque no es algo obligatorio, cada una de estas distribuciones están pensadas para ser usadas con una determinada versión de Ubuntu. Al igual que este, mantienen una serie de características

principales a las que se les da soporte hasta su final de vida (EOL, en inglés *End Of Life*).

Actualmente, la distribución más reciente de ROS es Noetic, la cual apareció en mayo del año 2020 y tendrá cinco años de soporte. Sin embargo, para este trabajo se ha empleado ROS Melodic, debido a que nació dos años antes y cuenta con los mismos años de soporte. Esta decisión se toma debido a que uno de los principales problemas de las nuevas distribuciones es que no se han migrado todos los paquetes de anteriores distribuciones, evitando así ciertos problemas que no aparecerán usando distribuciones más trabajadas en el tiempo.

Otro aspecto importante para tener en cuenta cuando elegimos una distribución de ROS es fijarnos en el tiempo de soporte. Al igual que Ubuntu, en ROS existen distribuciones *LTS*, que son aquellas en las cuales se da un soporte a largo plazo. Kinetic, Melodic o Noetic son ejemplos de ello, teniendo un soporte de cinco años. Por otro lado, otras como Jade o Lunar son menos recomendadas debido a que solo presentan dos años de soporte.

Desde 2015, y en paralelo a ROS, también se está trabajando en otra versión (que no distribución) de ROS, conocida como ROS2. Esta cuenta con sus propias distribuciones, y presenta diferencias sustanciales con respecto a su versión primigenia. Nace con la idea de hacer frente a diferentes necesidades en la robótica actual que, como tantos campos, está en constante cambio y crecimiento.

En líneas generales, ROS2 camina hacia la descentralización, la seguridad y los sistemas en tiempo real. Implementa algunos cambios como el soporte a nuevas versiones de Python y C++, la posibilidad de uso en Windows 10 sin virtualización, la descentralización del sistema eliminando el *ROS Master*, y el cambio en protocolos de comunicación y herramientas de compilación. Por todo ello, y aunque a priori ROS2 es el futuro, no se podría calificar alguna versión como la mejor, si bien cada usuario deberá elegir la que más convenga según el fin requerido (Geek Gasteiz, 2018).

2.1 Conceptos básicos

En ROS, la capa básica en la cual los procesos intercambian la información es llamada en inglés *Computation Graph*, que podría ser considerada como una red de procesos *peer-to-peer*. Dentro de la capa mencionada anteriormente, existen una serie de conceptos que contribuyen a la red de distintas formas.

2.1.1 Nodos

Son cada uno de los procesos encargados de la computación. Se podrían entender como módulos software que envían y reciben información a otros nodos. Aunque siempre pueden ser más complejos, básicamente los nodos en ROS son procesos POSIX, lo que ofrece cierta tolerancia al fallo: en caso de existir problemas de ejecución en cierto nodo, esto no afectará al resto, que seguirá intercambiando mensajes de la forma que se haya establecido (Quigley, Gerkey, & D. Smart, 2016).

Concretamente, un nodo es un fragmento de código que debe estar escrito usando una librería cliente de ROS. Actualmente, las librerías con soporte oficial y las más usadas son *roscpp* y *rospy*, que dan soporte a la programación en C++ y Python, respectivamente. Cada uno de estos *scripts* puede ser lanzado al inicio de la simulación o durante el tiempo de ejecución. Además, estos pueden lanzarse con argumentos opcionales, lo que permite ejecutar un mismo nodo con diferentes configuraciones al mismo tiempo.

En sistemas más complejos, lo normal es diversificar las tareas entre diferentes nodos. Por ejemplo, supongamos el caso de un robot con varias ruedas, que a su vez tiene dos sensores de proximidad y un procesador para realizar el mapeado de una habitación: diseñaremos un nodo que gestione la tracción, otro la detección y otro el procesamiento. Entre ellos, se comunicarán unos con otros para enviar y recibir la información necesaria para cada tarea.

Los nodos deben tener un nombre único y diferente al resto de nodos, o no podrán ser ejecutados. Por tanto, para el ejemplo anterior, podemos optar por diseñar un nodo que gestione todas las ruedas, o bien un nodo por rueda. Esto dependerá de las características del robot, y del fin especificado. Sin embargo, si optamos por la segunda opción, debemos dotar al nodo de cada rueda con un nombre diferente.

2.1.2 Master

El ROS Master se encarga de proveer a todos los nodos la información del resto de elementos presentes en el sistema. Su inicialización es imprescindible si queremos ejecutar cualquier nodo, el cual deberá notificar al Master de su existencia antes de poder comenzar a publicar información. Una vez el Master conoce de la existencia de un nuevo nodo, proporciona a este la capacidad de conectarse *peer-to-peer* con el resto.

Por tanto, la principal responsabilidad del Master es mantener un registro con todos los nodos existentes. Esto es algo fundamental ya que, como se comentó antes, nuevos nodos pueden aparecer en tiempo real durante la ejecución. El Master debe ser capaz de proporcionar conexiones dinámicas para enlazar todos los procesos. En la Figura 2-2 se puede apreciar un resumen gráfico de la labor del Master cuando aparecen dos nuevos nodos, tanto publicadores como suscriptores.

Aunque existen formas alternativas mucho más rápidas, en general, para inicializar el Master se usa en el terminal el comando `roscore`. Este comando, además del Master, también ejecuta otros componentes que son fundamentales para el correcto funcionamiento de ROS, como por ejemplo el servidor de parámetros.

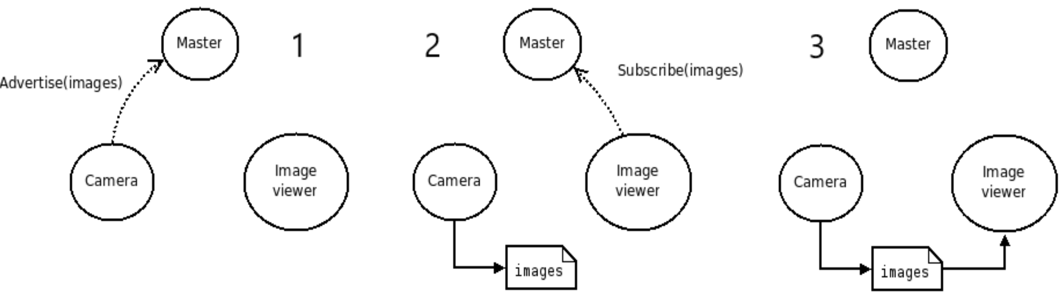


Figura 2-2. Ejemplo de la función del ROS Master, de izquierda a derecha (ROS Wiki, 2018).

2.1.3 Servidor de parámetros

El servidor de parámetros forma parte del Master de ROS. Se podría entender como un diccionario compartido de variables, que es accesible por el resto de los elementos mediante APIs y que es utilizado por los nodos como un servidor para alojar y extraer ciertos parámetros con un determinado valor en el tiempo de ejecución. Debido a que su diseño no está enfocado al alto rendimiento, está destinado a alojar elementos estáticos, como por ejemplo los de configuración inicial (ROS Wiki, 2018).

2.1.4 Topics

Los mensajes entre los distintos nodos siguen un sistema de transporte basado en la publicación y la suscripción. Cuando dos nodos quieren intercambiar información, no lo hacen de forma directa entre ambos. Existe un elemento intermedio, el *topic*, que se encarga de alojar el contenido de cierto tipo de mensaje y de ponerlo a disposición del resto de nodos existentes. Es importante que cada *topic* tenga una identificación única para evitar conflictos en la comunicación.

Por explicarlo de una forma más concreta, se podría hacer una analogía con la comunicación entre personas: la comunicación entre emisor y receptor (entre dos nodos) no es directa, si no que requiere de un canal comunicativo como puede ser el aire o un papel, en los casos de comunicación oral y escrita más básicos. Un *topic* se podría considerar como algo parecido al canal comunicativo entre dos nodos cualesquiera.

De este modo, podemos encontrar casos muy variados. Son desde uno a varios el número de nodos que pueden publicar o escuchar de cierto nodo. Además, en el caso de existir varios nodos enlazados a un *topic*, cada uno de ellos desconoce por completo si el resto están o no enlazados. De esta forma se confiere al sistema de gran modularidad, ya que el intercambio de mensajes está totalmente desacoplado entre nodos.

Todo este sistema de comunicación está basado por defecto en protocolos TCP/IP, conocido como TCPROS. Sin embargo, también existen métodos UDP (UDPROS) con menores latencia y pérdidas, aunque sólo disponibles al usar las librerías *roscpp*. Son los propios nodos los que negocian el tipo de comunicación durante el tiempo de ejecución (ROS Wiki, 2018).

En la Figura 2-3 podemos encontrar un ejemplo con algunos de los elementos del *Computation graph* vistos hasta ahora. En concreto, se trata de una simulación en la que se manejan dos tortugas idénticas (nodo */turtlesim*) a través de órdenes por teclado (nodo *teleop_turtle*). En azul podemos ver ambos nodos. En rojo, podemos ver cada uno de los *topics* necesarios para la gestión de velocidad, posición y sensor de cada tortuga. Además, el sentido de cada una de las flechas determina el flujo de la información entre nodos y *topics*.

Por otro lado, en verde podemos ver los llamados *namespaces*. En esta ocasión, su creación es de obligado cumplimiento, ya que se están usando dos tortugas en lugar de una. Se usan para diferenciar cada uno de los *topics* propios de cada tortuga. Si no usáramos diferentes espacios, estaríamos creando *topics* con el mismo nombre, lo que provocaría un conflicto e imposibilitaría la ejecución.

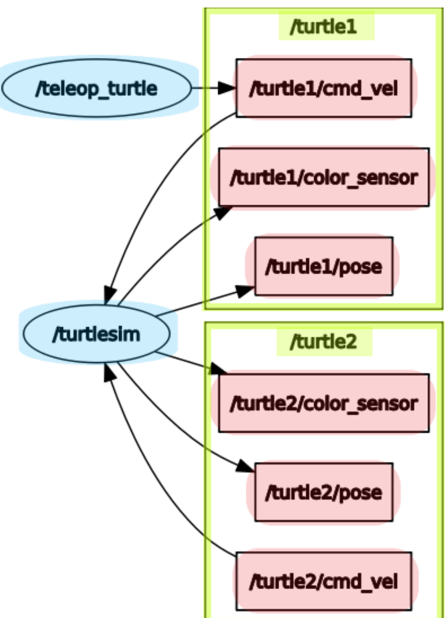


Figura 2-3. Ejemplo básico del *Computation graph* de ROS (The Robotics Back-End, 2021).

2.1.5 Mensajes

Representan la información intercambiada entre nodo y *topic*. Sin embargo, dicha información va más allá de un simple dato de cierto tipo. Los mensajes son estructuras de datos, muy similares a las que podemos encontrar en C, y por tanto pueden albergar múltiples datos de diferentes tipos. Así mismo, y al igual que en las estructuras de C, podemos encontrar mensajes con estructuras anidadas.

Aunque siempre existe la posibilidad de usar tipos de mensajes básicos predefinidos, lo normal es crear un tipo de mensaje propio que satisfaga las necesidades de la aplicación. En este caso, será necesario crear un fichero con extensión *.msg* dentro del paquete en el que trabajemos. Además, tendremos que hacer referencia a este nuevo mensaje en el archivo *CMakeLists.txt* del paquete, como se comentará posteriormente, para que este sea compilado previamente a la ejecución (ROS Wiki, 2018).

2.1.6 Servicios

En ciertas ocasiones, el modelo de suscripción y publicación de mensajes puede no ser tan apropiado, sobre todo

cuando el tráfico crece considerablemente. En estos casos, parece apropiado adoptar un modelo de solicitud y respuesta, como suele ser requerido en los sistemas distribuidos, y para ello existen los servicios.

Al igual que en los mensajes, los servicios son creados dentro de un paquete con archivos de extensión *.srv* con un nombre único, y deben contener tanto el tipo de mensaje de la solicitud como el de la respuesta. Cuando un nodo ejecute cierto servicio, haciendo las funciones de servidor, quedará a la espera de las peticiones que realicen otros nodos cliente.

2.1.7 Bags

Se encargan de almacenar información, además de poder analizarla, visualizarla e incluso reenviarla. Se le pueden dar diversos usos, desde un uso *online* en el que por ejemplo se realiza un reenvío de la información a los mismos *topics* de los que se recibió la información, hasta un uso *offline*, como podría ser la elaboración de registros a partir de los datos obtenidos de cierto sensor para analizar su correcto funcionamiento.

2.2 Estructura y jerarquía de archivos en ROS

A la hora de trabajar en ROS, nuestro directorio raíz será un espacio de trabajo que dependerá del sistema de compilación usado. Originalmente existió *rosbuild*, sin embargo, con el paso de los años el sistema más usado es *catkin*, que no es más que una sistema similar a *CMake* con una personalización enfocada a ROS.

Todo este entorno mencionado anteriormente quedará instalado por defecto a la vez que ROS, y nos permitirá utilizar simples comandos para compilar y generar nuestro espacio de trabajo. Esto es algo que habrá que realizar cada vez que modifiquemos ciertos archivo, como por ejemplo *CMakeLists.txt*.

En concreto, usaremos en el terminal el comando `catkin_make`, estando ubicados en nuestro *workspace*, para compilar y generar las modificaciones. Posteriormente, mediante el comando `source` debemos ejecutar el archivo *setup.bash* ubicado en la carpeta *devel*. Si bien la compilación es opcional según haya habido cambios o no, la ejecución del *setup.bash* es obligatoria en cada nuevo terminal que abramos, o de lo contrario no serán abiertos los archivos necesarios para la correcta ejecución de ROS.

Para evitar tener que ejecutar el *setup.bash* en cada nuevo terminal, en el presente trabajo se ha tomado una alternativa que agiliza la ejecución mediante la modificación del archivo *.bashrc*, ubicado en la raíz. Las alteraciones ejecutadas en dicho archivo nos permitirán realizar transformaciones en cada nuevo terminal que abramos, así como correr líneas de código que implementemos. En concreto, en esta ocasión se ha realizado la modificación que podemos ver en la Figura 2-4, que se encarga de ejecutar el comando necesario siempre que el archivo exista, y de mostrar por pantalla la confirmación de la ejecución.

```
if [ -f /home/edumbriz/TFM/devel/setup.bash ]; then
        source /home/edumbriz/TFM/devel/setup.bash
        echo "[~/.bashrc] /home/edumbriz/TFM/devel/setup.bash ejecutado!"
fi
```

Figura 2-4. Modificación implementada en *.bashrc*.

Podemos ver una estructura general de todos los archivos en la Figura 2-5. Dentro del *workspace* tenemos tres carpetas: *build*, donde se alojan todos los archivos que intervienen en el proceso de compilación; *devel*, destino de todos los objetos y ejecutables resultantes de la compilación; y *src*, que será el lugar donde escribiremos todo el código, y la única carpeta que manejaremos a partir de ahora. Dentro de esta, crearemos los diferentes paquetes. Estos son la unidad básica de compilación para *catkin*. Cada vez que compilemos en el *workspace*, *catkin* mirará uno a uno cada paquete y lo compilará individualmente, lo que nos permite tener todos los paquetes que queramos.

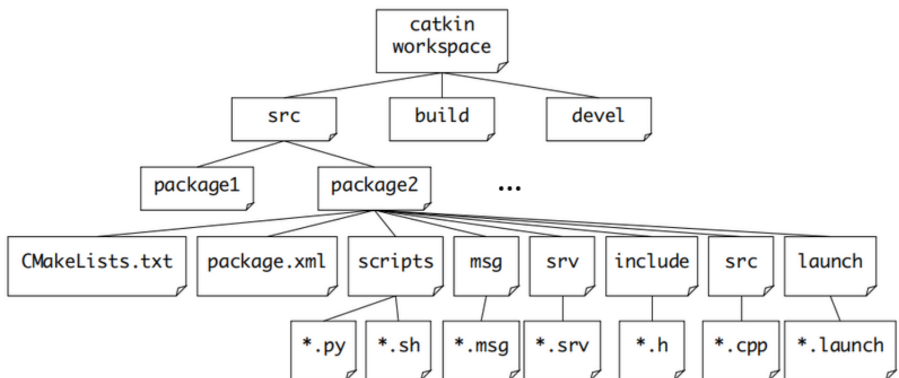


Figura 2-5. Ejemplo de la jerarquía de archivos dentro del *workspace* (Develop Paper, 2020).

En cada paquete siempre tendremos dos archivos fundamentales: *CMakelists.txt* y *package.xml*. En el primero se establecen todas las reglas para la compilación, así como la carpeta destino y las distintas dependencias, como por ejemplo mensajes, servicios o *plugins* creados por el usuario. En el segundo archivo encontramos todas las propiedades del paquete, como por ejemplo el nombre, el número de versión, el autor o las licencias.

Un paquete con los dos archivos anteriores es lo más básico posible. Para dotarlo de cierta funcionalidad, en paralelo a ellos añadiremos diferentes carpetas según la finalidad de los archivos que contengan. En el ejemplo de la Figura 2-5 encontramos carpetas para alojar código y librerías (*src*, *scripts* e *include*), archivos de lanzamiento de la ejecución (*launch*) y otros elementos creados (*msg* y *srv*).

El tipo y número de carpetas no tiene que ser el mencionado anteriormente, simplemente son criterios básicos para estructurar el paquete, y esto variará según las necesidades del proyecto. En el presente trabajo se ha trabajado con un *workspace* llamado *TFM*, y con un único paquete llamado *ptcplant*. La estructura del paquete de trabajo se puede visualizar de una forma gráfica con la orden `tree` en el terminal (ver Figura 2-6).

```
edumbriz@edumbriz-GT70-2PC:~/TFM/src/ptcplant$ tree -L 1 --dirsfirst
.
├── config
├── launch
├── rviz
├── scripts
├── src
├── urdf
├── worlds
├── CMakeLists.txt
└── package.xml

7 directories, 2 files
```

Figura 2-6. Organización del paquete *ptcplant*.

Además de algunas carpetas que ya aparecían en el ejemplo mencionado anteriormente, las cuales comparten la misma función, existen otras nuevas que desarrollan funciones básicas para la ejecución de la simulación:

- *Config*. Alberga ficheros de configuración de parámetros para iniciar la simulación, como por ejemplo el valor de las componentes del controlador PID. Estos archivos son de formato *.yaml*, y son requeridos por los archivos de lanzamiento ubicados en *launch*.

- *RViz*. Se trata de una herramienta básica de visualización en 3D para ROS, en la que podemos comprobar la dinámica del robot diseñado entre otras cosas. En este caso, en esta carpeta se guardan archivos de configuración para la visualización de la herramienta, de forma que al abrir *RViz* aparezca con el aspecto deseado.

- *Src*. En esta carpeta alojamos todos los *plugins* desarrollados por el propio usuario, si es que es necesario desarrollarlos, así como otro tipo de código fuente. En este caso, aquí se han colocado algunos modelos

SDF necesarios para la simulación, así como la descripción de ciertos materiales y texturas para dichos modelos.

- *Urdf.* Aquí se ubican todos los modelos URDF creados, así como todos los archivos necesarios para definirlos, como por ejemplo los mallados en 3D.
- *Worlds.* Contiene los archivos encargados de describir el mundo de la simulación en Gazebo, como las físicas, las características ambientales o la escena, entre otras muchas cosas.

2.3 Comandos principales

A la hora de instalar ROS, están disponibles para su uso una serie de comandos con diferentes finalidades. Algunos son opcionales y se utilizan para ahorrar tiempo, como los de navegación. Sin embargo, hay otros de uso obligado, como los de ejecución. A continuación, se describirán algunos de los que se consideran más útiles.

2.3.1 Comandos de navegación

En general, los comandos de navegación permiten realizar acciones a los paquetes ROS sin necesidad de saber su ruta exacta en el equipo. Algunos de los más importantes son:

- `roscd`: Funciona de una forma similar al comando `cd` que se usa en sistemas Unix, con la diferencia de que basta especificar el nombre del paquete en lugar de la ruta para cambiar a dicho directorio. Existen otras funcionalidades, como por ejemplo escribir el comando sin argumentos, que nos llevará a la raíz de ROS, cuya ubicación es descrita por la variable de entorno `$ROS_ROOT`.
- `rosed`: Al igual que el comando anterior, ambos forman parte de una suite llamada *rosbash*. En este caso, este comando permite editar un archivo cualquiera dentro de un paquete, si se usan como argumentos del comando el paquete seguido del archivo. El programa definido en la variable de entorno `$EDITOR` será el que permita realizar las modificaciones, mientras que si no hay ninguno definido se usará *vim* por defecto.

2.3.2 Comandos para obtención de información

Los siguientes comandos son de gran utilidad cuando queremos obtener información de los elementos que se encuentran activos durante la ejecución.

- `rosnode`: Nos proporciona información detallada sobre los nodos de ROS que se encuentran en ejecución. Cuenta con diversos *flags* para diferentes funciones. Entre los más importantes destacan `info`, que nos imprime por pantalla información de los nodos en activo; `kill`, que permite matar el proceso que corre el nodo; o `list`, que muestra una lista detallada de todos los nodos en activo.
- `rostopic`: Similar al anterior, pero en lo referido a los *topics* en lugar de nodos. Principalmente, podemos extraer información con diferentes *flags* que vimos antes como `info` o `list`, y también otros como `echo`, para que imprima por pantalla el contenido del *topic*. Por otro lado, un comando fundamental para realizar cambios en tiempo real es `rostopic pub`. Usando como argumentos el *topic* requerido, el tipo de mensaje que se publicará y el contenido de dicho mensaje, se cambiará de forma dinámica el contenido de dicho *topic*. Esto es básico para, por ejemplo, establecer manualmente la posición o velocidad de un modelo. La publicación puede ser única, si añadimos como primer argumento el *flag* `-1`, o bien discreta, si indicamos la frecuencia de publicación deseada.
- `rosmsg` y `rossrv`: Actualmente, ambos comandos forman parte de la librería de `rosmsg`, y su uso es prácticamente idéntico para mostrar información de mensajes y servicios, respectivamente. Nos serán de gran utilidad sobre todo cuando dichos mensajes y servicios hayan sido creados por el usuario, aunque también pueden usarse para tipos predefinidos, sobre todo en mensajes. Los *flags* fundamentales son `package`, para ver los mensajes en un paquete; `list`, para verlos en todos los paquetes; y `show`,

que muestra en detalle la estructura de un mensaje concreto.

- `rospack`: Se usa para obtener información sobre los paquetes ROS disponibles en el sistema, desde listados, hasta estructuras de dependencia entre los diferentes paquetes.

2.3.3 Comandos de ejecución

Estos comandos se utilizan para comenzar la ejecución, así como para iniciar nuevos ejecutables una vez que ROS se está ejecutando.

- `roscore`: Al ejecutar este comando, lanzamos un conjunto de programas que se encargan de inicializar todo el ecosistema de ROS. Más allá de lanzar el *Master*, estrictamente necesario para que los nodos puedan comunicarse entre ellos, también se encarga de iniciar el servidor de parámetros y el nodo *rosout*, encargado de llevar el registro de las ejecuciones.
- `rosrun`: Nos permite lanzar un ejecutable sin tener que navegar de forma manual hasta la ruta que lo contiene. Basta con añadir como argumentos el paquete y el nombre del archivo para ejecutarlo.
- `roslaunch`: Este comando es de los más útiles. Permite lanzar una simulación mediante la ejecución de un archivo de extensión *.launch*, escrito en formato XML. En dicho archivo podemos realizar diversas tareas, entre las que se podrían destacar el ajuste de los valores iniciales del servidor de parámetros, la posibilidad de lanzar múltiples nodos al mismo tiempo, o poder anidar otros archivos de lanzamiento creando lanzamientos anidados. Además, al ejecutar este comando, automáticamente se inicializan todos los programas que lo hacen cuando ejecutamos `roscore`, por lo que no es necesario usar este último comando si utilizamos `roslaunch`.

2.3.4 Comandos de herramientas gráficas

Mediante los siguientes comandos se obtiene la posibilidad de dar a la información un aspecto visual, que ayudará a verla de una forma mucho más rápida y eficiente. Los más destacados son:

- `rqt_graph`: Lanza una interfaz gráfica con todos los nodos y *topics* que encuentre en ejecución, de una forma ordenada, y muestra de una forma sencilla el flujo de la información. Es posible refrescar la interfaz para ir viendo los cambios en tiempo real, así como modificar diversas opciones en función del contenido que se desee ver.
- `rqt_plot`: Nos permite visualizar en una interfaz visual el contenido de uno o varios *topics* a lo largo del tiempo. De esta forma, se pueden elaborar gráficas de datos en tiempo real, para analizar y comparar datos de diferentes *topics*. En concreto, a lo largo de este trabajo esta herramienta ha sido muy útil para tunear los PID de los ejes de giro, ya que se puede ver la respuesta del eje ante una entrada determinada a medida que cambian los parámetros del controlador en tiempo real.

2.3.5 Otros comandos

- `rosclean`: Se usa para limpiar archivos del entorno de ROS, como por ejemplo los registros. También nos proporciona información sobre el espacio usado en disco, si se ejecuta con el *flag* `check`. Por último, con el *flag* `purge` borraremos los registros acumulados, siempre que confirmemos que los archivos a borrar son los deseados y evitemos pérdidas irreparables.
- `rosversion`: Nos dice la versión de ROS sobre la que se está trabajando.

2.4 Instalación y configuración de ROS

Como se ha comentado en puntos anteriores, a la hora de elegir qué versión de ROS se debe usar hay que tener en cuenta también la del sistema operativo. En el actual trabajo, y debido a la versión de Ubuntu instalada, lo más adecuado ha sido usar ROS *Melodic*. Ya no solo por ir en cierto modo emparejada a dicha versión del sistema operativo, sino porque también tiene cierto bagaje y uso en la comunidad. Esto será siempre un punto a favor a la hora de la resolución de posibles errores, consulta de dudas o recepción de soporte en la web.

2.4.1 Instalación

Aunque se podría realizar una instalación descargando el código fuente y compilándolo, esto tiene cierta complejidad y no se recomienda hacer. Normalmente, es algo a evitar a no ser que se le dé un uso a nivel experto y se dispongan de los conocimientos para realizarlo sin problemas. En este caso, se realizará una instalación directa desde los repositorios.

En primer lugar, hay que asegurarse que la lista de paquetes Debian disponibles está actualizada, para lo que se usa el siguiente comando:

```
>> sudo apt update
```

A continuación, se podrá instalar ROS con diferentes comandos, en función de las librerías y herramientas complementarias que queramos tener disponibles. Podemos optar por una opción básica, sin herramientas entre usuario-interfaz; una opción de escritorio, que complementa a la anterior instalando herramientas como *rqt*, *RViz*, y algunas librerías de robot genéricos; y una opción de instalación total, que también incluye simuladores y librerías de percepción, tanto en 2D como 3D.

En este caso concreto, se ha optado por la opción intermedia, la de escritorio. En primer lugar, esto es debido a que Ubuntu está instalado en una partición del disco duro con un espacio algo reducido. Por otro lado, los complementos que da la versión total no forman parte de los objetivos propuestos en el trabajo, ya que se instalará y usará el simulador Gazebo. En cualquier caso, si fueran necesarias algunas de las herramientas no instaladas en un comienzo, siempre se pueden instalar a posteriori individualmente.

Para instalar la versión elegida, se usa:

```
>> sudo apt install ros-melodic-desktop
```

Tal y como se ha comentado anteriormente, si fuera necesaria la instalación de una herramienta complementaria fuera del paquete por defecto, se podría instalar de la misma forma tras buscarla con el siguiente comando:

```
>> apt search ros-melodic
```

Tras esto, ROS quedaría instalado.

2.4.2 Configuración del entorno de trabajo

Una vez realizada la instalación, se deben estructurar los directorios en los que se trabajará. En primer lugar, es necesario crear una nueva carpeta en el directorio que se quiera, la cual será el *workspace*, y se llamará *TFM*. A su vez, dentro de esta, es necesario crear otra carpeta con el nombre *src*. Dentro de esta última, se debe usar el siguiente comando:

```
>> catkin_init_workspace
```

Con este comando se habrá configurado la carpeta *TFM* como *workspace*, y se habrá creado el archivo *CMakeLists.txt* de dicho entorno. Este último se enlazará con *catkin*, el sistema de compilación.

A continuación, se lanza el siguiente comando para compilar todo el *workspace* (ver Figura 2-7). Este ejecuta un conjunto de órdenes de compilación en cadena, y crea las carpetas necesarias en caso de que estas no existan. Esto sucederá únicamente en la primera compilación. El comando es:

```
>> catkin_make
```

Con esto, el entorno de desarrollo queda inicializado. A partir de aquí, se habrán creado las carpetas *build* y *devel*. Esta última contiene el archivo *setup.bash*, al que habrá que hacer `source` cada vez que queramos inicializar ROS en un nuevo terminal.

```
edumbriz@edumbriz-GT70-2PC:~/TFM/src$ catkin_init_workspace
Creating symlink "/home/edumbriz/TFM/src/CMakeLists.txt" pointing to "/opt/ros/melodic/share/catkin/c
make/toplevel.cmake"
edumbriz@edumbriz-GT70-2PC:~/TFM/src$ cd ..
edumbriz@edumbriz-GT70-2PC:~/TFM$ catkin_make
Base path: /home/edumbriz/TFM
Source space: /home/edumbriz/TFM/src
Build space: /home/edumbriz/TFM/build
Devel space: /home/edumbriz/TFM/devel
Install space: /home/edumbriz/TFM/install
####
#### Running command: "cmake /home/edumbriz/TFM/src -DCATKIN_DEVEL_PREFIX=/home/edumbriz/TFM/devel -D
CMAKE_INSTALL_PREFIX=/home/edumbriz/TFM/install -G Unix Makefiles" in "/home/edumbriz/TFM/build"
####
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
```

Figura 2-7. Inicialización y compilación del *workspace*.

Como ya se comentó en el capítulo 2.2, para evitar tener que ejecutar el *setup.bash* en cada nuevo terminal que se ejecute, sería conveniente añadir las líneas pertinentes al archivo *.bashrc*. Hay que tener en cuenta que, si por cualquier motivo hay más de una versión de ROS instalada, en dicho archivo sólo podemos hacer referencia a la versión que se vaya a usar.

A continuación, sólo queda crear tantos paquetes como se desee. Para ello, basta con hacer `cd` a la carpeta *src* y ejecutar el comando `catkin_create_pkg <nombre> <dependencias>`. En el argumento correspondiente a `<nombre>`, se especificará el nombre del paquete que se creará. En `<dependencias>`, se debe hacer referencia a toda dependencia que se quiera añadir. Por ejemplo, en el caso del paquete creado para el presente proyecto, se desea tener la posibilidad de compilar tanto en Python como C++. Por ello, las dependencias serán las librerías `rospy` y `roscpp`, respectivamente. Por tanto, el comando final a ejecutar en este caso sería el siguiente:

```
>> catkin_create_pkg ptcplant rospy roscpp
```

Se podría realizar este proceso tantas veces como paquetes se deseen crear. Cada uno podrá tener nombre y dependencias diferentes, y se compilarán separadamente, dado que cada paquete es la unidad mínima de compilación para *catkin*.

A partir de este momento, solo quedaría comenzar a agregar carpetas y archivos al paquete para dotarlo de funcionalidad. Aunque esto se puede hacer de diversos modos, lo normal es usar un IDE por simplicidad y comodidad. Al trabajar en ROS se usarán diferentes formatos de archivos, por lo que resulta apropiado poder aprovechar las ventajas de dicho software, en lugar de modificar archivos con cualquier editor de texto plano.

Durante el presente trabajo, se ha trabajado usando el IDE Eclipse. El primer paso es crear un proyecto nuevo, vacío, el cual asociaremos a la ruta del *workspace* creado. En segundo lugar, configuramos el editor para que reconozca el lenguaje deseado según la extensión del archivo con el que se trabaje. También es posible instalar complementos que por defecto no se instalan con Eclipse, como el editor de Python.

3 GAZEBO

Gazebo es un software de código abierto enfocado a la simulación robótica, que cuenta con el motor de físicas ODE, entre otros, y un renderizado mediante OpenGL. En sus inicios, entre 2004 y 2011, se desarrolla como una parte de *Player Project*, destinado a crear software libre para la investigación en robótica y sistemas de sensores. Posteriormente, siguiendo un camino similar al de ROS, pasa a ser un proyecto independiente desde que *Willow Garage* comienza a darle soporte. Posteriormente, será la OSRF la encargada de mantener el proyecto de Gazebo.

Hoy en día, Gazebo cuenta con la posibilidad de usar diferentes motores para las físicas, más allá del usado por defecto, ODE. Soporta la creación de diferentes entornos ambientales, renderizando de una forma realista sombras, iluminación y texturas. Permite la creación de modelos realmente complejos, así como la interacción entre ellos mediante plugins para controlar sensores, actuadores o cámaras.

En los últimos años, desde su consideración como proyecto independiente, son diversos los concursos y competiciones que se han celebrado utilizando a Gazebo como base. Entre ellos, los más conocidos son la *DARPA Robotics Challenge* (2012-2015), *NASA Space Robotics Challenge* (2016-2017) o la *Toyota Prius Challenge* (2016-2017) (Wikipedia, 2021).

3.1 Instalación de Gazebo

A diferencia de ROS, la instalación de Gazebo y su configuración se realiza de una forma mucho más sencilla. El primer hecho para tener en cuenta es que la versión de Gazebo sea acorde a la que se haya instalado de ROS. El desarrollo de ambas plataformas es separado, sin embargo, no todas las versiones son compatibles y es recomendable seguir las instrucciones de instalación que nos proporciona Gazebo según la versión de ROS instalada.

En el caso del presente trabajo, al haber instalado previamente ROS Melodic, es necesario instalar la versión 9.0 de Gazebo. Al existir versiones posteriores, las cuales no interesa instalar porque habría problemas de compatibilidad con ROS, es necesario cerciorarse de que la versión instalada es la correcta. Así, se debe evitar la instalación rápida de Gazebo, ya que usaría la versión más reciente. Es necesario realizar una instalación personalizada donde indiquemos la versión requerida.

Para ello, se instalará Gazebo desde el repositorio de la OSRF, el cual se configura con los siguientes comandos:

```
  >> sudo sh -c 'echo "deb
http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -
cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
  >> wget https://packages.osrfoundation.org/gazebo.key -O - | sudo
apt-key add -
```

Con ello, se ha configurado el PC para que acepte la instalación de paquetes de dicho repositorio. A partir de aquí instalar la versión deseada de Gazebo, previa actualización de la base de datos de Debian, y por último comprobar que el software funciona correctamente. Para ello se introduce en el terminal:

```
>> sudo apt-get update
>> sudo apt-get install gazebo9
>> gazebo
```

Si todo ha salido bien, no habrá ningún mensaje de error, y tras ejecutar el último comando se podrá visualizar la interfaz de Gazebo tal y como se aprecia en la Figura 3-1.

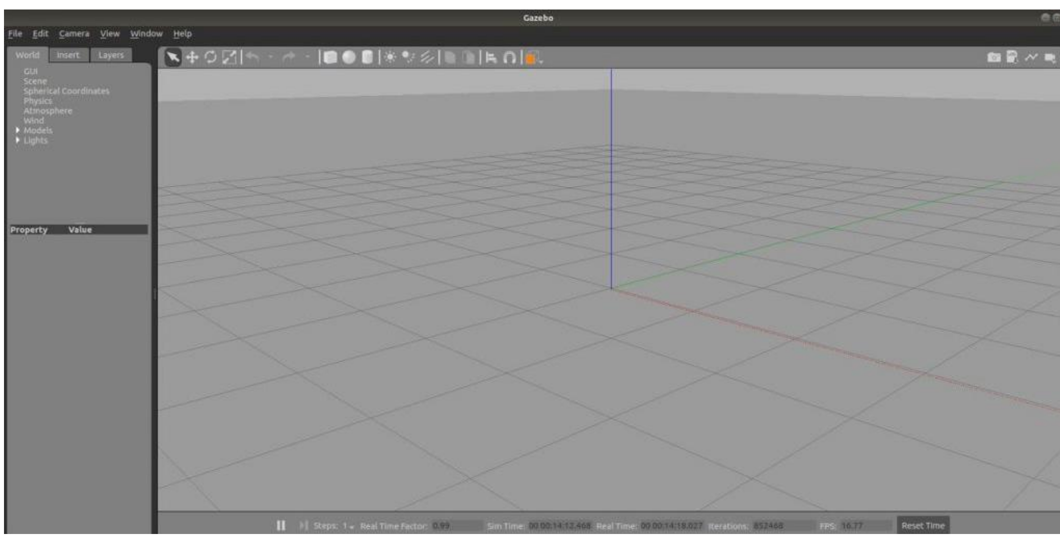


Figura 3-1. Interfaz de Gazebo tras la primera ejecución.

Una particularidad de lanzar Gazebo del modo anterior es que el comando usado realmente lanza dos comandos distintos. Por un lado, lanza `gzserver`. Podríamos considerarlo como la parte *servidor*, y se encarga de procesar y simular las físicas, sensores o renderizado entre otras tantas cosas. Por otro, también corre el comando `gzclient`, el cual proporciona la interfaz gráfica para visualizar la simulación, así como controles para intervenir en algunos aspectos de la simulación. Se comunican entre ellos mediante diferentes librerías de Gazebo, y pueden ser lanzados por separado desde el terminal mediante el uso de cada comando asignado.

A esto se debe sumar la existencia de ROS, y el hecho de que también debe estar conectado tanto con el servidor como con el cliente de Gazebo si queremos realizar una simulación conjunta. La mejor opción para lanzar una simulación en la que todos estos puntos se interconecten es mediante el uso del comando `roslaunch` mencionado en el capítulo 2.3.3, como se verá en el siguiente punto.

3.2 Simulaciones conectando Gazebo y ROS

Como se ha comentado en capítulos anteriores, tanto ROS como Gazebo son independientes entre sí, pudiendo cada uno desarrollar diferentes funciones sin la necesidad del otro. Sin embargo, también existe la posibilidad de usarlos en conjunto, de forma que Gazebo dé una representación visual de todo lo programado en ROS y viceversa, obteniendo en ROS información extraída de la simulación de físicas en Gazebo. Para ello, es necesario lanzar ambos en conjunto, y el mejor modo es mediante el comando `roslaunch` y un archivo de extensión *.launch*, el cual se analizará a continuación en base al ejemplo mostrado en la Figura 3-2.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<launch>

    <arg name="robot" default="machines"/>
    <arg name="debug" default="false"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="pause" default="false"/>

    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find ptcplant)/worlds/simple.world"/>
        <arg name="paused" value="$(arg pause)"/>
        <arg name="use_sim_time" value="true"/>
        <arg name="gui" value="$(arg gui)"/>
        <arg name="headless" value="$(arg headless)"/>
        <arg name="debug" value="$(arg debug)"/>
    </include>

</launch>
```

Figura 3-2. Ejemplo de archivo *.launch* básico.

Los archivos de extensión *.launch* son escritos en lenguaje XML. En la Figura 3-2 se aprecia un ejemplo muy básico, que nos permite lanzar ROS y Gazebo al mismo tiempo. Suponiendo que dicho archivo se llamara *basico.launch*, y que perteneciera al paquete *ptcplant*, para ejecutarlo basta con escribir en el terminal el siguiente comando:

```
>> roslaunch ptcplant basico.launch
```

En este instante se podrá observar cómo aparece el cliente de Gazebo cargando un mundo. A la vez, en el terminal se verá cómo se ha ejecutado implícitamente la orden `roscore` tras ejecutar el comando especificado. De esta forma, se inicializa ROS, y Gazebo se ejecuta como un nodo más de todos los que pueden existir en el ecosistema. Para tener una idea más gráfica de los nodos que se ejecutan, se puede usar el comando `rqt_graph` en un nuevo terminal (Figura 3-3).

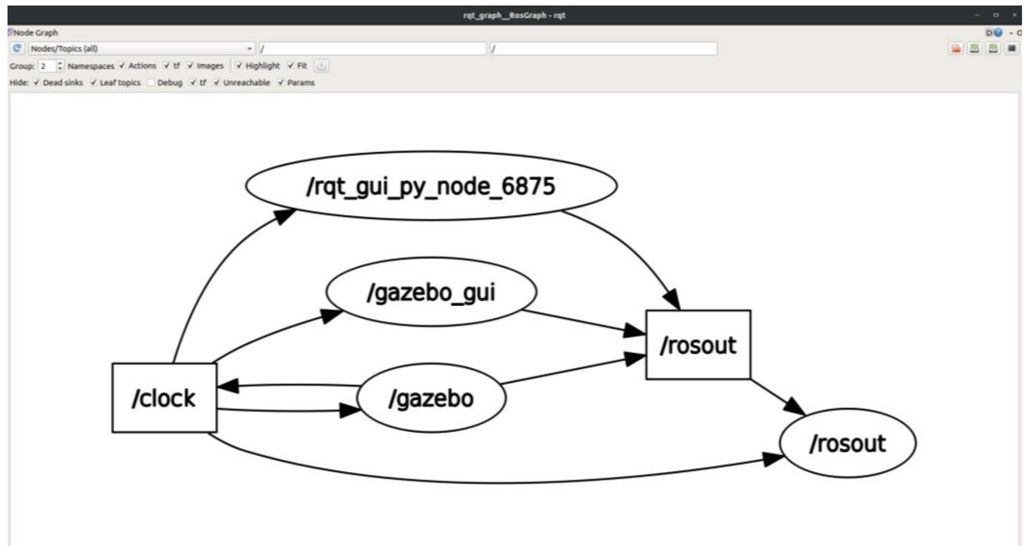


Figura 3-3. Resultado de *rqt_graph* tras ejecutar el archivo *.launch* básico (nodos en elipses, *topics* en rectángulos).

Si se analiza el archivo *basico.launch*, diferenciamos dos partes dentro de las etiquetas `<launch>`. En primer lugar, existen una serie de argumentos a los que se les especifican ciertos valores por defecto. Todos estos argumentos pueden ser también definidos en la línea de comandos, al lanzar el comando `roslaunch`. Tienen diversas funciones según el valor booleano que adopten: `gui` cargará la interfaz gráfica; `headless` lanzará la simulación sin cliente, solo con servidor; y `pause` comenzará la simulación en pausa.

La segunda parte es la comprendida entre las etiquetas `<include>`. Se usa dicha etiqueta cuando queramos anidar otro archivo *launch* y que sea ejecutado al alcanzar esta línea. En esta ocasión, se lanza el archivo *empty_world.launch*, el cual es un archivo predefinido en el paquete *gazebo_ros* que lanza un mundo totalmente vacío, como el que se aprecia en la Figura 3-1. Como se está lanzando otro archivo distinto, también se especifican una serie de argumentos que modifican el comportamiento del segundo archivo lanzado, *empty_world.launch*.

Como se podrá observar, en algunos de estos argumentos no se da un valor booleano, si no que se realiza una referencia al valor de otro argumento anterior. Por ejemplo, en el argumento `headless`, en lugar de especificar un valor, se le dice que use el mismo que se especificó en el argumento del primer archivo *.launch*, de forma que se crea una dependencia. Esto se hace mediante la línea `value="$(arg headless)"`.

La línea más importante de las presentadas es la que define el argumento `world_name`. En este caso, en lugar de asignar un valor booleano, se especifica la ruta de un archivo *.world*. Estos archivos, basados en SDF, definen el mundo que cargará Gazebo. Si no se especificara ninguno, cargaría un mundo vacío sin suelo, por lo que si se añadieran modelos caerían al vacío. Para evitar esto, se ha creado el archivo *simple.world*, que contiene un mundo que sólo dispone de suelo y Sol. Como vemos en la Figura 3-2, se usa la orden `$(find ptcplant)` para que busque el mundo en dicha ruta, y posteriormente se indican las carpetas donde se encuentra y el nombre y extensión del archivo.

A partir de este punto, bastaría con comenzar a diseñar un mundo para el entorno de la simulación, así como incluir los diferentes modelos requeridos. Por otro lado, en el archivo *.launch* se podrán añadir líneas con diversas funciones, desde anidar otros archivos de lanzamiento, hasta otras para crear nodos en ROS a partir de librerías.

3.3 Creación de mundos

En Gazebo, un mundo es un archivo de extensión *.world* que define la descripción, condiciones y propiedades del entorno donde se desarrolla la simulación. Principalmente, se recogen características como modelos, la escena, las físicas o los plugins, aunque también se podría incidir en aspectos secundarios como las condiciones atmosféricas, la gravedad o el campo magnético. Como se menciona en los puntos anteriores, generalmente este archivo será cargado en Gazebo al inicio de la simulación. Posteriormente, se pueden modificar algunos aspectos, incluso añadir modelos descritos en otros archivos.

En el presente trabajo se ha usado el mundo para describir dos bloques de elementos. Por un lado, los modelos estáticos, como pueden ser la zona de potencia, los caminos y el suelo, así como los materiales y texturas que los definen. Por otro, se definen parámetros abstractos como la escena (colores ambientales, sombras, nubes), la iluminación, la posición inicial de la cámara, o las físicas. Este último factor resulta de gran importancia, ya que es necesario ajustar sus componentes para optimizar el rendimiento de la simulación.

Para comenzar un mundo nuevo, lo ideal es obtener una plantilla proporcionada en la web de Gazebo, que contiene únicamente suelo y Sol, desde la cual partir y comenzar a modificar y añadir el resto de los elementos. Otra opción sería abrir el cliente de Gazebo y guardar desde la aplicación el mundo en el lugar deseado. En cualquier caso, esto sienta una base. Sin embargo, para seguir describiendo el mundo podremos optar, a grandes rasgos, por dos opciones: añadir y editar objetos en el cliente e ir guardando, o modificar el archivo en formato XML usando la descripción SDF.

3.3.1 Modificación manual

En primer lugar, se puede optar por realizar modificaciones manualmente en el cliente. Gazebo tiene un modo edición que permite añadir y modificar objetos en tiempo real, lo que está muy bien para realizar pruebas, ya que podemos ver los diferentes modelos añadidos en la misma simulación, sin necesidad de reiniciarla.

Por el contrario, este método solo da opción a crear modelos muy básicos, ya que solo pueden usarse formas primitivas como esferas, cubos o cilindros. También permite introducir otros elementos, como articulaciones. Además, existe una amplia base de datos con modelos complejos que se pueden añadir a la simulación con un solo clic.

Se podría decir que este es el mejor método para dotar al modelo de una estructura básica aproximada, que luego podría afinarse mucho más cambiando el código fuente del archivo. Es por ello por lo que, generalmente, se usa la descripción SDF en lugar de realizar modificaciones manualmente en el cliente.

3.3.2 Modificación en *scripts* SDF

La configuración de los archivos *.world* también puede realizarse modificando el código del propio archivo, el cual se escribe en SDF, un formato del lenguaje XML. En general, se usa SDF para describir mundos, así como modelos. Se podría optar por describir un modelo en el mismo archivo del propio mundo. Sin embargo, si dicho modelo fuera de alta complejidad se podría optar por describirlo en un archivo separado y posteriormente incluirlo referenciando el archivo y su ruta. Esta última opción es la mejor de cara a generar varios modelos iguales.

```
<physics name="fisicas_agiles" type="ode">
  <max_step_size>0.005</max_step_size>
  <real_time_factor>1</real_time_factor>
  <real_time_update_rate>200</real_time_update_rate>
  <ode>
   <solver>
     <type>quick</type>
     <iters>50</iters>
     <sor>1.3</sor>
   </solver>
  </ode>
</physics>
```

Figura 3-4. Apartado de las físicas en el archivo *.world*.

Como se mencionaba anteriormente, aparte de definir modelos, en el archivo *.world* introduciremos aspectos relacionados con las físicas de la simulación, la escena o la iluminación. En la Figura 3-4 se pueden apreciar algunos de los valores estipulados para las físicas configuradas en el presente trabajo. En primer lugar, es necesario dar un nombre único, y por otro lado especificar el tipo de motor que se usará. En este caso es ODE, el usado por defecto.

Los valores `max_step_size`, `real_time_factor` y `real_time_update_rate` son críticos para hacer que la simulación sea fluida. El primero de ellos corresponde al tiempo máximo en el que un objeto de la simulación puede interactuar con el mundo. El segundo representa el valor objetivo para el cociente velocidad de simulación entre velocidad real, el cual se establecerá en uno siempre que se quiera alcanzar una velocidad de simulación al menos igual a la realidad. El último valor es el ratio al que se actualizan las físicas del motor.

Se ha encontrado una mejora fundamental a la hora de modificar los parámetros primero y tercero. El valor por defecto de la actualización de las físicas era una frecuencia de 1000 Hz, lo que provoca un rendimiento muy bajo en simulaciones cargadas de elementos, sobre todo en computadoras cuyo hardware no es el más actualizado. Por otro lado, si se quieren obtener resultados lógicos sin alterar el factor de tiempo real, es necesario

mantener constante el producto entre el primer y tercer parámetro. De ahí que los valores hayan cambiado a 0.005 (originalmente 0.001) y 200.

```
<scene>
  <ambient>0.4 0.4 0.4 1</ambient>
  <background>0.7 0.7 0.7 1</background>
  <shadows>0</shadows>
<sky>
 <clouds>
     <speed>7</speed>
     <humidity>5</humidity>
     <mean_size>0.3</mean_size>
   </clouds>
 </sky>
 <grid>0</grid>
 <origin_visual>0</origin_visual>
</scene>
```

Figura 3-5. Apartado de la escena en el archivo *.world*.

Por otro lado, ya mirando dentro de las etiquetas `<ode>`, se ha conseguido una mejora de rendimiento estableciendo el tipo de *solver* en rápido, así como reduciendo el número de iteraciones, especificado en el campo `<iters>`. Tras realizar todos los cambios mencionados, se ha observado que la simulación mejora sustancialmente, tanto en procesamiento como en número de FPS mostrados por el cliente. La configuración inicial era muy exigente, y aunque puede ser útil en otras aplicaciones, en un caso con movimientos lentos como el que se estudia no es necesario exigir tanto.

Algo similar sucede en los campos asociados a la escena y la iluminación. En el primero (Figura 3-5) se definen los colores de las luces ambiente y del fondo, las sombras, características del cielo y otras propias de la interfaz como los ejes de coordenadas. En el segundo (Figura 3-6), características específicas de la iluminación y la capacidad de producir sombras. Se observa una mejora de rendimiento si se desactivan las sombras en ambos campos, sobre todo cuando la simulación se compone de muchos modelos.

```
<light type="directional" name="sol">
  <cast_shadows>false</cast_shadows>
  <pose>0 0 10 0 0 0</pose>
  <diffuse>0.8 0.8 0.8 1</diffuse>
  <specular>0.2 0.2 0.2 1</specular>
  <attenuation>
    <range>1000</range>
    <constant>0.9</constant>
    <linear>0.01</linear>
    <quadratic>0.001</quadratic>
  </attenuation>
  <direction>-0.5 0.1 -0.9</direction>
</light>
```

Figura 3-6. Apartado de la iluminación en el archivo *.world*.

Si bien a lo largo del presente punto la descripción se ha centrado en físicas, escena e iluminación, esto es porque son los que se han modificado significativamente. Existen otros campos que no se han incluido en la descripción del mundo debido a que sus valores por defecto, es decir, los que toma si estos no se incluyen en el archivo, son válidos para la simulación. Entre ellos se podrían destacar, por ejemplo, algunos que permiten introducir audio, viento o una cantidad de modelos repetidos en un patrón especificado.

1.1.1.1 Descripción de modelos

Como ya se ha comentado, dentro del mundo los modelos pueden ser descritos manualmente, o bien incluidos referenciándolos. Esto variará según las necesidades, si bien para modelos únicos y con una estructura sencilla lo mejor es seguir la primera de las alternativas. A continuación, se realiza una breve descripción de los apartados para tener en cuenta para su descripción.

En primer lugar, cada modelo que se defina debe tener un nombre único de forma obligatoria. También se pueden establecer ciertos parámetros no requeridos pero muy útiles, como pueden ser `<static>`, para hacer el modelo inamovible; `<self_collide>`, para definir si deben existir las colisiones entre diferentes partes de un mismo modelo; o `<pose>`, para definir la posición del modelo mediante un vector de seis componentes.

En paralelo a los anteriores, también se definen los campos `<plugin>`, `<joint>` y `<link>`. El primero será de utilidad cuando queramos que el modelo ejecute código programado en un archivo definido para modificar su comportamiento. Mediante el segundo, `<joint>`, se definen las articulaciones que estarán presentes en el modelo, así como el tipo de articulación, el eje de giro o su posición, entre otros.

Por último, en `<link>` se definen cada uno de los bloques que forman el modelo. Al igual que con las articulaciones, se podrán usar tantos como sean necesarios. Aparte de parámetros complementarios como los mencionados dentro de la etiqueta de modelo, en primer lugar, se deben especificar las inercias. Posteriormente, hay dos grandes bloques que aparecerán en la mayoría de las ocasiones, `<collision>` y `<visual>`.

El primero de ellos indicará la forma y características del bloque de cara a la computación de las físicas y colisiones, mientras que el segundo dará una descripción meramente visual. En ambos campos podremos establecer parámetros como la posición, el mayor número de contactos admitidos (para el bloque de colisiones) o la transparencia (para el bloque visual). Sin embargo, más allá de esto, ambos comparten en su interior el bloque `<geometry>`, que define la geometría del bloque en cuestión.

Es obligatorio definir la etiqueta de geometría. Existe la opción de no representarla, mediante el parámetro `<empty>`, pero se debe especificar. Aquí se define la geometría de diferentes formas. La más sencilla es usando una etiqueta para formas primitivas, como `<box>` para representar una caja, `<sphere>` para una esfera, o `<ellipsoid>` para un elipsoide.

Sin embargo, lo normal es usar geometrías más complejas. Para ello, se usa la etiqueta `<mesh>`. Esta nos permite cargar un modelo 3D diseñado mediante software específico para ello. En el presente trabajo, esto ha sido algo fundamental, ya que ha sido necesario elaborar un modelo propio en Blender de los colectores usados, siendo imposible elaborarlos en detalle usando únicamente formas primitivas.

Los modelos importados pueden tener distintos formatos. En este caso se ha trabajado con los formatos *Collada* (.dae) para las representaciones visuales, y con .stl para las colisiones. Al ser cargados en Gazebo, el primero nos da una representación fiel al modelo, así como los propios materiales definidos en el software 3D. Por el contrario, los archivos STL nos dan una geometría aproximada, sin materiales ni texturas, pero suficiente para describir una geometría de colisiones de cierta calidad.

3.3.2.1 Inclusión de modelos y materiales personalizados

Los modelos también pueden ser descritos de la misma forma a la mencionada anteriormente, y ser guardados en un archivo de extensión *.sdf* para ser importados en un mundo de otra forma. Esto es muy útil cuando se quiere cargar un modelo definido en repetidas ocasiones.

Para comenzar, la importación de modelos en un mundo se realiza mediante la etiqueta `<include>`. Dentro de ella especificaremos obligatoriamente un nombre único para el modelo, así como la ruta donde se debe buscar el archivo a cargar. Opcionalmente, se puede indicar otro parámetro, como por ejemplo la posición.

Algo a tener muy en cuenta es que Gazebo sea capaz de encontrar la ruta especificada en la base de datos de los modelos. Para ello, en el sistema en el que se trabaje, se debe realizar una modificación en la variable de entorno `$GAZEBO_MODEL_PATH`, y añadir a ella la ruta donde se alojarán los modelos que se vayan a incluir. Si esto no se realiza de forma correcta, Gazebo no será capaz de cargar el modelo en la simulación, y además no dará ningún tipo de mensaje de error.

Por otro lado, algo muy utilizado tanto en modelos descritos manualmente como en los incluidos por referencia

es la inclusión de materiales personalizados. A diferencia de los modelos importados en SDF desde un archivo *collada*, el cual introduce en Gazebo el modelo con los materiales y texturas que se le hayan definido en su creación, al crear manualmente modelos básicos estos aparecerán con un color grisáceo por defecto, por lo que es necesario asignarles un material concreto.

Esto se implementa en la etiqueta `<material>`, que irá en paralelo a la geometría, un nivel por debajo de `<visual>`. Al igual que sucede con los modelos, es posible definir explícitamente las características del material en estas líneas, o bien cargarlo a partir de un archivo de extensión *.material*. Si se quiere reutilizar el material en distintos modelos, lo ideal es definirlo en un archivo externo y cargarlo cada vez que sea necesario su uso. Además, proceder de esta forma en lugar de escribir las líneas sobre la geometría posibilita la creación de materiales mucho más complejos.

Para cargar el material, dentro de la etiqueta de `<material>`, definimos otra llamada `<script>`. Dentro de esta, se especifica la ruta donde se aloja el archivo del material, así como un nombre único para este. De esta forma, el objeto con el que se trabaje adoptará el material definido en el archivo *.material*. Este último podrá ser extraído de alguna librería, o bien creado desde cero por el usuario, como es el caso actual.

Los archivos *.material* son scripts basados en OGRE 3D, un motor gráfico de renderizado orientado a la programación de escenas en 3D. Están basados en código C++, y desde la propia web se pueden obtener plantillas para desarrollar materiales conforme a las necesidades del usuario (OGRE Wiki, 2012).

```
material sand_ground
{
  technique
  {
    pass
    {
      lighting on

      ambient 1.0 1.0 1.0 1.0
      diffuse 0.8 0.8 0.8 1.0
      specular 0.0 0.0 0.0 1.0 2.0
      emissive 0.5 0.5 0.5 1.0

      texture_unit
      {
        texture sand.jpg
        filtering trilinear
        scale 0.005 0.005
      }
    }
  }
}
```

Figura 3-7. Material implementado para el suelo en la simulación.

Si se adopta como ejemplo el script creado para desarrollar el material del suelo en el presente trabajo (Figura 3-7), es posible apreciar dos bloques que afectarán severamente al material. En primer lugar, el ejecutado tras `pass`, en el que definimos la iluminación y los colores de sus diferentes componentes, y que dará una característica fundamental tanto en color como en el comportamiento del material ante la luz.

Por otro lado, en el bloque `texture_unit`, se define la textura usada, los filtros y el escalado. Se ha usado una imagen de tierra, que al ser grande debe ser escalada a un tamaño muy inferior. Mediante los valores dados en el campo `scale`, estamos indicando el valor por el que queremos multiplicar el tamaño de la imagen en sus dos componentes vectoriales del plano.

Esta estructura para la creación de materiales es muy básica, aunque suficiente para mostrarse en una simulación de una forma más que decente. Sin embargo, es posible dotar al material de cierta complejidad si esto fuera

necesario, como por ejemplo desarrollar un terreno con varias texturas y degradación para representar un suelo de césped y tierra con diferentes plantas distribuidas aleatoriamente.

4 MODELADO EN 3D CON BLENDER

Debido a la necesidad de incluir en la simulación modelos de cierta complejidad geométrica que no pueden ser desarrollados desde Gazebo o scripts SDF, es preciso el uso de un software dedicado exclusivamente al modelado en 3D. Para ello, a lo largo del presente proyecto, se ha elegido Blender, un software libre y multiplataforma realmente potente enfocado principalmente al modelado, renderizado y creación de gráficos tridimensionales.

A pesar de existir otros programas que podrían cubrir la necesidad de modelado 3D con un aprendizaje mucho más rápido y menos complejo, desde un inicio se optó por Blender para poder añadir a los modelos tantos detalles y mejoras como fueran requeridos. Además, su diversidad de *add-ons* opcionales, así como el apoyo brindado por la comunidad, convierten al software en algo mucho más liviano de usar.

Otro de los motivos por el que se optó por esta alternativa fue la cantidad de formatos en los que es capaz de exportar los modelos, así como la variedad de opciones para hacerlo. Esto, junto al hecho de trabajar con otros softwares en paralelo, hace que sea un punto a favor para evitar problemas de compatibilidad a la hora de cargar los modelos.

4.1 Introducción

La instalación de Blender es realmente sencilla, y no requiere de pasos adicionales para realizar una correcta configuración. Se puede optar por la instalación desde el gestor de aplicaciones en Ubuntu, tal y como se ha escogido en el presente trabajo, por la descarga desde la web oficial del proyecto *(https://www.blender.org/download/)*, o bien mediante *snap* y la ejecución del siguiente comando en un terminal:

```
>> sudo snap install blender –-classic
```

Una vez se haya instalado correctamente el software, bastará con ejecutarlo para poder comenzar a usarlo, ya que no requiere de configuraciones específicas. Para ello, se accederá desde el acceso directo creado, o bien con el siguiente comando en el terminal:

```
>> blender
```

A lo largo del presente proyecto, fundamentalmente se ha usado Blender para modelar mallas manualmente, con el apoyo en ciertos complementos y modificadores que aportan un mayor nivel de detalle al modelo. Así mismo, se han hecho varios *renders* del modelo final con mayor detalle de un módulo, gestionando la iluminación y perspectiva de las cámaras para ello.

Si bien el uso comentado se considera como lo más básico en Blender, este software tiene unas posibilidades muy amplias, tantas como el usuario desee. Partiendo de un modelado básico, existe la posibilidad de realizar animaciones y *rigging*, renderizados en detalle, usar físicas, o realizar esculturas y pintado. Esto lo convierte en un programa usado con frecuencia en el sector artístico, y con un enfoque un tanto más alejado de campos

técnicos, si bien esto no impide su uso en ellos.

Para diseñar los modelos se trabajará con mallas, partiendo desde elementos primitivos que irán tomando forma a medida que se manipulen con las diferentes herramientas que nos proporciona Blender. Existen diferentes modos para modelar:

- Modo objeto. Es el modo de trabajo por defecto cuando iniciamos Blender. Permite la creación, unión o separación de objetos, modificación de formas mediante escalado, extrusión, rotación o traslación, y modificación de capas de color y materiales, entre otras cosas. En general, será el modo en el que se dará una forma básica al modelo.
- Modo edición. Tal y como indica su nombre, sirve para editar la malla de una forma más precisa. Al entrar en este modo, se podrán observar cada una de las subdivisiones del modelo, así como modificarlas para que este adopte la forma deseada. En este modo podremos trabajar sobre tres elementos básicos del modelo: vértices, aristas o caras (ver comparación con modo objeto en Figura 4-1).
- Modo escultura. Es un modo más enfocado a creaciones artísticas, ya que deja de lado los elementos geométricos mencionados anteriormente para permitir la edición con pinceles y brochas, de diferentes formas, opacidad y grosor, que actúan sobre el conjunto de elementos que sean capaces de abarcar según la configuración de la herramienta.

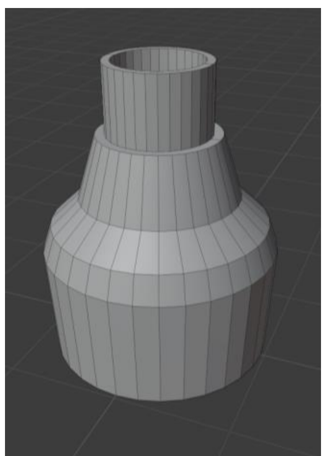


Figura 4-1. A la izquierda, modo edición. A la derecha, modo objeto.

La estructura de una malla está formada por vértices, aristas y caras. Los vértices son el elemento más básico de esta. Son guardados por el programa como un vector de coordenadas en tres dimensiones, y se pueden observar en el modo edición como puntos negros pequeños.

Las aristas son el resultado de unir dos vértices en línea recta, y se definen para crear las caras al unirse varias de ellas. Por su parte, las caras están formadas por un conjunto de aristas que forman un bucle cerrado. Las caras son el único elemento que aparece cuando se renderiza el modelo, a diferencia de vértices y aristas. Además, todas las caras poseen una normal, definida como un vector perpendicular al plano formado por la cara.

Aunque depende del modelo diseñado, la mejor forma de proceder es definir las normales siempre hacia fuera del objeto que se esté creando. Esto es debido a, entre otras muchas cosas, el uso de una propiedad muy útil de las normales, el autosuavizado o *autosmooth*. Si se accede a esta propiedad, podemos definir un ángulo límite entre normales de dos caras contiguas. Todas las caras cuyas normales formen un ángulo inferior al ángulo límite especificado, se unirán con un suavizado automático (Figura 4-2). Sin embargo, es conveniente revisar el suavizado, ya que al ser automático no siempre actuará como se desee, modificando la malla si es necesario para corregirlo en caso de fallo.

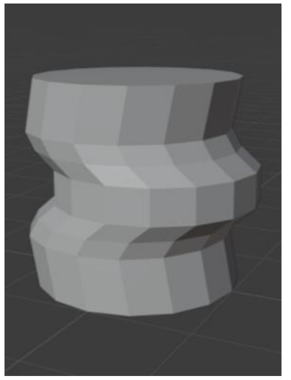
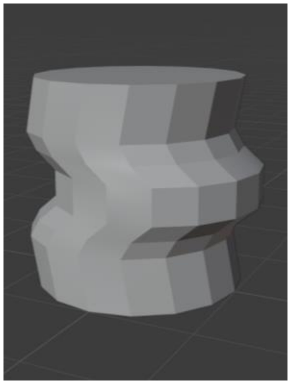
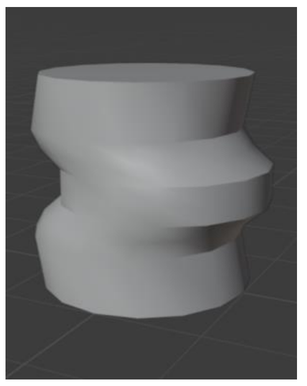
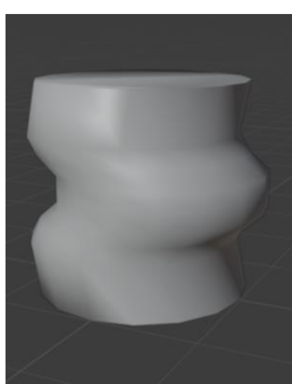
   

Figura 4-2. De izquierda a derecha: *autosmooth* aplicado con ángulos de 0º, 15º, 30º y 90º.

4.1.1 Operadores

Blender proporciona una serie de formas primitivas, como cubos, esferas, conos o cilindros, a partir de las cuales es posible iniciar el modelado. Lo ideal es optar por la que más similitud presente al modelo final, ya que esto ayuda a ahorrar tiempo de trabajo y simplifica las tareas. Sin embargo, realmente se puede partir de casi cualquier primitiva para modelar. De hecho, cada vez que se inicia un nuevo proyecto en Blender, aparece un cubo como primitiva por defecto para modelar.

A la hora de editar la primitiva, Blender ofrece una serie de herramientas básicas llamadas operadores, que permiten modificar el modelo de diferentes formas. Estas funcionarán en modo objeto, edición, o en ambos, y afectarán siempre a los elementos que se hayan seleccionado. Algunas de estas, junto a su atajo por teclado especificado entre paréntesis, son:

- *Move* (G). Permite realizar un movimiento de traslación a la selección. Es posible guiar dicho movimiento si se especifica el eje en el que se quiere hacer el movimiento, pulsando la tecla correspondiente (X, Y o Z) tras ejecutar *Move*. Este bloqueo de ejes se puede usar en muchas de las órdenes que se verán a continuación. Además, se puede colocar el objeto manualmente, o bien especificando una distancia numérica.
- *Rotate* (R). Se utiliza para realizar un movimiento de rotación. Al igual que en el movimiento de traslación, se puede ajustar el eje de rotación, así como los grados a girar. También es posible modificar el centro de la rotación, pudiendo usarse el origen de coordenadas local o global, otros objetos, o el cursor 3D.
- *Scale* (S). Cambia las proporciones de la selección. Se pueden especificar tanto ejes como factores de multiplicación numéricos. Además, dependiendo de si se usa en el modo objeto o en el modo edición, cambiaremos las proporciones del objeto completo o de una(s) arista(s) o cara(s) seleccionada(s), respectivamente.
- *Extrude* (E). Se usa para crear nueva geometría a partir de la existente (Figura 4-3). En general, al usar la extrusión estamos duplicando los elementos seleccionados, que permanecerán unidos a los que ya existían creando una figura geométrica de una dimensión superior. Por ejemplo, como resultado de extruir una arista transversalmente obtendremos un cuadrilátero, mientras que al extruir un vértice obtendremos una arista. Cuando se utilice la orden en una cara poligonal, se conseguirá un poliedro de tantas caras como aristas tenga el polígono original.

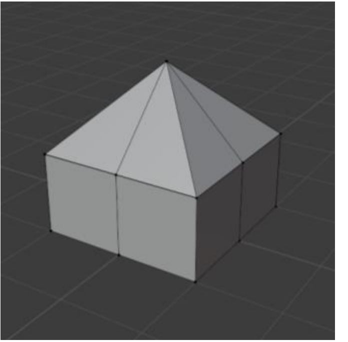
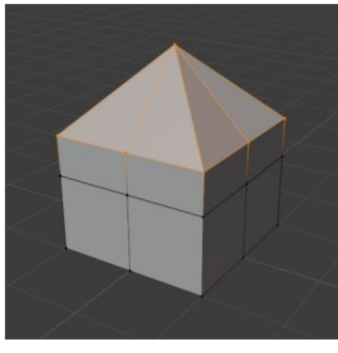
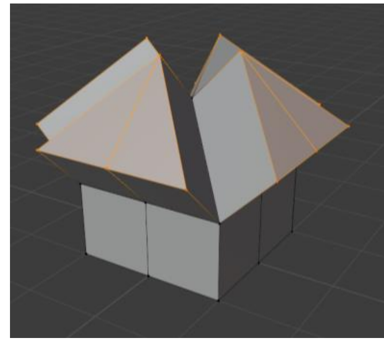
  

Figura 4-3. Aplicación de *extrude*: en el centro, por defecto; a la derecha, *extrude individual*.

- *Inset faces* (I). Permite incrustar caras adicionales al conjunto de caras seleccionadas (Figura 4-4). Por ejemplo, si realizamos esta operación a un cuadrado, se creará otro cuadrado más pequeño dentro de este, conservando el centro. El espacio restante será ocupado por cuatro nuevas caras poligonales con forma de trapecio, resultado de unir los vértices del cuadrado antiguo con el nuevo.

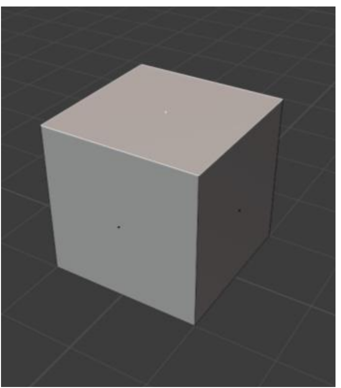
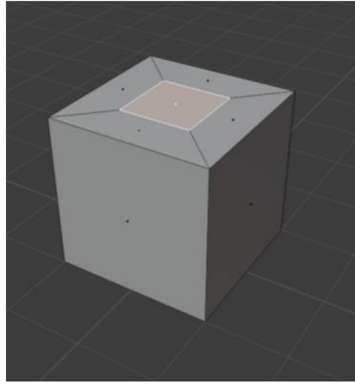
 

Figura 4-4. Aplicación de *inset faces* en la cara superior de un cubo.

- *Bevel* (Ctrl+B). El biselado se encarga de suavizar los bordes de un objeto (Figura 4-5). Se aplica sobre todo en aristas con exactamente dos caras adyacentes, aunque también puede usarse el biselado de vértices. Existen multitud de parámetros para definir el biselado, aunque destacan el ancho, para seleccionar a que altura de la cara comienza el suavizado, y el número de segmentos, que define el número de nuevas caras que se crearán (y, por tanto, lo suave que parecerá el borde).

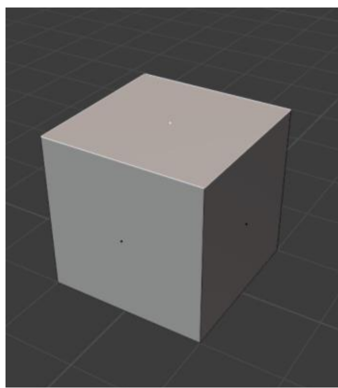
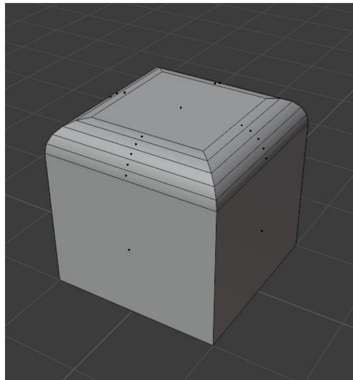
 

Figura 4-5. Aplicación de *bevel* en la cara superior de un cubo.

- *Loop cut* (Ctrl+R). En el modo edición, esta herramienta permite realizar una serie de cortes en la selección para dividir el elemento. Por ejemplo, si se usa en una cara rectangular, podremos ajustar el número de cortes con la rueda del ratón, para luego elegir dónde situarlos moviendo el ratón. Es un operador muy útil para dividir geometría en un determinado número de partes iguales, por ejemplo, para extruir solamente una parte concreta de una cara, como se aprecia en la Figura 4-6.

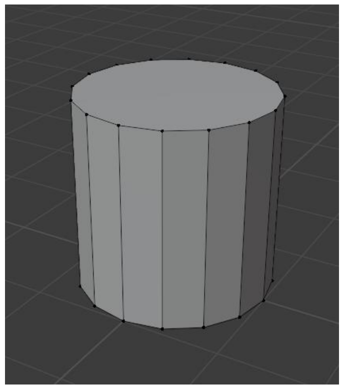
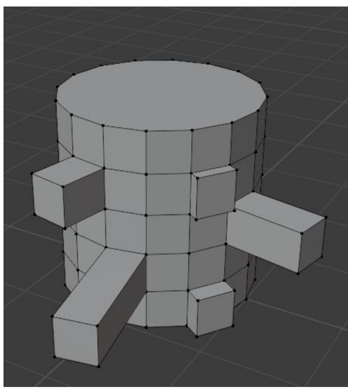



Figura 4-6. Aplicación de *loop cut* en un cilindro para posterior extrusión.

- *Knife* (K). Partiendo desde un punto inicial en una selección, permite realizar cortes manuales en la geometría, en línea recta, a medida que seleccionamos nuevos puntos. Dispone de algunos controles opcionales que facilitan su uso, como por ejemplo para seleccionar el centro de una arista, o para dar cierta inclinación al corte. Sin embargo, es una herramienta que no se recomienda usar al dar diversos problemas como el duplicado de vértices o la creación de cortes inconexos.
- *Spin*. Se usa para extruir una superficie, a la vez que se rota en torno a un determinado punto (Figura 4-7). Por ejemplo, al hacerlo sobre un círculo y extruir en 360º, se creará un poliedro con forma de anillo circular. Es posible ajustar diversos parámetros, como el eje de giro, el número de copias de la figura (y, por tanto, la suavidad resultante) o la fusión automática de vértices. Esta última propiedad, denominada *Auto Merge*, es de gran utilidad, y también aparece en otros operadores. Permite establecer un umbral de distancia, que servirá como límite para fusionar dos vértices cuando entre estos no se supere dicha distancia.

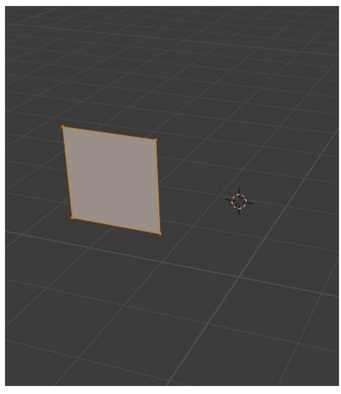
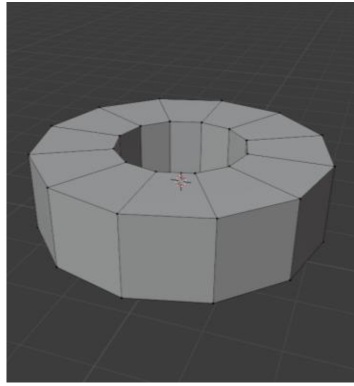



Figura 4-7. Aplicación de *spin* en un cuadrado, usando 12 pasos y 360º de giro.

4.1.2 Modificadores

Los modificadores en Blender se encargan de realizar operaciones automáticas, para sustituir otros procesos que tendrían que resolverse manualmente de una forma muy tediosa. Se pueden añadir varios modificadores a un mismo objeto o elemento geométrico, quedando apilados en lo que Blender llama *The Modifier Stack*.

Una particularidad de los modificadores es que, como es de esperar, el orden en el que se apliquen hace que el producto final cambie. Por ello, es fundamental determinar el orden correcto en la pila de modificadores. Por otro lado, es posible seleccionar si queremos visualizar los cambios que produzcan estos modificadores en los distintos modos (objeto, edición, *viewport* o renderizado). El modificador no aplicará cambios en la selección hasta que no se aplique. Una vez realizada esta operación no se podrá deshacer, y debe ser realizada en la pila de arriba hacia abajo, ya que el programa ignorará los modificadores que no estén aplicados.

Blender posee una gran cantidad de modificadores, aunque en el presente texto se cubrirán los que se consideran fundamentales para operaciones básicas. Se agrupan en cuatro tipos (Blender Reference Manual, 2021):

- *Modify.* Son similares a las herramientas de deformación comentadas anteriormente, aunque no afectan de forma directa a la geometría de los modelos. Más bien, modifican otros datos, como por ejemplo las normales de un grupo de caras o las propiedades de un grupo de vértices.
- *Generate.* Es el grupo con más modificadores, así como el que posee los más usados para un modelado básico. Estas herramientas, tanto constructivas como destructivas, afectan a la topología del mallado de una forma global.
- *Deform.* Afectan también de forma global en el modelo, aunque a diferencia de los anteriores no modifican la topología, únicamente la forma del objeto.
- *Physics.* Enfocados a la simulación de las físicas, no son necesarios en un modelado estático, siendo útiles para otros usos como el modelado de vestimentas, colisiones o fluidos.

Cada modificador suele tener gran cantidad de opciones, que varían según la herramienta usada. Sin embargo, existen una serie de opciones comunes a todos ellos que son de gran utilidad. Entre las más reseñables destacan los grupos de vértices y el uso de texturas.

Los grupos de vértices permiten definir diferentes conjuntos a los que se aplicará, o no, el modificador en cuestión. De esta forma, se pueden disponer mallas que sufrirán diferentes cambios según los grupos de vértices creados dentro de ellas, así como de los modificadores aplicados.

Por otro lado, las texturas son ideales para controlar el efecto de un modificador, actuando como una máscara. El valor entre ceros y unos que toman los píxeles de una imagen en el canal de grises resulta ideal para ponderar el efecto del modificador, actuando la textura como un factor dinámico que multiplica al modificador. Además, en algunos de ellos se podrán aprovechar incluso los canales RGB de la imagen.

Al realizar un modelo como el requerido en el presente trabajo, aun teniendo cierta complejidad geométrica, la mayoría de los modificadores usados son del tipo *Generate*, y puntualmente alguno de tipo *Deform*. Los modificadores del resto de grupos están enfocados a otros objetivos, como puede ser la posterior animación de estos modelos. De entre los usados, a continuación, se comentan algunos de los modificadores más destacables:

- *Array*. Es utilizado para realizar copias de un modelo. Por ello, resulta de gran utilidad para elaborar modelos de cierta repetitividad. Permite determinar el número de copias, así como la distancia entre ellas, relativa o absoluta. También es posible indicar un objeto, como una curva creada por el usuario, a lo largo de la cual deben crearse las copias, como se emplea en la Figura 4-8 para girar los eslabones de la cadena. Esto ha sido de gran utilidad para replicar un módulo y crear un colector a partir de varios módulos.

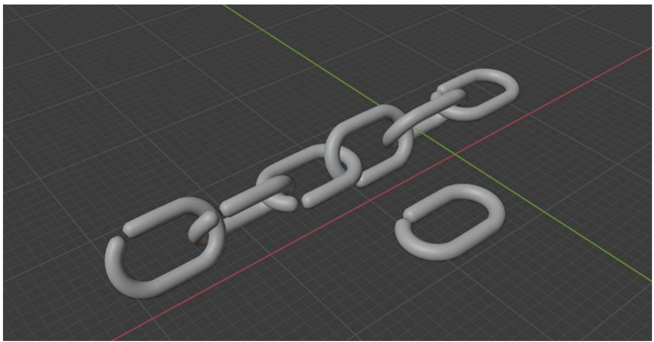


Figura 4-8. Uso de *array* con rotación para crear una cadena a partir del modelo de un eslabón.

- *Boolean*. Permite aplicar operaciones booleanas a dos objetos, como la intersección, la unión y la diferencia entre ellos (Figura 4-9). Por ejemplo, a partir de un cilindro y un cubo, podemos obtener la figura resultante de usar el cilindro como perforador del cubo. En el presente proyecto se ha utilizado este modificador para representar modelos de tuberías y ejes. Sin embargo, es necesario revisar el mallado resultante, ya que la herramienta presenta fallos en algunas mallas más complejas.

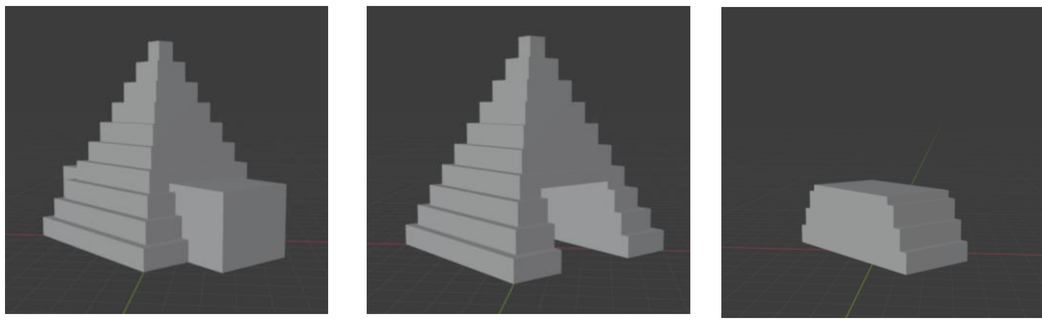


Figura 4-9. De izquierda a derecha, uso de *boolean* con parámetros *union, diference* e *intersect*.

- *Mirror*. Realiza una función de simetría, para duplicar el objeto a lo largo de un eje o plano (Figura 4-10). Cuenta con diversas opciones más allá del eje de simetría para elaborar los nuevos modelos, como por ejemplo el uso de otros objetos como espejo. Esto ha sido una herramienta fundamental en la elaboración del modelo de un lazo, ya que se puede crear por simetría a partir del modelo del colector.

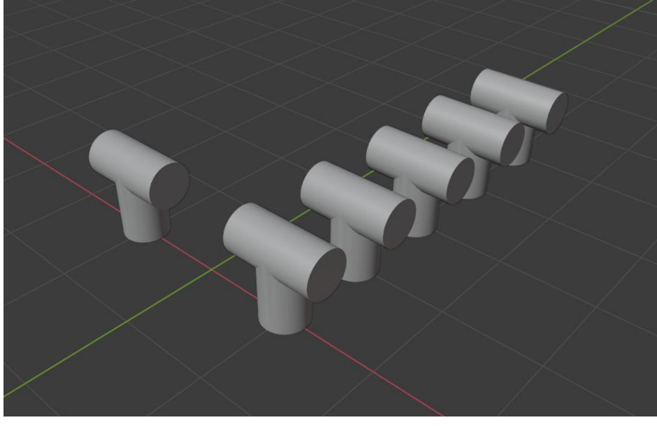


Figura 4-10. Uso de *mirror* sobre el eje en verde, y posterior uso de *array*.

- *Solidify*. Mediante este modificador es posible dotar a una superficie de cierta profundidad (Figura 4-11). Dispone de diversas opciones en el modo complejo, enfocado a tratar mallados muy elaborados. Sin embargo, basta usar el modo sencillo y el espesor deseado para una malla en forma de superficie si esta no es demasiado compleja. A priori, podría pensarse que con el comando de extrusión se podría realizar esta labor. Sin embargo, en ciertas superficies como la creada para la parábola del módulo, lo ideal es aplicar este modificador, capaz de añadir el grosor por la cara de la superficie que no afecta a las propiedades de reflexión de rayos en el interior de la parábola.

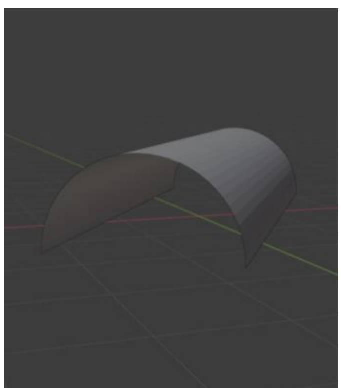
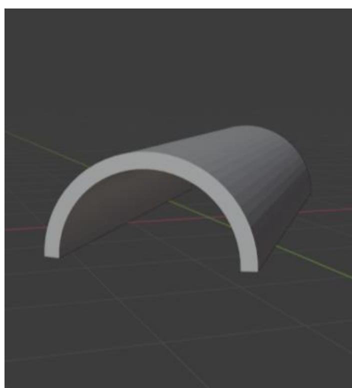



Figura 4-11. Aplicación de *solidify* en superficie cilíndrica.

- *Subdivision Surface*. En su modo simple, toma el mallado del objeto indicado y añade una subdivisión. Por ejemplo, ante un cuadrado, este modificador convertiría su malla en otros cuatro cuadrados conservando las dimensiones del objeto original. Existe otro modo que usa el algoritmo de Catmull-Clark, y que añade al simple un suavizado (Figura 4-12). El uso recursivo de este modificador se emplea para dotar al modelo de un mayor suavizado, a costa de un mayor número de elementos en el mallado y, por tanto, de un mayor impacto en el procesamiento para el ordenador. (Blender Reference Manual, 2021)

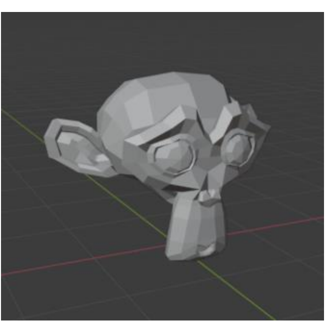


Figura 4-12. De izquierda a derecha: modelo original; *subdivision surface* aplicado una vez; *subdivision surface* aplicado tres veces.

4.2 Modelado de un módulo

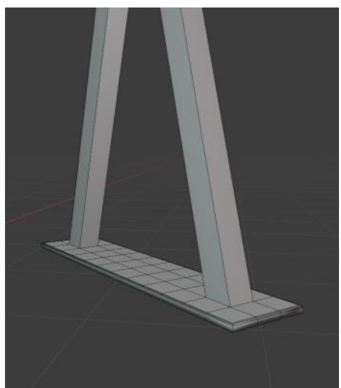
Debido a las numerosas herramientas que proporciona Blender, como las que se han comentado anteriormente, la mayoría del diseño manual del modelo se centra en un módulo, así como en las tuberías y soportes que los interconectan. El resto puede ser alcanzado con cierta facilidad si se usan de forma correcta los operadores y modificadores oportunos. Además, si se consigue realizar el modelo e implementar con éxito la simulación de

un solo módulo, el resto será cuestión de escalar lo programado a otros modelos superiores (colectores y lazos).

Dadas las características del movimiento del módulo en la simulación, se podría decir que el modelo contará con dos partes fundamentales: una estática, formada por las patas que hacen de soporte del módulo; y otra dinámica, comprendida por el eje de giro, la parábola y su estructura de soporte, y el tubo portador del líquido junto a su estructura. Por tanto, lo ideal es realizar el modelo en una posición de origen, y luego separar y exportar los mallados de cada una de las dos partes mencionadas.

4.2.1 Soporte

Para el soporte, de ahora en adelante las patas, se ha comenzado el modelo dando grosor a un rectángulo que hará de soporte para ambas. Tras realizar una subdivisión del mallado de la capa superior, se han extruido dos cuadrados en el plano vertical que forman las patas, con cierta inclinación. Posteriormente, estas se unen en otro poliedro rectangular similar al inferior trabajando los vértices y aristas. Por último, en la cara superior se inserta un cubo, el cual se agujereará para dar paso al eje de la parábola. Esto se realiza una vez que está diseñada, mediante un modificador booleano de diferencia.



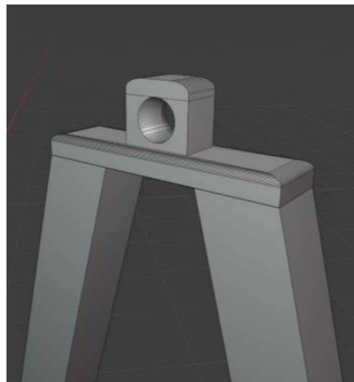


Figura 4-13. Base y eje de las patas del módulo.

Como se observa en la Figura 4-13, para finalizar el modelo se ha aplicado un suavizado selectivo en ciertas zonas, sobre todo en los poliedros rectangulares que unen ambas patas, y en el cubo que contiene el eje de giro. Además, basta con realizar el modelado manual de las patas de un lado del módulo. Las otras se incorporarán al modelo una vez se diseñe la parábola y su eje mediante un modificador *mirror*, y utilizando como espejo la propia parábola. De esta forma, las patas quedarán a la misma distancia, totalmente simétricas respecto al centro de gravedad de la parábola (Figura 4-14).

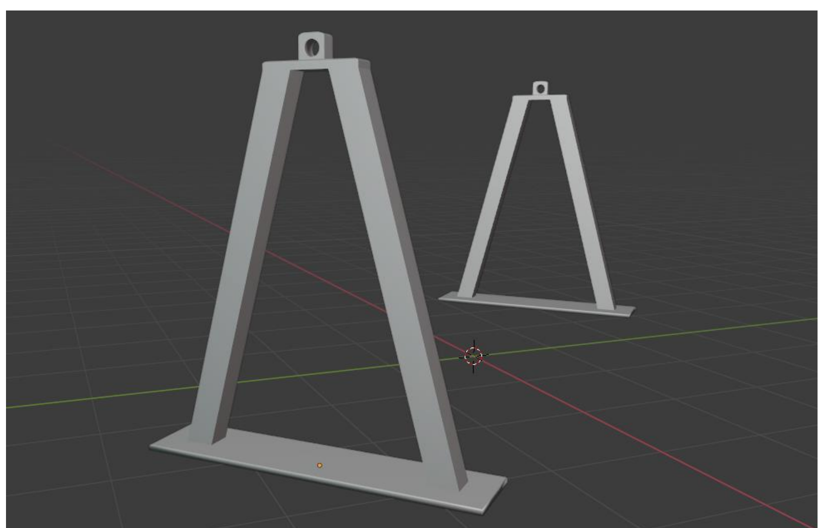


Figura 4-14. Modelo final del soporte del módulo.

4.2.2 Parábola

Esta parte del modelo es más compleja que la anterior, tanto por la cantidad de elementos a diseñar como por la forma de estos. El elemento principal en torno al que se disponen el resto de los elementos es la superficie parabólica. Para el diseño de esta, aunque se puede realizar manualmente con las herramientas que proporciona Blender y una buena referencia, se ha preferido optar por un complemento opcional que permite el trazado de superficies más complejas y su generación a partir de datos matemáticos.

El *AddOn* en concreto es "*Add Mesh: Extra Objects"*, y es posible instalarlo desde Blender con una simple búsqueda en el menú de complementos. Una vez hecho, a la hora de insertar una malla nueva se nos ofrecerán muchas más posibilidades, entre las que se presenta una para introducir una superficie tridimensional según una ecuación matemática para el eje Z.

Para la orientación requerida en la parábola, se trabajará en el plano XZ, por lo que la ecuación general de la parábola será la siguiente:

$$(x-h)^2 = 4p(z-k)\,, \tag{4–1}$$

Donde el vértice de la parábola estará en las coordenadas *(h, k)*, el foco estará en las coordenadas *(h, k+p)* y la directriz (el eje del tubo portador del líquido) será una recta de ecuación $z = k - p$. Sin embargo, Blender requiere especificar la superficie en el eje Z, por lo que, despejando, la ecuación a introducir cambiará a la siguiente forma:

$$z = \frac{(x-h)^2}{4p} + k \tag{4–2}$$

Por sencillez en la ecuación, se tomará como vértice de la parábola el origen de coordenadas de Blender. Esto no supone un problema, ya que luego se podrá desplazar la parábola cómodamente al lugar necesario. Por otro lado, establecerá una distancia focal *p* de 1.8 metros. Por tanto:

$$\begin{gathered} h, k = 0 \\ p = 1{,}8 \end{gathered} \tag{4–3}$$

Sustituyendo los valores anteriores en la ecuación (4-2), obtenemos la ecuación final a introducir en Blender:

$$z = \frac{x^2}{7,36} \tag{4–4}$$

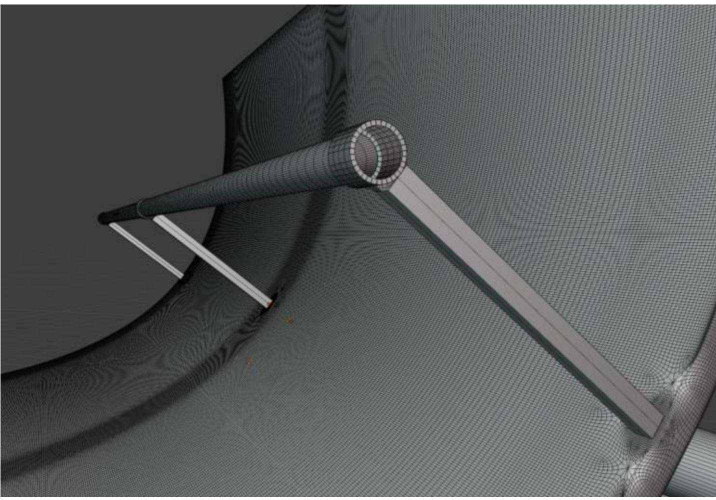


Figura 4-15. Mallado de la parábola creada y de los soportes del tubo portador del líquido.

Además de lo anterior, también se requieren ciertos valores complementarios para poder crear la parábola. Por un lado, las subdivisiones en los ejes X e Y, y, por otro lado, los límites de los ejes X e Y. Las subdivisiones serán un número entero en cada eje, que representan cada uno de los planos que formarán la parábola. A mayor subdivisión, la curva será más realista y menos plana. Sin embargo, esto conlleva un coste computacional para tener en cuenta. En el caso actual, se ha usado una subdivisión en cada eje de 32.

Por otro lado, es necesario definir los límites de la curva en los ejes. Estos irán determinados en función del tamaño final deseado para la parábola. Además, hay que tener en cuenta que estos límites se dan respecto al origen de coordenadas, por lo que habrá que indicar los límites como la mitad del valor deseado. Como se desean parábolas con un ancho de 5 metros y un largo de 7,8 metros, los valores en Blender serán, respectivamente, $x_{max} = 2,5$ e $y_{max} = 3,9$.

Con todo lo anterior, se dispone de una superficie parabólica con las características deseadas, como se aprecia en la Figura 4-15. Únicamente quedaría darle grosor, para lo que se utiliza el modificador para solidificar. Aunque se podría realizar de forma manual mediante extrusión, resulta más complejo realizarlo en una sola

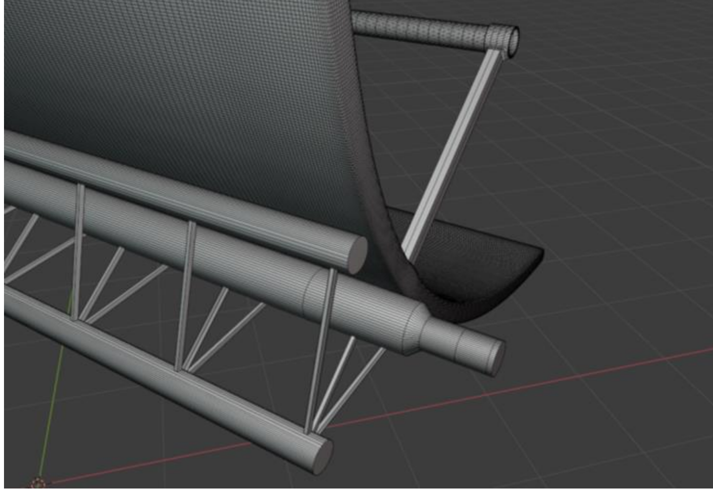


Figura 4-16. Mallado del eje y la estructura que soportan la parábola.

dirección. Si no se hiciera de dicho modo, las propiedades geométricas de la parábola se verían afectadas. Por ello, se opta por dar grosor con el modificador en la dirección exterior de la parábola.

A continuación, ya es posible modelar un cilindro que, unido a la cara exterior de la parábola a lo largo del eje Y, hará las funciones de eje de giro en el movimiento del módulo. También se crean otros tres cilindros en paralelo, de un radio menor, para dar forma a la estructura. Para crear las conexiones entre ellos, basta con realizarlo una vez en el extremo del módulo y usar el modificador *array* para que se distribuyan sus repeticiones a lo largo del eje. Con todo ello, queda creada la estructura y la parte exterior de la parábola (Figura 4-16).

Para la zona interior, es necesario crear tres poliedros rectangulares, que harán de soporte para el tubo, y unirlos a la cara interior. En el otro extremo, se acopla un pequeño cilindro de un diámetro ligeramente superior al del tubo portador de líquido. Finalmente, basta con crear el cilindro del tubo portador y usar un modificador booleano para que quede por dentro de los tres cilindros de los soportes.

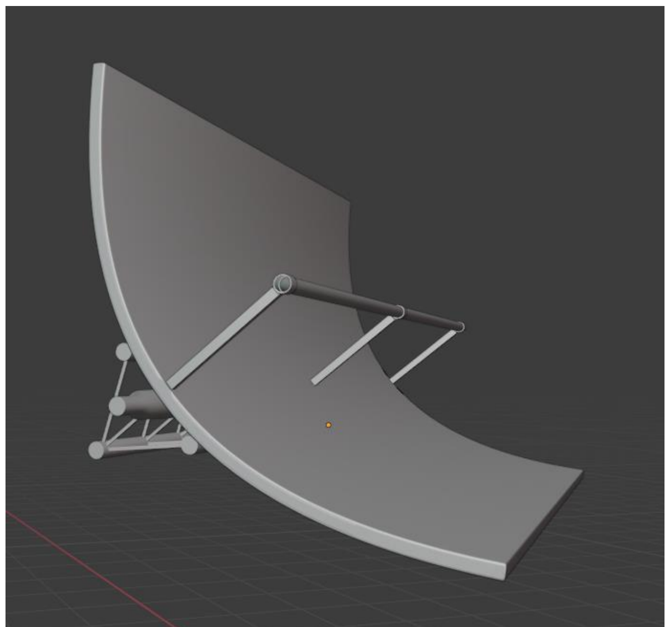


Figura 4-17. Modelo final de la parábola.

A lo largo de todo este proceso, es muy posible encontrar dificultades e incoherencias en el mallado, sobre todo a la hora de incorporar nuevos elementos y fusionarlos con los existentes mediante modificadores. Por ello, es necesario analizar el mallado, modificando vértice a vértice en las zonas afectadas para corregir la malla. También es conveniente usar el biselado para evitar bordes afilados y dar un aspecto más realista, así como suavizados en las zonas oportunas mediante la aplicación en determinados conjuntos de vértices. De esta forma, se obtiene un modelo final de parábola como el mostrado en la Figura 4-17.

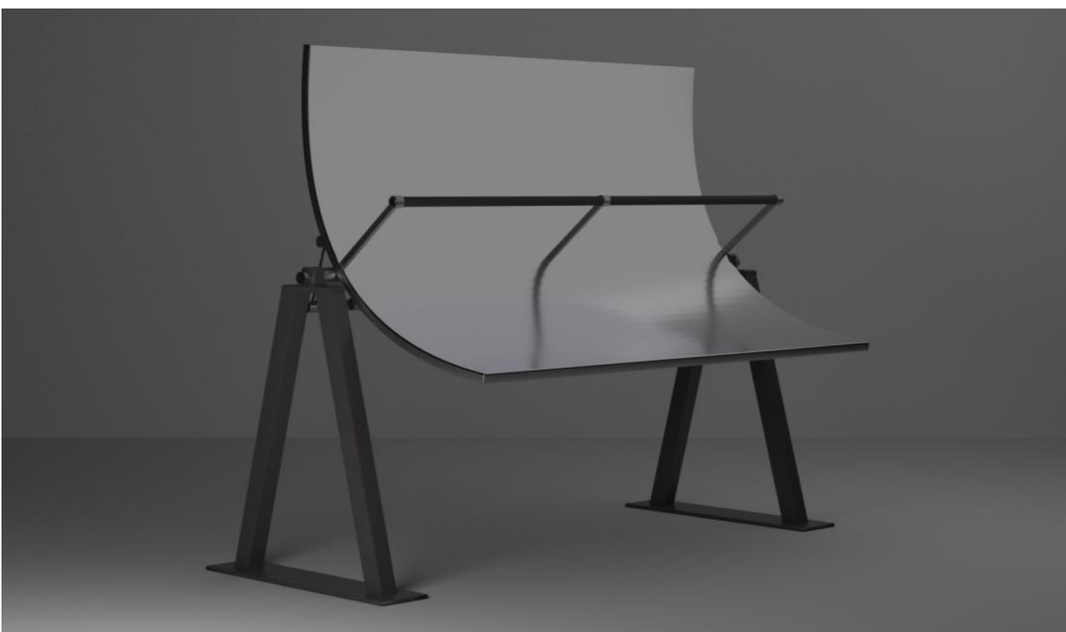


Figura 4-18. Renderizado frontal del módulo completo.

Finalmente, tras componer el módulo en su conjunto, se han realizado unos renderizados con el mayor detalle posible del modelo. Para ello, ha sido necesario incorporar en Blender varios puntos de luz, así como una cámara a partir de la cual obtener las imágenes. La posición y orientación de los elementos es clave a la hora de intentar obtener una imagen final que realce los perfiles del modelo, así como los parámetros de ajuste de la cámara.

Figura 4-19. Renderizado trasero del módulo completo.

En la Figura 4-18 y Figura 4-19 podemos apreciar un renderizado desde una vista frontal y trasera, respectivamente. La obtención de una imagen renderizada frente a una simple captura de pantalla del modelo permite observar con mayor lujo de detalle el comportamiento del modelo frente a la luz y las sombras generadas, así como un aspecto realista de los materiales aplicados en el desarrollo del modelo.

4.3 Modelado de un módulo simplificado

Tras realizar varias pruebas en la simulación con el modelo de módulo creado en el punto anterior, se encuentra que la ejecución de esta ofrece un rendimiento muy bajo. Si a esto se le suma que posteriormente habrá que multiplicar el número de modelos de forma considerable, se requiere de alguna solución para llevar a cabo de forma correcta la simulación.

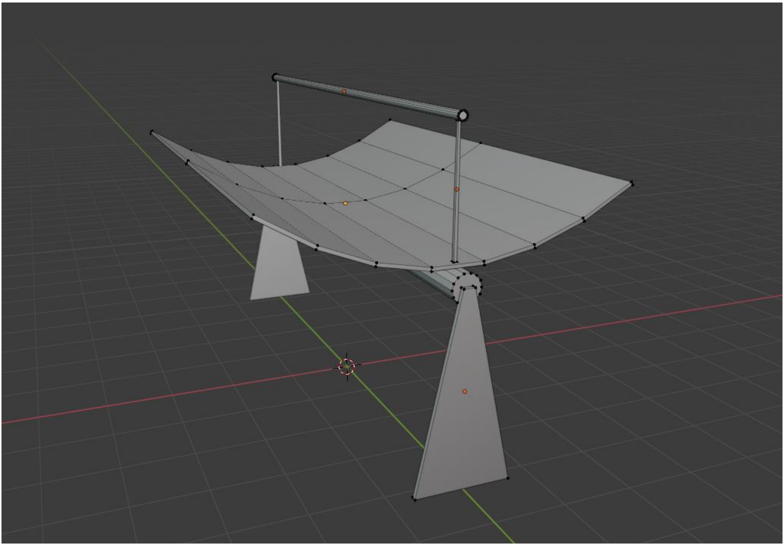


Figura 4-20. Modelo de un módulo simplificado.

Una vez revisados todos los puntos donde la simulación podía fallar, y al no encontrar ningún motivo claro para dar respuesta a tal lentitud, se prueba a realizar un modelo simplificado como el que se puede apreciar en la Figura 4-20. Tal y como se puede apreciar en la Figura 4-15 y en la Figura 4-16, la malla del modelo es realmente compleja, dado el nivel de detalle que se ha tenido en cuenta durante el modelado.

Salta a la vista cómo en el modelo simplificado el número de vértices disminuye drásticamente, lo cual parece repercutir en Gazebo de una forma muy positiva. Usando este modelo, las simulaciones recuperan la fluidez esperada. Es por ello por lo que a partir de este punto se decide proseguir con el modelo simplificado, a pesar de haber realizado ya el modelo complejo, con el fin de obtener mayor fluidez en la simulación.

4.4 Modelado de un colector

Para el desarrollo del modelo de un colector, se parte del módulo simplificado. Según las medidas usadas, se ha decidido implementar una longitud de colector de trece módulos. Estos se dispondrán en la misma dirección, unidos uno tras otro, y realizarán solidariamente el movimiento de giro según el eje.

La manera más sencilla de realizar esta tarea es usando el modificador *array*. Mediante este, podemos crear un número determinado de copias en una dirección concreta, además de poder establecer la separación entre ellas de distintos modos gracias a los *offsets* relativos y constantes. Ajustando los diferentes valores, se puede obtener el modelo final de colector simplificado (Figura 4-21).

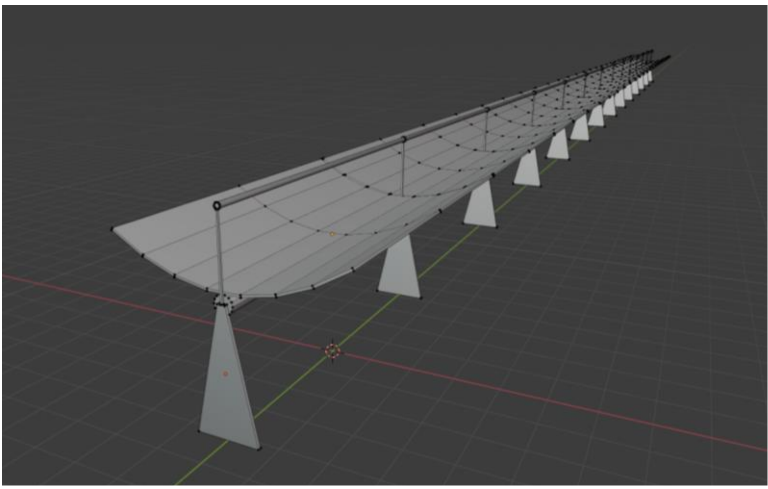


Figura 4-21. Modelo de colector a partir del modelo de módulo simple.

4.5 Modelado de un lazo

La elaboración del modelo de un lazo es más compleja que la del colector. Un lazo, formado por cuatro colectores encadenados uno tras otro, comprende el circuito total que recorre el fluido para calentarse. Por ello, como se observa en el modelo final en la Figura 4-22, se comunica con dos tuberías colectoras generales, una fría y otra caliente, para dar entrada y salida al fluido, respectivamente.

Dentro del lazo, los colectores se sitúan de forma que comparten el eje de giro dos a dos. En la realidad, cada colector tiene un movimiento independiente, pudiendo ser diferente el de dos que compartan un mismo eje. Sin embargo, de cara a la simulación, se ha establecido que realicen el mismo movimiento, por simplicidad en la ejecución y ahorro de medios de computación. Esto es más necesario aún tras ver la inestabilidad de la simulación cuando se satura de elementos.

Por otra parte, es necesario modelar individualmente las uniones entre los colectores, así como la de estos con las tuberías generales. En concreto, es necesario modelar en tres zonas diferentes. En primer lugar, la conexión

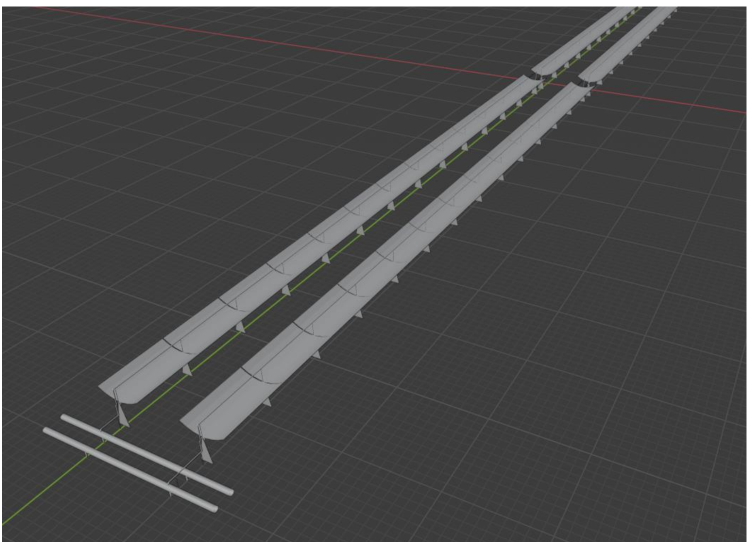


Figura 4-22. Modelo de lazo a partir del modelo de colector.

con las tuberías generales, así como el modelado de estas. Por otro lado, la unión de los colectores paralela al eje de giro, en concreto las uniones de los colectores uno con el dos y la del tres con el cuatro. Por último, será necesario modelar la unión entre los colectores dos y tres, la cual se da de forma perpendicular al eje de giro.

En primer lugar, se ha comenzado modelando las uniones intermedias (Figura 4-23). Al igual que para el resto de las zonas, y al estar trabajando el modelado de conductos, ha sido de mucha ayuda el operador *spin*, debido a que permite tomar la superficie final del tubo portador del líquido y darle continuidad según el giro dado.

Saliendo de los colectores, se han elaborado sendas tuberías con dos giros de 90º, las cuales giran solidarias a los colectores. Estas permiten reducir la altura del tubo horizontal que las conecta, ahorrando material y simplificando la construcción de los soportes de este. Así mismo, el diseño de ha realizado de forma que dicho tubo horizontal se sitúe exactamente en el eje de giro de los colectores, evitando el uso de elementos flexibles o de mecanismos más complejos.

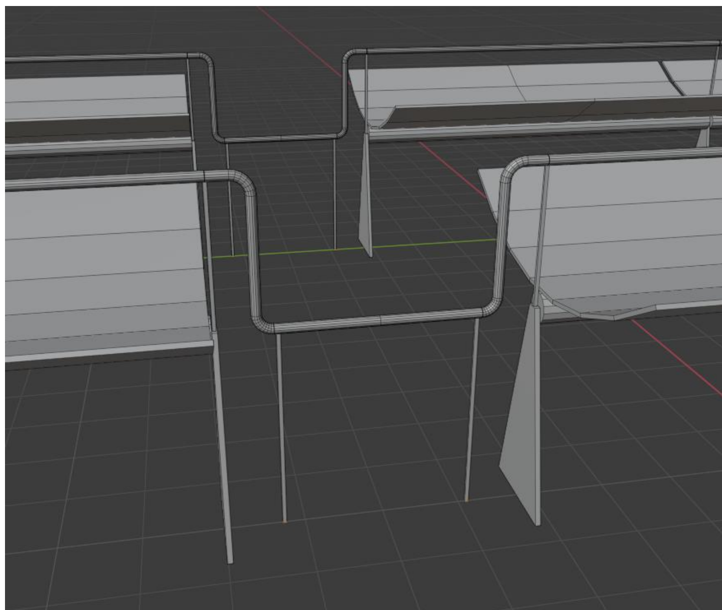


Figura 4-23. Uniones entre colectores que comparten eje de giro.

Posteriormente, se ha realizado el diseño de la unión perpendicular al eje entre los colectores segundo y tercero (Figura 4-24). Para ello, se ha partido de la idea de la unión explicada anteriormente, con la complejidad añadida de girar las tuberías unos 90º adicionales en el plano horizontal. Así, se hace posible elaborar una unión perpendicular al eje de giro.

Una particularidad de esta unión, al igual que sucedía en la anterior, es que es necesario que el tubo horizontal que conecta las tuberías giradas se sitúe contenido en el mismo plano del eje de giro, a la vez que en una dirección perpendicular a este. Para ello, el giro horizontal de 90º de las tuberías de conexión se debe dar exactamente a la altura del eje de giro.

Para el diseño de los soportes, como en la zona comentada previamente, se ha realizado una subdivisión del mallado del cilindro horizontal con el fin de extruir hacia abajo las superficies deseadas, y hacer que estas hagan de patas del soporte del tubo horizontal.

Así mismo, siempre que ha sido posible se ha utilizado el modificador *mirror*, no sólo de cara al ahorro de trabajo redundante, sino también con el objeto de elaborar un modelo simétrico sin perder detalle. Esto se ha realizado en las dos zonas comentadas hasta ahora, tanto esta como la anterior.

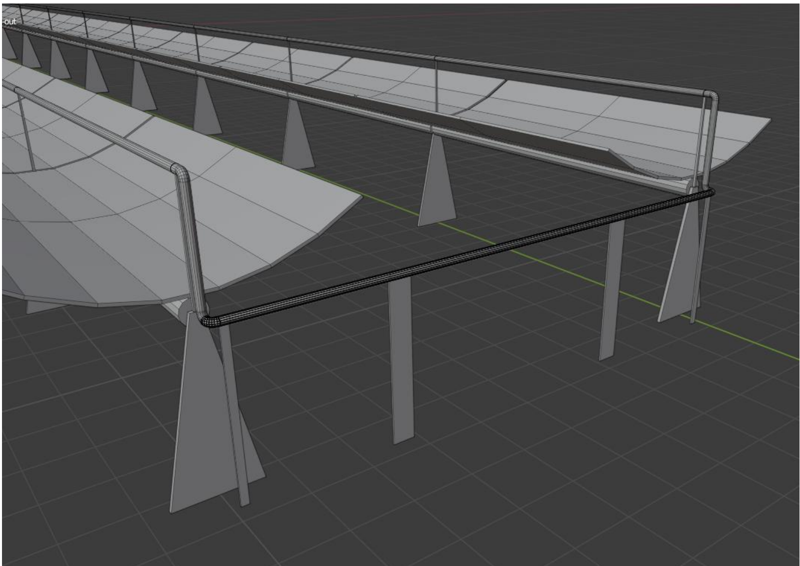


Figura 4-24. Unión entre colectores, perpendicular al eje de giro.

Por último, se ha procedido al diseño de las tuberías generales de distribución, así como la unión de estas con los módulos iniciales del colector (Figura 4-25). Para las tuberías se han usado cilindros, y para el soporte de estas se ha seguido la técnica de subdivisión del mallado y posterior extrusión.

Para la conexión de los colectores con las tuberías generales, se han prolongado las superficies del tubo portador del líquido realizando varios giros de 90º. Un detalle que comentar es que, al igual que pasa en los casos anteriores, es necesario realizar un codo a la altura del eje de giro de los colectores, para permitir que esta parte del tubo gire con el propio módulo.

Los apoyos de los tubos de conexión también parten de una extrusión de la superficie inferior del cilindro que forma el tubo. Finalmente, uno de los puntos más complejos del diseño en general ha sido la conexión de la tubería general con los tubos pequeños. Esto es debido a que es necesario fusionar dos superficies que a priori no encajan: por un lado, las caras rectangulares del cilindro mayor, y por otro, el polígono que forma el corte transversal del tubo pequeño.

Para solucionar dicho problema, se intentó generar una unión con el modificador *boolean*. Sin embargo, dada la forma tan distinta de ambos elementos, el algoritmo no funciona como debería y produce formas muy extrañas. Por ello, finalmente se decidió aproximar ambas mallas, para posteriormente crear vértices a mano que unieran las dos mallas. De esta forma también se evitaron incoherencias en la estructura del mallado a la hora de aplicar otros algoritmos como el suavizado automático.

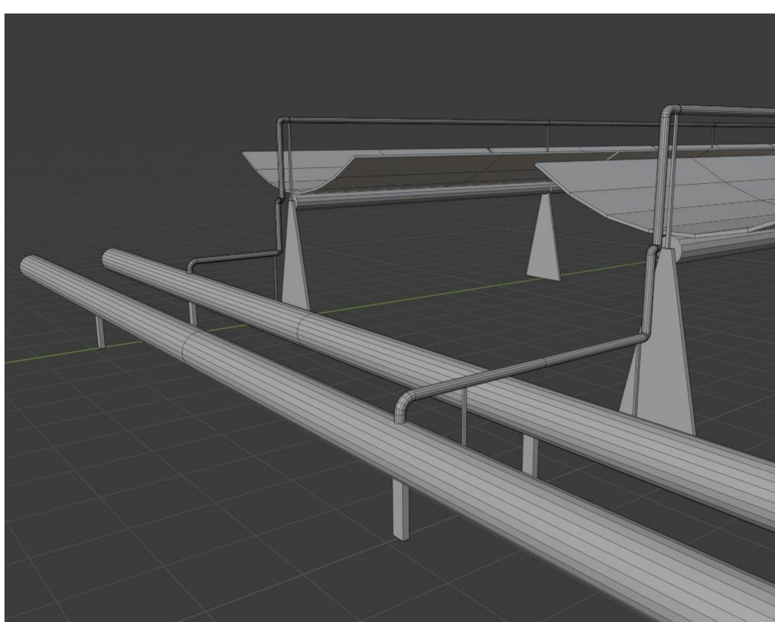


Figura 4-25. Unión entre colectores y distribución general.

Dada la estructura de la planta, se creará más de una fila de lazos. Además, la orientación de los lazos está dispuesta de forma que las tuberías generales quedan siempre lo más cercanas posibles dos a dos. Esto obliga a crear un modelo distinto de lazo, simétrico al ya creado respecto al plano vertical que contiene la dirección de las tuberías generales. Sin embargo, gracias al modificador *mirror*, esto puede realizarse de una forma muy sencilla.

5 DINÁMICA DE LA SIMULACIÓN

Una vez que se ha definido el entorno básico de la simulación, y se han modelado todos los elementos necesarios para dar forma a esta, es necesario adentrarse en la gestión de los movimientos de los modelos. Esto implica conocer a fondo el funcionamiento de Gazebo y ROS, así como la forma de comunicación entre los distintos procesos.

Es muy probable que se puedan alcanzar los objetivos siguiendo diferentes métodos. De hecho, debido a que en la simulación se ven involucrados diferentes softwares, es necesario gestionar correctamente las órdenes que se den. Sin embargo, lo ideal sería hacerlo de la forma en la que se obtenga el mayor desempeño posible en la simulación, ya que esta es susceptible de sufrir bajones de rendimiento debido a la alta carga computacional que conlleva el diseño de toda una planta termosolar.

En general, la idea que se persigue es dotar a los colectores de un movimiento de rotación individualizado, en torno a su eje de giro, en el que se trata de maximizar en todo instante la exposición de estos a los rayos de luz solar. Este movimiento podrá ser ordenado manual o automáticamente. Además, también se pretenderá desarrollar la capacidad de parar colectores de forma individual para proceder con tareas como el mantenimiento programado o la reparación puntual de estos.

En el presente capítulo se irán exponiendo las variantes adoptadas para conseguir el movimiento deseado, así como algunas de las alternativas que también se pensaron en un inicio, y las soluciones planteadas ante los diferentes problemas que han ido surgiendo a lo largo del desarrollo de la simulación.

5.1 Movimiento de un colector

Hasta este punto, se han conseguido generar en la simulación cualquiera de los modelos diseñados, aunque únicamente de forma estática. Desde un comienzo, el principal reto es hacer que el colector se mueva a una posición dada de forma manual, a partir del único grado de libertad disponible para hacerlo.

Para incorporar a la simulación el modelo deseado y poder dar comienzo a las pruebas, se crea un archivo SDF básico en el que coexistirán un suelo, el Sol y el colector. El colector es un solo modelo, aunque compuesto por dos *links*, es decir, dos partes rígidas y diferenciadas con movimientos independientes: el soporte, el cual es estático, y la parábola, que realizará los movimientos. También se especifica la articulación (*joint*) que une ambas partes, la cual es de tipo *revolute*. Esta permite que se realicen giros en torno a un eje determinado.

Dadas las características del modelo mencionadas anteriormente, la diferencia entre el modelo del colector completo y un módulo es prácticamente nula. Únicamente variarían las propiedades físicas, como masa e inercias, y el tipo de malla cargada para dar forma al modelo. El resto de los elementos en el archivo SDF son prácticamente iguales, por lo que es posible realizar las pruebas con un colector completo en lugar de empezar por el módulo.

Respecto al movimiento en general, se pretende desarrollar u obtener una función que mueva el colector a una posición angular especificada, gracias a la introducción de comandos por terminal. Idealmente, tras conseguir esto, se podría elaborar un script que automatizara los movimientos del colector realizando un giro determinado cada cierto tiempo, así como interrumpir el movimiento para volver a la posición de reposo cuando se desee. Si

se englobara todo esto a su vez en otro archivo que gestione todos los elementos de la simulación, se habrá conseguido automatizar el movimiento de toda la planta, incluidos los paros y puestas en marcha individualizados de los elementos que se desee.

5.1.1 Idea inicial: uso de *topics* de Gazebo

En un comienzo, se intenta dotar al modelo de cierto movimiento de la manera más sencilla posible. Para ello, la primera idea es publicar un mensaje por terminal en un *topic* de Gazebo, de forma que podamos indicar una posición concreta a cualquier elemento introducido en la simulación. Tras realizar una búsqueda de los *topics* disponibles cuando se inicia la simulación, parece apropiado publicar mensajes en el *topic* `/gazebo/set_link_state`. Al igual que este, también existe otro para modelos, aunque el objetivo en este caso es mover solo el *link* superior, la parábola.

Para realizar una publicación se usa el comando `rostopic pub`. Tras escribir dicho comando, y usando la tecla de tabulación en el terminal, se pueden visualizar todos los *topics* disponibles donde será posible publicar mensajes. Así mismo, una vez escrito el *topic* en cuestión, si se pulsa tabulador se autocompletará el comando siempre que sea necesario añadir más información, como por ejemplo el tipo de mensaje o algún argumento requerido para el funcionamiento.

Teniendo en cuenta todo lo anterior, y tras ejecutar la simulación, se ejecuta el siguiente comando en una nueva ventana del terminal:

```
>> rostopic pub /gazebo/set_link_state gazebo_msgs/LinkState '{link_name: link_parabola, pose: { position: { x: 0, y: 0, z: 0 }, orientation: {x: 0, y: 0, z: 0, w: 0 } }, twist: { linear: { x: 0, y: 0, z: 0 }, angular: { x: 0, y: 1, z: 0}  }, reference_frame: link_patas }'
```

En dicho comando se especifica un giro de la parábola en radianes. En el instante de la ejecución, la parábola alcanza la posición deseada. Sin embargo, tras ello, lejos de mantener la posición especificada, esta comienza a realizar movimientos libremente hasta retornar a la posición de equilibrio. Es decir, con este comando se consigue situar al modelo en una posición deseada, aunque no mantenerlo en ella.

Para intentar dar solución a esto se intenta complementar el comando `rostopic` con el flag `-r`, que permite hacer que la publicación se ejecuta de forma recursiva con una frecuencia concreta especificada por el usuario. Por ejemplo, ejecutando `rostopic pub -r 10`, conseguiremos que el mensaje deseado se publique con una frecuencia de 10 Hz.

A priori, se podría pensar que con una frecuencia determinada se podría evitar que el modelo comience a oscilar, y sería una forma de mantenerlo estático en la posición especificada. Sin embargo, esto no sucede como se piensa, debido a que el comando mueve el modelo desde la posición de reposo hasta la especificada. Si se establece la recursividad en la ejecución, lo que se consigue es realizar este movimiento de una forma muy rápida, no fijarlo en la posición de destino. Se provoca una rotación caótica, la cual no supone ningún tipo de solución al problema existente.

Tras buscar otros *topics* donde practicar un método similar, no se encuentran alternativas para conseguir lo deseado. Sin embargo, al menos ha sido posible verificar que el movimiento del modelo es el correcto, ya que la parábola se mantiene fijada al eje durante el giro en todo momento, así como las patas permanecen estáticas y unidas al suelo.

Por tanto, será necesario buscar alternativas que vayan más allá de trabajar con simples publicaciones en *topics* existentes. Probablemente, el principal fallo haya sido intentar realizar la comunicación de forma directa con Gazebo. Lo ideal es realizar toda la computación y el intercambio de mensajes en ROS, y dejar a Gazebo únicamente como visualizador para la simulación y como fuente de datos para conocer los parámetros en tiempo real de los objetos de la simulación.

5.1.2 Creación de *plugins* propios como primera alternativa

Una forma distinta de proceder para buscar solución al problema existente es la creación de *plugins* propios. Estos son fragmentos de código que pueden estar escritos en Python o en C++, y que se componen de órdenes propias de librerías de ROS para dotar de cierta funcionalidad a los modelos o el mundo de la simulación. Estos plugins pueden también crearse para modificar elementos directamente en Gazebo, hecho que, como sucedía en la alternativa anterior, no sirve de mucho dados los objetivos de este trabajo.

Para desarrollar un plugin es necesario tener bastante conocimiento de las librerías y de las funciones que otorgan estas. Esto se acentúa más aún cuando entran softwares adicionales, como en este caso Gazebo, ya que es necesario conocer bien cómo se integran con ROS. Más allá de esto, como es de esperar, es necesario un nivel medio en programación de dichos lenguajes si se quiere realizar con cierta fluidez.

Cuando se creen plugins, es necesario referenciarlos en la variable `$PLUGIN_PATH`. De lo contrario, ROS será incapaz de encontrarlos. Además, es necesario configurar de forma correcta el archivo *CMakeLists*, para especificar las dependencias y componentes necesarios. Así, no habrá problemas a la hora de compilar los plugins con la orden `catkin_make` tras realizar alguna modificación en ellos.

En primer lugar, se realizó un plugin para mover un modelo, el cual funciona correctamente, así como también otro *world plugin* capaz de modificar parámetros del entorno de la simulación e imprimir mensajes por pantalla. Sin embargo, empiezan a surgir problemas cuando se quieren modificar elementos dentro del modelo, como los *links* y las articulaciones.

La complejidad realizando el plugin deseado aumenta bastante. Empieza a ser necesario el uso de órdenes a más bajo nivel, para realizar modificaciones que, en la mayoría de las ocasiones, suelen resolverse mediante librerías. Por todo ello, antes de proseguir con este método se intentan buscar otras alternativas que ofrezcan mayor versatilidad con menor desempeño.

5.1.3 Idea final: control en bucle cerrado

5.1.3.1 La librería ROS Control

Todo lo comentado en los dos puntos anteriores, más allá de la complejidad que implica, no dejan de ser más que soluciones de control en bucle abierto. Sin embargo, aunque su implementación tampoco resulta sencilla, existen alternativas que permiten implementar un control en bucle cerrado.

La opción tomada en este caso, que es la que se ha implementado con éxito desde este punto hasta el final del proyecto, es el control en bucle cerrado gracias al uso de *ROS Control*. Se trata de una serie de paquetes que permiten implementar controladores y hacerlos genéricos para todos los robots. Para ello, se toma como entrada el estado de la articulación del actuador, con el fin de controlar la salida mediante un mecanismo en bucle retroalimentado, normalmente un PID.

Aunque es posible elaborar desde cero cualquier tipo de controlador, lo normal es usar alguno de los ya existentes en la librería *ros_controllers*, o al menos partir desde ellos para modificarlos. Dentro de dicha librería, se encuentran diversos controladores de posición, velocidad y esfuerzo, entre otros. En el presente trabajo se ha usado un controlador de esfuerzo basado en posición, es decir, se corrige la posición angular de la articulación aplicando cierta fuerza.

Sin embargo, uno de los puntos por los que no se ha tomado esta alternativa hasta el final es por la cantidad de modificaciones requeridas respecto a lo trabajado anteriormente. Entre los principales inconvenientes para el uso de *ROS Control* frente a lo que se ha desarrollado hasta el presente capítulo, así como algunas nuevas cuestiones para tener en cuenta, destaca lo siguiente:

- Modelado en URDF. El tipo de archivo para la descripción de modelos es URDF, siendo imposible el uso de SDF como hasta ahora. Esto implica realizar una traducción de los archivos desarrollados, así como de los modelos creados hasta ahora.
- Error en texturas importadas. Los archivos *collada* (formato .dae) importados en los archivos SDF permitían mantener las texturas y materiales diseñados en el software de modelado en 3D. Sin embargo,

al importarlos en archivos URDF, estas características no son representadas en Gazebo, apareciendo los modelos en un color blanco totalmente plano. La única forma de dar color es usando colores y materiales como se explica en el capítulo 3.3.2.1, aunque es muy complejo alcanzar la calidad que aporta el uso de los que contiene la malla original. Por todo ello, para la elaboración de modelos en URDF basta con importar la malla de colisiones (formato .stl), tanto para el apartado respectivo como para el visual.

- Bloque *transmission*. En el archivo URDF, es necesario crear una nueva etiqueta llamada *transmission*. Esto representa una interfaz en la que se crea un actuador para cada articulación que se desee controlar (Figura 5-1). Su función principal es mantener constante la potencia del eje, es decir, el producto de la fuerza aplicada por su velocidad (ROS Wiki, 2018). Es necesario indicar el tipo de transmisión usada, en este caso una simple, así como la reducción mecánica.

```
<transmission name="trans_patas_a_parabola">
  <type>transmission_interface/SimpleTransmission</type>
  <actuator name="motor_rotacion">
    <mechanicalReduction>1</mechanicalReduction>
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </actuator>
  <joint name="joint_patas_a_parabola">
    <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
  </joint>
</transmission>
```

Figura 5-1. Bloque de transmisión usado para el control de un colector.

- Archivos de configuración YAML. En ROS, el uso de estos archivos es necesario para establecer la configuración de los parámetros del *parameter server* al ejecutar cualquier nodo. En concreto, al usar *ROS Control*, en este archivo establecemos los parámetros proporcional, integral y derivativo del controlador PID, así como también la tasa de refresco de publicación del valor de posición del eje (Figura 5-2). Estos valores son obligatorios para el funcionamiento de los controladores. Sin embargo, estos pueden ser modificados incluso durante el tiempo de ejecución, mediante la ejecución de comandos desde el terminal.

```
# Publish joint state
joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

# Position controllers
patas_parabola_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint_patas_a_parabola
    pid: {p: 160.0, i: 0.0, d: 250.0}
```

Figura 5-2. Archivo de configuración YAML para controlar un colector.

Tras solventar todos los inconvenientes mencionados en los puntos anteriores, se encuentra un error al intentar ejecutar la simulación. Dicho error surge cuando ROS no es capaz de cargar el tipo de controlador requerido, debido que no existe. En concreto, se puede ver el mensaje obtenido en el terminal tras intentar iniciar la simulación en la Figura 5-3.

Al comprobar los paquetes instalados cuando se configuró inicialmente ROS, se comprueba que los controladores no existen en ningún directorio, a diferencia de lo que se indica en la instalación general de ROS. Esto es debido a que, debido a problemas de seguridad en los repositorios de instalación de ROS, cambiaron las claves de seguridad de estos, y por consecuencia la ubicación de los paquetes para la instalación de los controladores.

Para solventar dicho error, es necesario actualizar las claves de los repositorios. En primer lugar, mediante el

siguiente comando en un nuevo terminal se borran las claves existentes:

```
>> sudo apt-key del 421C365BD9FF1F717815A3895523BAEEB01FA116
```

```
edumbriz@edumbriz-GT70-2PC: ~/TFM
Archivo Editar Ver Buscar Terminal Ayuda
[colector/urdf_spawner_colector-4] process has finished cleanly
log file: /home/edumbriz/.ros/log/735aa7b2-eeba-11eb-b50e-f816543b4b92/colector-urdf_spawner_colector-4*.lo
g
[INFO] [1627377015.124711, 0.400000]: Controller Spawner: Waiting for service /colector/controller_manager/
switch_controller
[INFO] [1627377015.126800, 0.402000]: Controller Spawner: Waiting for service /colector/controller_manager/
unload_controller
[INFO] [1627377015.128571, 0.404000]: Loading controller: /colector/joint_state_controller
[INFO] [1627377015.136526, 0.412000]: Loading controller: /colector/patas_parabola_position_controller
[ERROR] [1627377015.140040941, 0.410000000]: Could not load controller '/colector/patas_parabola_position_c
ontroller' because controller type 'effort_controllers/JointPositionController' does not exist.
[ERROR] [1627377015.140085217, 0.410000000]: Use 'rosservice call controller_manager/list_controller_types'
 to get the available types
[ERROR] [1627377016.141418, 1.415000]: Failed to load /colector/patas_parabola_position_controller
[INFO] [1627377016.143493, 1.415000]: Controller Spawner: Loaded controllers: /colector/joint_state_control
ler
[INFO] [1627377016.155344, 1.426000]: Started controllers: /colector/joint_state_controller
^C[colector/colector_controller_spawner-5] killing on exit
[gazebo_gui-3] killing on exit
```

Figura 5-3. En rojo, mensaje de error obtenido al iniciar la primera simulación con *ROS Control*.

A continuación, para introducir la nueva clave, se usa el comando:

```
>> sudo -E apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --
recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Por último, es necesario ejecutar un último comando, tras el cual no se recibirá ningún tipo de advertencia tras intentar actualizar los repositorios:

```
>> sudo apt clean && sudo apt update
```

Con todo ello, quedan configurados correctamente los repositorios, tras lo cual se pueden instalar las librerías con todos los controladores con éxito. Después de ello, no aparecen mensajes de error al iniciar la simulación y esta se ejecuta sin ningún problema.

Sin embargo, para llegar a este punto ha sido necesario modificar los archivos de lanzamiento en ROS, dado que ahora intervienen en la simulación nuevos nodos, generados por las nuevas librerías añadidas. En concreto, los nuevos nodos son los siguientes:

- Nodo generador de modelos *(urdf_spawner)*. Se realiza una carga de la configuración de los parámetros del PID, necesarios para ejecutar el nodo de generación del modelo (Figura 5-4). Además, es necesario especificar las coordenadas tridimensionales de posición donde se generará el modelo, lo cual se realiza en las líneas previas a la definición del nodo.

```
<!-- carga de config. de controladores -->
<rosparam file="$(find ptcplant)/config/colector_control.yaml" command="load" ns="/colector" />

<param name="/colector/robot_description" command="cat $(arg urdf_robot_file)" />

<node name="urdf_spawner_colector" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
      args="-urdf -x $(arg x) -y $(arg y) -z $(arg z) -model $(arg robot_name) -param /colector/robot_description"/>
```

Figura 5-4. Nodo generador de modelos.

- Carga y ejecución de controladores. Este nodo se encarga de generar un controlador asociado a un modelo. Es necesario especificar en los argumentos los controladores que se desean cargar de entre los disponibles en el archivo de configuración YAML. No es necesario cargarlos todos, sin embargo, si es necesario que se definan en dicho archivo si se quieren cargar en la simulación, de ahí la importancia del archivo de configuración YAML (Figura 5-5).

```
<!-- carga (y ejecuta) los controladores. Elegimos los que queramos de los especificados en el config -->
<node name="colector_controller_spawner" pkg="controller_manager" type="spawner" respawn="false" output="screen"
    args="--namespace=/colector /colector/joint_state_controller
                                /colector/patas_parabola_position_controller">
</node>
```

Figura 5-5. Nodo de carga de controladores.

- Nodo publicador del estado del robot. Dicho nodo se encarga de publicar la información de estado de las diferentes partes del robot para poder analizarla en RViz, software diseñado para la visualización y verificación del funcionamiento del modelo creado. Además, se especifica que se publique dicha información con una frecuencia exacta de 5 Hz (Figura 5-6).

```
<!-- robot state publisher. nos servira para visualizar el robot en rViz -->

<node name="robot_state_publisher_colector" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false" output="screen">
    <param name="publish_frequency" type="double" value="5.0" />
</node>
```

Figura 5-6. Nodo publicador del estado del robot.

Tras incorporar las líneas anteriores al archivo de lanzamiento e iniciar la simulación, ya es posible comenzar a lanzar comandos para cambiar la posición del modelo generado. El sistema se ha configurado de forma que, desde un nuevo terminal, se publique en un determinado *topic* la posición angular a la que se desea mandar el colector, en radianes. Para mayor sencillez, la posición cero es la que toma el colector cuando está totalmente vertical.

En concreto, esta información debe ser publicada en un *topic* con el nombre del modelo en cuestión, seguido del nombre del controlador. Dentro de este, existen varios *topics* donde publicar información, entre ellos los que permiten cambiar los parámetros del PID. Sin embargo, es necesario acudir al *topic* .../`command` mediante la orden `rostopic` para publicar la información de la posición. Para más ayuda, lo ideal es usar el autocompletado en el terminal, mediante el tabulador, ya que se indica automáticamente el tipo de mensaje y la forma de introducir el mismo, como se puede apreciar en la Figura 5-7.

```
edumbriz@edumbriz-GT70-2PC:~$ rostopic pub -1 /lazoA1/patas_parabola_in_position_controller/
/lazoA1/patas_parabola_in_position_controller/command
/lazoA1/patas_parabola_in_position_controller/pid/parameter_descriptions
/lazoA1/patas_parabola_in_position_controller/pid/parameter_updates
/lazoA1/patas_parabola_in_position_controller/state
edumbriz@edumbriz-GT70-2PC:~$ rostopic pub -1 /lazoA1/patas_parabola_in_position_controller/command
std_msgs/Float64 "data: 1.2"
publishing and latching message for 3.0 seconds
edumbriz@edumbriz-GT70-2PC:~$ 
```

Figura 5-7. Uso de *rostopic* para mover un modelo a la posición angular 1,2 radianes.

Una vez ejecutada la orden, se observa cómo el modelo en cuestión comienza a moverse con destino a dicha posición. Sin embargo, se observa que sobrepasa dicha posición por su propia inercia, y comienza a oscilar en torno a la posición objetivo. Esto es debido a que los valores del PID configurado no son los adecuados, y es necesario recalcularlos de cara a un funcionamiento óptimo según las características físicas del modelo.

5.1.3.2 Sintonización manual del PID

Para ello, se realizará una sintonización manual del controlador gracias a la interfaz *rqt_gui*. Esta permite visualizar e interactuar con ROS de una forma más automatizada y visual: desde publicar mensajes programados en determinados *topics*, incluso mediante funciones, hasta la visualización gráfica del contenido de los *topics*, pasando por la capacidad de configurar de forma dinámica ciertos parámetros, en el tiempo de la ejecución.

Se aprovecharán las funciones mencionadas anteriormente para la sintonización. En primer lugar, con la simulación inicializada y en un nuevo terminal, se ejecutará la interfaz gráfica con el siguiente comando:

```
>> rosrun rqt_gui rqt_gui
```

Una vez dentro de ella, en la pestaña *Plugins*, y dentro de *Topics*, se usa el *Message Publisher* (Figura 5-8). En este apartado, se dispone de un publicador de mensajes, que escribirá en el *topic* elegido y a una frecuencia determinada. Para ello, se elige el *topic* …`/command` de cualquier modelo generado, con una frecuencia de 50 Hz, y se introduce en el panel con la cruz verde.

Se desea provocar un movimiento sinusoidal en un colector. Para ello, en el campo *data*, introducimos una función seno de amplitud 2, en la cual el valor *i* representa el tiempo en la simulación. Una vez introducida la expresión, se comenzarán a publicar mensajes una vez quede marcado el tic negro a la izquierda del nombre del *topic*.

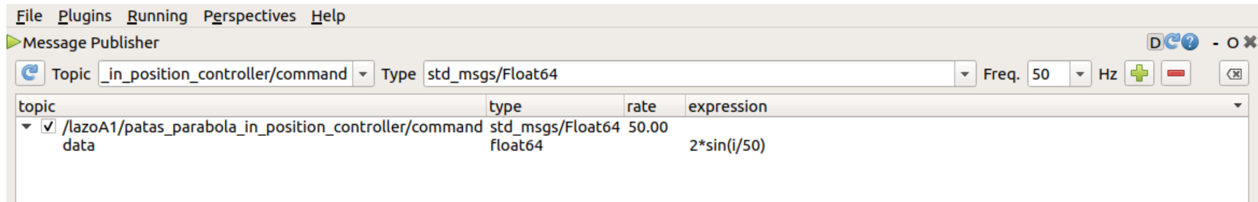


Figura 5-8. *Message Publisher* de *rqt*.

Tras ello, es posible visualizar en Gazebo como el colector realiza un movimiento oscilatorio constante, aunque no como se ha establecido. El movimiento configurado anteriormente, en el *topic* …`/command`, es la señal de entrada, y el que se visualiza en Gazebo es la salida. A continuación, la idea es modificar los parámetros del PID en tiempo real, de forma que la salida sea lo más parecida posible a la entrada, con una reacción rápida.

Para poder comparar entrada y salida, se usará el visualizador de datos gráficos. Para ello, en la pestaña *Plugins* y posteriormente *Visualization*, se selecciona la orden *Plot*. Para añadir la información a mostrar en la gráfica, en primer lugar, se elige el *topic* de entrada, donde se publica la información (…`/command/data`). Por otro lado, se introduce la posición real del eje, la salida, introduciendo el *topic* (…`/state/process_value`). De este modo, podemos comparar visualmente ambas curvas, con el fin de aproximarlas lo máximo posible (Figura 5-9).

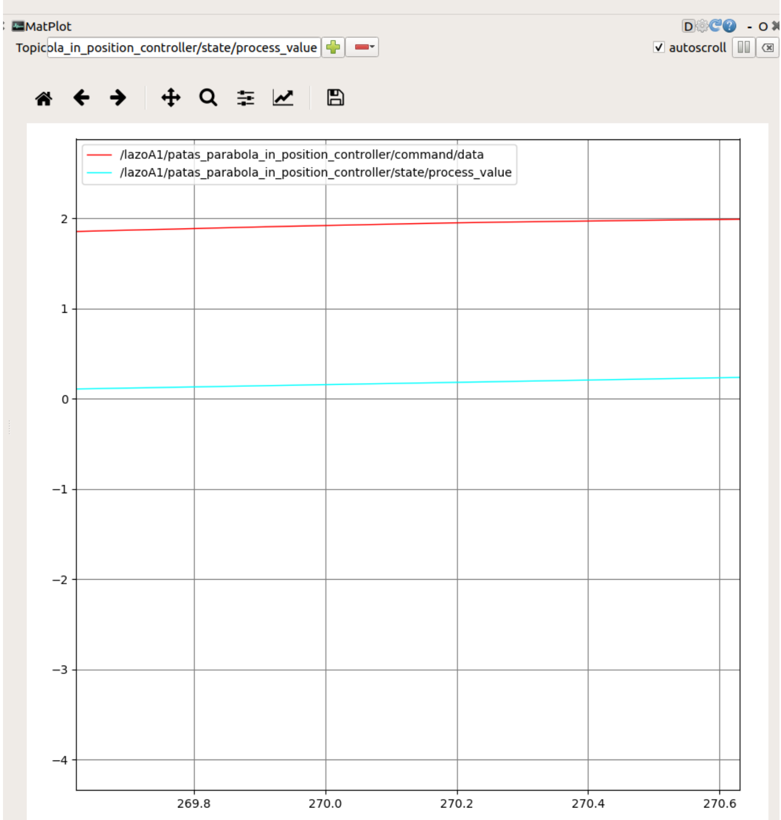


Figura 5-9. Visualizador gráfico de *rqt*.

Por último, sólo queda tener la capacidad de modificar los valores del PID en tiempo real para mejorar la respuesta del controlador. Para ello se abre la pestaña de *Plugins*, *Configuration* y posteriormente *Dynamic Reconfigure*. Como representa la Figura 5-10, aparecen unos controladores en forma de barra para cada uno de los parámetros, y es posible viendo si la respuesta mejora o empeora en función de cómo se modifiquen. Aunque este proceso se hace a ojo, lo ideal es comenzar modificando la constante integral para mejorar la respuesta en tiempo permanente, seguido de una modificación de la proporcional para agilizar la respuesta.

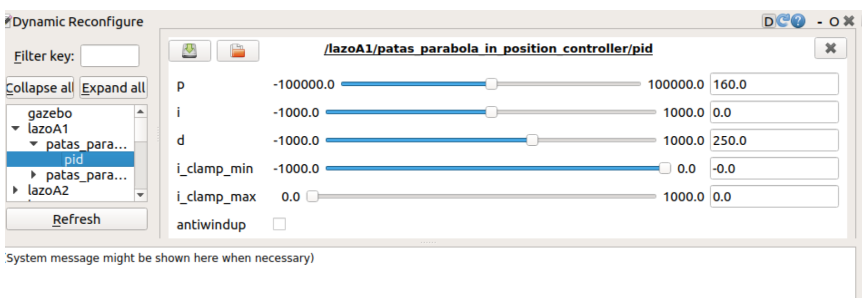


Figura 5-10. *Dynamic Reconfigure* de *rqt*.

Tras obtener unos valores acertados para el controlador, se observa como la respuesta mejora notablemente. La oscilación al alcanzar la posición objetivo es nula, también gracias a que la velocidad angular del motor de la articulación se ha limitado numéricamente en el modelo URDF del colector. Por ello, se puede dar por finalizada la sintonización del PID, siempre que no se realice ninguna modificación de las propiedades físicas y geométricas del colector.

5.2 Movimiento de un lazo

Una vez configurado correctamente el movimiento para un colector, el siguiente paso es importar el modelo de lazo para llegar al mismo fin. En este caso, a diferencia del anterior, el modelo de lazo cuenta con dos ejes de giro distintos que actúan independientemente, aunque permanecen al mismo modelo.

En el modelo de colector, el modelo constaba de dos partes básicas: una estática (patas) y otra dinámica (parábola). Sin embargo, en el modelo de lazo existen dos partes dinámicas, las cuales se identificarán como parábola de entrada y parábola de salida. No es necesario distinguir las partes estáticas, y parece mejor idea aunarlas todas en una sola.

```yaml
# Publish joint state
joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

# Position controllers
patas_parabola_in_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint_patas_a_parabola_in
    pid: {p: 160.0, i: 0.0, d: 250.0}

patas_parabola_out_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint_patas_a_parabola_out
    pid: {p: 160.0, i: 0.0, d: 250.0}
```

Figura 5-11. Archivo *lazo_control.yaml*.

Así mismo, en el modelo URDF del lazo ha sido necesario introducir una nueva articulación, habiendo ahora dos distintas, así como dos bloques distintos de tipo *transmission*. Debido a ello, lo ideal es definir un nuevo controlador, pasando a haber dos distintos, uno para la parábola de entrada y otro para la de salida. Por tanto, es necesario crear un archivo de configuración YAML nuevo (Figura 5-11), en el cual se especifiquen por separado los parámetros de cada controlador.

La creación de un nuevo archivo de configuración de parámetros es algo obligatorio, ya que al introducir un nuevo controlador este tiene que saber dónde encontrar los parámetros con los cuales inicializarse. Sin embargo, en el caso que nos atañe, y al ser idéntico el control de dos colectores distintos dentro de un mismo lazo, basta con replicar las características del controlador ya diseñado, cambiando únicamente el nombre para poder distinguirlos.

Por otro lado, también es necesario modificar el archivo de lanzamiento de la simulación, en los puntos mencionados en el capítulo 5.1.3.1. En concreto, se creará un nuevo archivo para el lanzamiento y control del lazo para no perder la posibilidad de lanzar un colector individualmente si fuera necesario.

De cara al nuevo archivo, el robot cambiará de nombre y pasará de colector a lazo. Por ello, es necesario cambiar el nombre de este allá donde se mencione. Además, hay que hacer alusión al nuevo controlador incorporado, para lo que es necesario añadirlo como un argumento más en el nodo encargado de generar y ejecutar los controladores.

```
<!-- carga (y ejecuta) los controladores. Elegimos los que queramos de los especificados en el config -->
<node name="lazo_controller_spawner" pkg="controller_manager" type="spawner" respawn="false" output="screen"
    args="--namespace=/lazo /lazo/joint_state_controller
                            /lazo/patas_parabola_in_position_controller
                            /lazo/patas_parabola_out_position_controller">
</node>
```

Figura 5-12. Nodo de carga de controladores para un lazo.

Como se puede apreciar en la Figura 5-12 si se observan los argumentos especificados al nodo, se puede ver que, aparte del *joint_state_controller* (encargado de publicar el estado de todas las articulaciones existentes), se cargan dos controladores de eje distinguidos por las palabras *in* y *out*, los cuales se asociarán a las parábolas de entrada y salida, respectivamente.

De la forma en la que se ha procedido en la simulación, el control mínimo se realiza sobre un solo eje, el cual contiene dos colectores. En la realidad, el control se realiza de forma individual para cada colector. Se ha decidido no proceder de esta forma en el trabajo debido a que esto implicaría duplicar el número de controladores presentes en la simulación, y conlleva una pérdida de rendimiento importante al generar numerosos modelos. Sin embargo, si se quisiera realizar, bastaría con extrapolar el método seguido anteriormente, desde un colector a un lazo, para añadir tantos ejes como se deseen.

5.3 Gestión de varios lazos

Para completar la generación de modelos en la simulación, el último paso es replicar el número de lazos en diferentes posiciones de acuerdo con la organización que presenta la planta termosolar al completo. Para ello, se aprovechará una característica de los archivos de lanzamiento de la simulación, y es que dentro de ellos es posible anidar otros archivos de lanzamiento, de forma que ejecutando el primero también se estén lanzando a la vez todos los que este incluya.

De esta forma, la idea pasa por diseñar un archivo de lanzamiento principal, en el cual se lance en primera instancia el mundo de la simulación, y seguidamente se ejecuten cada uno de los archivos de lanzamiento de un lazo, especificando para cada uno de ellos la posición concreta en la que se generarán dentro del mundo de la simulación. Se puede apreciar un resumen esquematizado del proceso de lanzamiento de la simulación en la Figura 5-13.

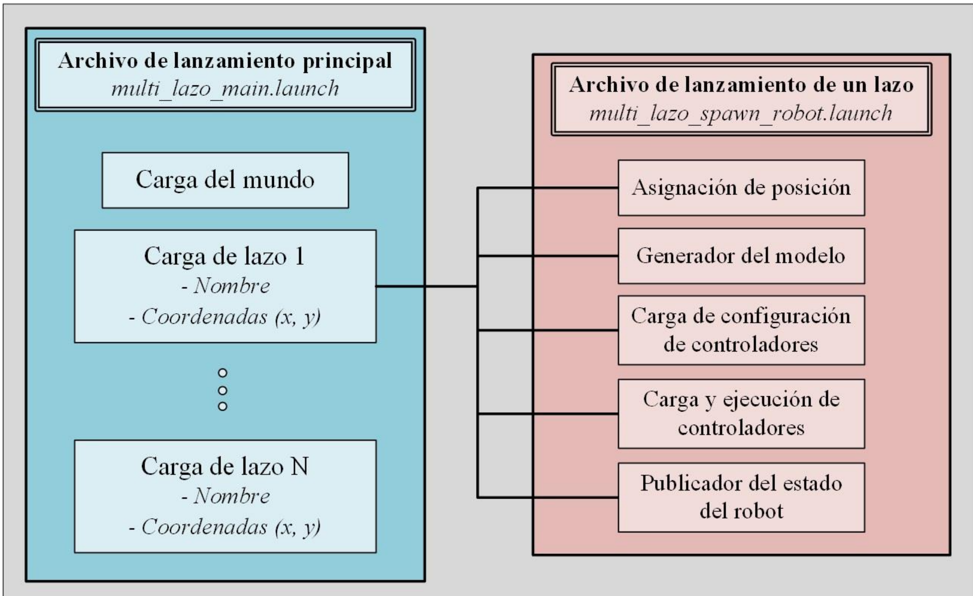


Figura 5-13. Esquema de lanzamiento de la simulación.

Sin embargo, la implementación de la idea planteada no es inmediata, y surgen errores al intentar inicializar una simulación con varios lazos. Esto es debido a que, al generar varios modelos iguales, también lo hacen los *topics* que se crean a partir de ese modelo. Es este el error del que apercibe ROS, ya que no es posible definir varios *topics* con el mismo nombre.

Como solución a dicho error, se propone la diversificación de cada uno de los lazos mediante el uso de espacio de trabajos en los archivos de lanzamiento (*namespaces*). El uso de esta etiqueta, englobando a todo el modelo individual de cada lazo, permite dotar a cada uno de ellos de un nombre concreto, el cual puede ser proporcionado como argumento desde el archivo principal al igual que sucede con la posición. Para ello, habrá que añadir el *namespace* en cada parámetro que afecte al modelo, como se puede apreciar en amarillo en la Figura 5-14.

```
<group ns="$(arg robot_name)"> <!-- lo ponemos en grupo para distinguir cuando tengamos +1 colector -->

    <!-- carga de config. de controladores -->
    <rosparam file="$(find ptcplant)/config/lazo_control.yaml" command="load" ns="/$(arg robot_name)" />

    <param name="/$(arg robot_name)/robot_description" command="cat $(arg urdf_robot_file)" />

    <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
            args="-urdf -x $(arg pos_x) -y $(arg pos_y) -z $(arg pos_z) -model $(arg robot_name) -param /$(arg


    <!-- carga (y ejecuta) los controladores. Elegimos los que queramos de los especificados en el config -->
    <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false" output="screen"
        args="--namespace=/$(arg robot_name) /$(arg robot_name)/joint_state_controller
                                             /$(arg robot_name)/patas_parabola_in_position_controller
                                             /$(arg robot_name)/patas_parabola_out_position_controller">
    </node>
```

Figura 5-14. Fragmento del nuevo archivo de lanzamiento de modelos usando *namespaces*.

Otro punto en el que es necesario intervenir si se aplica la medida anterior es en el modelo URDF del lazo. En dicho archivo, en la etiqueta `<plugin>`, se requiere escribir el espacio de trabajo del robot, dentro de `<robotNamespace>`. Si se especifica un nombre en este punto, se obtendrán mensajes de error, ya que los espacios de trabajo en URDF y archivo de lanzamiento serán discordantes. La solución para evitarlo es eliminar dicha línea en el archivo URDF, ya que de esta forma cargará como espacio por defecto el usado en el archivo de lanzamiento.

La distribución para denominar los diferentes lazos generados en la simulación sigue un orden por letras, según la fila a la que pertenezcan, y un orden numérico según su posición dentro de dicha fila. Por tanto, cuando se desee realizar una publicación para mover un determinado colector, será necesario determinar primero el nombre del lazo, y posteriormente si se trata del eje de entrada o el de salida. De esta forma, si se quisiera mover el eje de salida del tercer lazo de la fila "B" (ver Figura 5-15 para entender la organización), a la posición 0.5 rad, se usará el siguiente comando:

```
 >> rostopic pub -1
/lazoB3/patas_parabola_out_position_controller/command
std_msgs/Float64 "data: 0.5"
```

Dada la distribución de lazos empleada en la planta, y ya que se desea que estos se sitúan de forma simétrica respecto a la zona central de potencia, es necesario realizar modificaciones en los lazos de las filas "B" y "D" respecto al modelo original. Para ello, tal y como se comentó en el capítulo 4.5, se ha creado un modelo de lazo simétrico al original, que permite dar la distribución deseada a la planta.

Un problema que ha surgido en ocasiones puntuales, sin seguir un patrón determinado, es un comportamiento extraño en los modelos generados. De forma ocasional, en estos se produce un temblor en los extremos, independientemente de si se han generado solos o junto a otros modelos. Puntualmente, otro error ha sido la colisión entre modelos cuando se generan juntos.

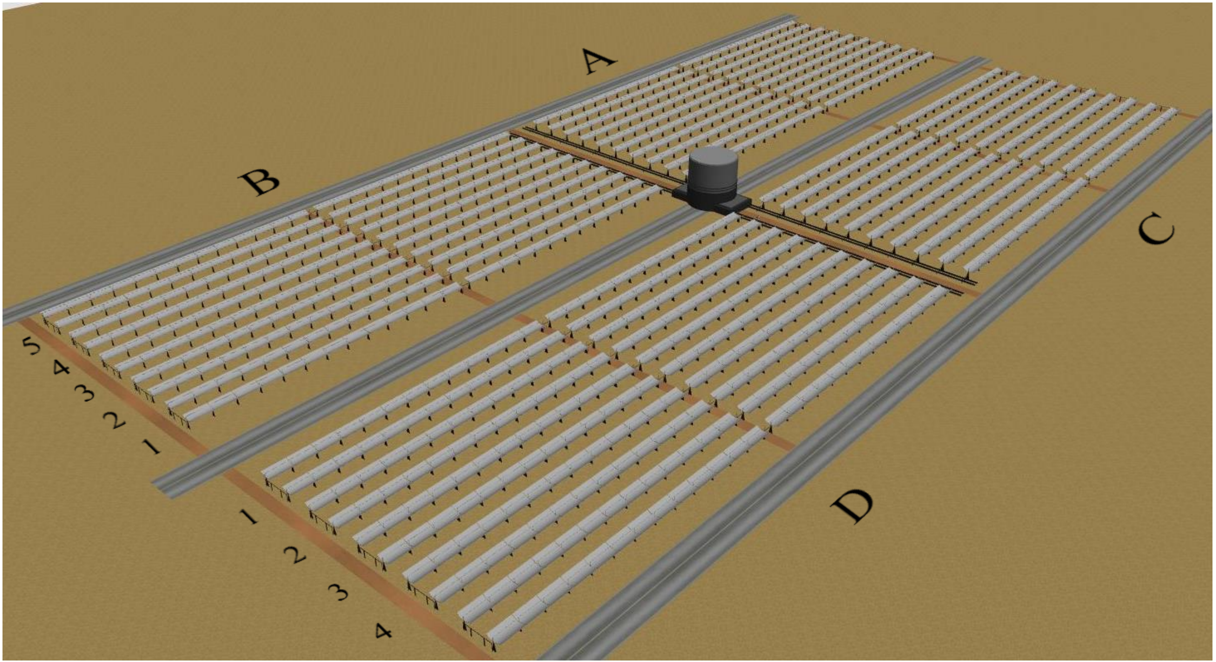


Figura 5-15. Organización de los lazos en la simulación.

Para solucionar el primero de los errores, ha sido necesario modificar las masas e inercias de los archivos URDF, con la consecuente resintonización de los controladores PID, ya que parece ser que el motor de físicas de Gazebo presenta algunos fallos. Por otro lado, para solventar el segundo fallo, se ha modificado la posición de cada uno de los modelos dejando una separación suficiente que evite la colisión entre ellos, así como se ha elevado en unos centímetros su posición para hacer que no presenten problemas al interponerse con el suelo.

Hasta este punto, se ha conseguido hacer funcionar correctamente una simulación de 20 lazos, distribuidos en 4 filas distintas. Los valores adoptados son los que permiten que, computacionalmente, la simulación sea fluida. En el caso de disponer de un PC con características superiores, es posible modificar fácilmente el número de lazos ejecutados, así como la distribución de estos, personalizando el archivo de lanzamiento principal, *multi_lazo_main.launch*.

Sin embargo, aunque los lazos están preparados para desarrollar un movimiento lo más fiel posible a la realidad, no se dispone de la posibilidad de hacerlos funcionar todos en paralelo de una forma automatizada, así como hacer que alguno de ellos mantenga una posición estática de cara a posibles reparaciones. Para ello, se intentarán integrar las órdenes de movimiento en un *script* de Python que gestione el movimiento de toda la planta.

5.4 Scripts de automatización del movimiento

De cara a automatizar los procesos y movimientos que se producen en la planta termosolar, lo ideal es que todas las órdenes se gestionen desde un mismo punto. Para ello, se ha trabajado con Python, y se han desarrollado una serie de *scripts* capaces de publicar y leer mensajes en los distintos *topics* activos mediante el uso de la librería *rospy*.

El programa principal lo compone *move_auto.py*, que se encarga de gestionar la lógica general de movimiento de todos los ejes. Por otro lado, *home_listener.py* hace las veces de sensor en la posición de reposo, teniendo el control de aquellos ejes que se han mandado a dicha posición. Por último, todo esto es posible a un archivo auxiliar llamada *topic_pub_generator.py*, que trata de identificar todos los *topics* activos en la simulación y clasificarlos para comunicarlo a las otras dos funciones.

Cada uno de estos scripts mencionados anteriormente se ejecutarán una vez la simulación se haya iniciado correctamente, y se hayan generado todos los modelos en ella. Formarán parte del sistema como nuevos nodos que propiciarán el intercambio de mensajes. Por ello, para ejecutarlos habrá que usar la orden `rosrun`.

5.4.1 Detección de *topics* activos

La comunicación general consiste en un intercambio de mensajes entre distintos nodos. Estos nodos serán tanto publicadores de la información, como lectores. Sin embargo, lo fundamental antes de todo es determinar cuál es el canal comunicativo, es decir, los *topics*. Para ello se crea en Python el script *topic_pub_generator.py*.

En un principio, si el número de modelos a controlar fuera un número fijo y conocido, se podrían declarar sus nombres en las variables correspondientes, lo que ahorraría trabajo de computación. Sin embargo, los programas desarrollados no tienen ningún tipo de restricción en ese sentido. Su diseño se ha dado con una orientación dinámica, es decir, el control automático de los modelos se hará indistintamente del número y nombre de ellos.

Para conseguir esto se crea el presente script. Gracias a la librería *rospy*, es posible el uso de la función `get_published_topics()` en Python. Esta es capaz de incorporar en una lista dada todos los *topics* activos en el momento de su ejecución. Junto a esto, se realiza un tratamiento de esta lista para saber cuáles de todos son *topics* que interesen en este momento para el intercambio de mensajes.

Todo esto se desarrolla dentro de la función *generator()*. Esta será posteriormente importada en los otros dos scripts, en los cuales se usará para poder saber cuál es el nombre de los topics donde se publicará o leerá información, algo obligatorio a la hora de definir un *publisher* o un *subscriber*.

5.4.2 Escucha en posición de reposo

Para dotar al sistema de un sensor en la posición de reposo, se crea el script *home_listener.py*. Este se encarga de crear los suscriptores necesarios, uno por eje, al topic `…/state` de cada controlador. De esta forma, en todo momento se estará conociendo la posición de estos.

A la hora de crear suscriptores, es necesario crear una función de tipo *callback* que desarrolle cierto código con los datos escuchados. Sin dicha función, la utilidad del nodo *subscriber* sería totalmente nula. Es en dicha función donde, implementando cierta lógica, es posible desencadenar acciones cuando se cumplan las condiciones de posición que impliquen estar en la posición reposo, como, por ejemplo, mandar un mensaje a la función de movimiento principal.

Una particularidad de esta función, y en concreto de la *callback*, es que es necesario conocer qué eje ha superado la condición implementada. Por ello, no basta con que el suscriptor tenga como argumento la posición, sino que también es necesario incluir como segundo argumento el nombre del topic, y, por tanto, del eje.

5.4.3 Movimiento general

Para la gestión principal de todo el movimiento se ha diseñado el archivo *move_auto.py*. En este se ha creado una clase llamada *Lazo*, a partir de la cual se crearán tantos objetos como ejes de giro se pretendan controlar (Figura 5-16). Por otro lado, aparte de los diferentes *publishers* y *subscribers* con sus respectivas funciones *callback*, se ha programado un bucle continuo mientras ROS esté activo, en el que se diseña toda la lógica de movimiento para los modelos.

```
class Lazo:
    def __init__(self, pub, pos_comm, home_comm, home_sens, home_going, finish, names):
        self.pub        =   pub         # publisher
        self.pos_comm   =   pos_comm    # orden de posicion
        self.home_comm  =   home_comm   # orden de ir a home
        self.home_sens  =   home_sens   # sensor home
        self.home_going =   home_going  # lazo yendo a home
        self.finish     =   finish      # lazo ha finalizado ciclo
        self.names      =   names       # nombre del lazo
```

Figura 5-16. Clase *Lazo* creada para registrar las características de cada eje.

Algunas de las variables que participan en el código permiten cambiar características del movimiento con tan solo modificarlas, como por ejemplo el incremento de posición entre iteraciones, o el tiempo que se ponen en reposo los colectores durante la noche, para reducir el tiempo equivalente en la simulación.

6 IMPLEMENTACIÓN DEL ROBOT DE INSPECCIÓN

Una vez finalizada con éxito la simulación de la planta termosolar al completo, parece oportuno introducir ciertos modelos que la complementen y aporten funciones adicionales al conjunto. Entre las principales ideas para ello, desde el comienzo ha destacado la introducción de un robot que desarrolle funciones de inspección en la planta. Para ello, se puede implementar desde un control manual básico, hasta la programación de rutas preestablecidas o verificación mediante sensores y cámaras.

Para la implementación del robot existen diversas posibilidades. Existe la posibilidad de usar un robot ya programado, descargándolo y usándolo en la simulación sin necesidad de modificar ningún parámetro. Por otro lado, en el otro extremo, estaría la opción de diseñar un robot desde cero, en el que se diseñe el modelo y se programe todo el movimiento, procediendo de igual forma que con los modelos diseñados para la planta.

En este caso, no se pretende que este apartado sea el centro del proyecto, sino más bien que siente las bases a futuras ampliaciones y diversificaciones. Por ello, para la implementación del robot en la simulación se ha optado por una solución alternativa a las mencionadas anteriormente. Se implementará un modelo muy básico de un robot con dos ruedas, al que se le aplicará un control diferencial para ambas, el cual requerirá de ciertas modificaciones para coordinarlo con el modelo en cuestión.

6.1 Modelado del robot

Para el diseño del robot se ha creado un modelo de este en un archivo URDF. En primer lugar, se ha empleado un rectángulo para hacer las funciones de chasis. A este se le han incorporado sendas ruedas en una zona algo retrasada respecto al eje medio del cuerpo, y se han unido a este mediante articulaciones de tipo continuo. Por otro lado, también se ha diseñado un apoyo adicional en la parte frontal, el cual consiste en una porción de esfera que sobresale bajo el chasis.

```
<gazebo>
    <plugin filename="libgazebo_ros_diff_drive.so" name="differential_drive_controller">
      <legacyMode>false</legacyMode>
      <alwaysOn>true</alwaysOn>
      <updateRate>20</updateRate>
      <leftJoint>joint_left_wheel</leftJoint>
      <rightJoint>joint_right_wheel</rightJoint>
      <wheelSeparation>0.6</wheelSeparation>
      <wheelDiameter>0.6</wheelDiameter>
      <torque>0.3</torque>
      <commandTopic>cmd_vel</commandTopic>
      <odometryTopic>odom</odometryTopic>
      <odometryFrame>odom</odometryFrame>
      <robotBaseFrame>link_chassis</robotBaseFrame>
    </plugin>
  </gazebo>
```

Figura 6-1. Plugin usado para el movimiento del robot y argumentos de entrada.

Debido al tipo de robot, y a que la coordinación con el plugin que le otorga movimiento debe ser plena, la elección de las masas, inercias y rozamientos es un factor crítico para el correcto funcionamiento de este. Además, existen ciertos factores geométricos que, de ser modificados en el robot, deben ser también correctamente referenciados en los argumentos de entrada del plugin, como se puede apreciar en la Figura 6-1.

Como se puede apreciar, el plugin requiere de ciertos valores de entrada para funcionar, como por ejemplo la separación entre las ruedas o el diámetro de ambas. Así mismo, también se puede especificar el nombre del *topic* en el cual se debe publicar la velocidad que se desea dar al robot. También aparecen otros argumentos, como los *topics* relacionados con la odometría, los cuales no tienen por qué ser usados si no se desea.

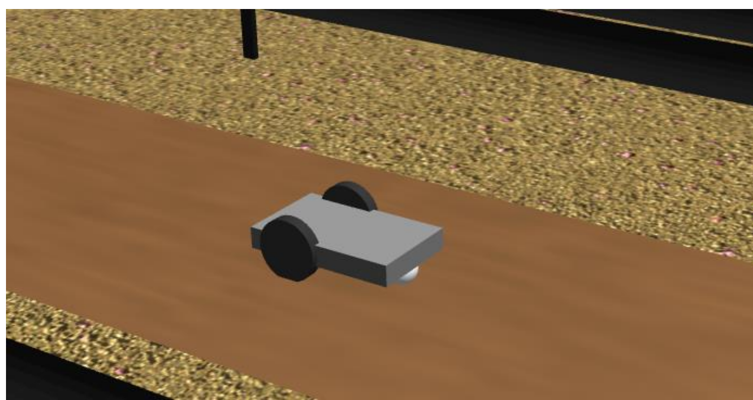


Figura 6-2. Aspecto visual del robot de inspección.

Otro punto para tener en cuenta es cómo proceder con la incorporación del robot a la simulación, tal y como aparece en la Figura 6-2. Lo ideal es integrarlo en el archivo de ejecución principal, de forma que el modelo se genere en la simulación al igual que los lazos. Para ello, es necesario incluir las líneas que se pueden ver en la Figura 6-3 en el archivo *multi_lazo_main.launch*:

```
<arg name="urdf_m2wr" default="$(find ptcplant)/urdf/big_m2wr.urdf" />
<param name="robot_description" command="cat $(arg urdf_m2wr)" />

<node name="m2wr_spawn" pkg="gazebo_ros" type="spawn_model" output="screen"
      args="-urdf -param robot_description -model m2wr -x 25 -y 0 -z 0.5" />
```

Figura 6-3. Parámetros y nodo de generación del robot de inspección.

6.2 Control manual del robot

Como se comentaba al comienzo del capítulo, y a pesar de que el robot está preparado para desempeñar funciones que van más allá del control manual, en este punto se explicará cómo se ha conseguido implementar de una forma sencilla el manejo por teclado del robot para su uso en paralelo a la simulación en pleno funcionamiento.

La forma usual de controlar por teclado ciertos modelos en ROS es mediante el uso de la librería *teleop_twist_keyboard*, la cual se proporciona dentro de ROS y puede ser instalada desde el terminal sin complicaciones. Para ello, empleamos el siguiente comando:

```
>> sudo apt-get install ros-melodic-teleop-twist-keyboard
```

Cabe destacar que este comando variará en función de la versión de ROS instalada. En caso de tener otra versión, esta deberá reemplazar en el comando a *melodic*. Tras la ejecución, y si no hay mensajes de error, esta quedará instalada correctamente.

El control por teclado es un script en Python, que al ser ejecutado desde el terminal permite asociar ciertas teclas a las funciones básicas del robot (Figura 6-4). En este caso, permite realizar movimientos hacia delante y detrás, movimientos de rotación en estático y movimientos de rotación en curva. Además, también habilita ciertas teclas que permiten cambiar la velocidad del robot, tanto lineal como angular.

Para comenzar a usarlo, es necesario esperar a que el robot se genere en la simulación y quede habilitado para ser controlado. A partir de ese instante, desde un nuevo terminal, se usa la siguiente orden:

```
>> rosrun teleop-twist-keyboard teleop-twist-keyboard.py
```

Esta orden introducirá un nuevo nodo en la simulación para gestionar el control. Una vez ejecutada, se recibe por pantalla toda la información necesaria para mover el robot. Como es de esperar, es necesario tener abierto el terminal para poder controlar el robot por teclado. De lo contrario, el terminal no puede registrar las teclas pulsadas para mover el robot (ROS Wiki, 2018).

```
edumbriz@edumbriz-GT70-2PC:~$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py

Reading from the keyboard  and Publishing to Twist!
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently:      speed 0.5       turn 1.0
```

Figura 6-4. Control manual por teclado del robot de inspección.

7 CONCLUSIONES Y LÍNEAS FUTURAS

El desarrollo del presente trabajo ha planteado retos y dificultades de diversa naturaleza, los cuales han llegado a implicar mucho trabajo en vano, o bien en una dirección que no era del todo la correcta para el objetivo deseado. Los temas tratados desde el principio hasta el final del proyecto requieren de un aprendizaje muy profundo desde el comienzo. Por otro lado, a medida que se avanza en los diferentes pasos del proyecto, siempre se encuentran nuevas formas de proceder que superan tanto en sencillez como en eficiencia a las anteriores.

Aunque no se dispone de métricas para analizar el rendimiento general de la simulación, más allá de las que aporta Gazebo, parece lógico pensar que la dimensión de la planta y los futuros modelos para añadir a esta dependerán en gran medida de la capacidad computacional del PC que ejecute la simulación. En el actual, y con las dimensiones de planta comentadas en el trabajo, se llega a alcanzar un factor de tiempo real cercano al 0.8, lo cual indica que la simulación no es capaz de moverse en tiempo real. En el mejor de los casos, este cociente debería ser 1, lo que implicaría que los tiempos de simulación y real son coincidentes.

Otro aspecto para tener en cuenta es la complejidad de los modelos incorporados a Gazebo. Si bien desde un comienzo se ideó diseñar un módulo lo más detallado posible, el software de la simulación parece funcionar mucho mejor con mallados con menor cantidad de puntos. Desde este punto de vista, mientras más se simplifiquen los modelos, así como las mallas de las colisiones, mejor rendimiento y fluidez se obtendrán en la simulación.

Este trabajo sienta las bases para una futura ampliación de la planta, tanto en número de lazos, como en diferentes robots para inspección y diversas tareas. La idea es poder aprovechar que el movimiento de la planta está completamente automatizado, así como también es posible detectar en todo momento si los modelos introducidos colisionan con algún elemento perteneciente a la planta. Todos los modelos en esta han sido diseñados de forma que su malla es totalmente coincidente con su visualización.

Como ideas generales, se podría añadir un robot más elaborado que realice rutas programadas y tareas de mantenimiento. Existen diversos sensores y cámaras en ROS que permitirían dotar a cualquier robot de estas funcionalidades. Así mismo, siguiendo las pautas mencionadas en el capítulo 6, la inclusión de cualquier elemento diseñado en la simulación es casi trivial.

ANEXO A. CÓDIGO IMPLEMENTADO

A continuación, se expondrán los fragmentos de código más destacados que permiten ejecutar la simulación. Estos irán escritos en distintos lenguajes, sobre todo XML y Python, y se complementan con más código que no se ha añadido debido a que no se considera esencial.

A.1 Lanzamiento de la simulación: multi_lazo_main.launch

```
<?xml version="1.0" encoding="UTF-8"?>

<launch>
    <arg name="robot" default="machines"/>
    <arg name="debug" default="false"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="pause" default="false"/>
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
          <arg name="world_name" value="$(find
ptcplant)/worlds/simple.world"/>
          <arg name="paused" value="$(arg pause)"/>
          <arg name="use_sim_time" value="true"/>
          <arg name="gui" value="$(arg gui)"/>
          <arg name="headless" value="$(arg headless)"/>
          <arg name="debug" value="$(arg debug)"/>
    </include>

    <!-- spawn de los lazos -->

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoA5" />
      <arg name="pos_x" value="-110" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoB5" />
      <arg name="pos_x" value="-110" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoA4" />
      <arg name="pos_x" value="-90" />
      <arg name="pos_y" value="125" />
    </include>
```

```
    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoB4" />
      <arg name="pos_x" value="-90" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoA3" />
      <arg name="pos_x" value="-70" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoB3" />
      <arg name="pos_x" value="-70" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoA2" />
      <arg name="pos_x" value="-50" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoB2" />
      <arg name="pos_x" value="-50" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoA1" />
      <arg name="pos_x" value="-30" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoB1" />
      <arg name="pos_x" value="-30" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoC1" />
      <arg name="pos_x" value="20" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoD1" />
      <arg name="pos_x" value="20" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoC2" />
```

```
      <arg name="pos_x" value="40" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoD2" />
      <arg name="pos_x" value="40" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoC3" />
      <arg name="pos_x" value="60" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoD3" />
      <arg name="pos_x" value="60" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoC4" />
      <arg name="pos_x" value="80" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoD4" />
      <arg name="pos_x" value="80" />
      <arg name="pos_y" value="-125" />
    </include>

    <include file="$(find ptcplant)/launch/multi_lazo_spawn_robot.launch">
      <arg name="robot_name" value="lazoC5" />
      <arg name="pos_x" value="100" />
      <arg name="pos_y" value="125" />
    </include>

    <include file="$(find
ptcplant)/launch/multi_lazo_espejo_spawn_robot.launch">
      <arg name="robot_name" value="lazoD5" />
      <arg name="pos_x" value="100" />
      <arg name="pos_y" value="-125" />
      <arg name="pos_z" value="0.5" />
    </include>

      <!-- spawn del robot de dos ruedas, para pruebas de inspeccion -->

    <arg name="urdf_m2wr" default="$(find ptcplant)/urdf/big_m2wr.urdf" />
    <param name="robot_description" command="cat $(arg urdf_m2wr)" />

    <node name="m2wr_spawn" pkg="gazebo_ros" type="spawn_model"
output="screen"
          args="-urdf -param robot_description -model m2wr -x 25 -y 0 -z 0.5"
/>

</launch>
```

A.2 Generación de un modelo de lazo: multi_lazo_main.launch

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>

      <!--  cargamos el modelo en el param. server con el nombre 
robot_description: -->
      <arg name="urdf_robot_file" default="$(find ptcplant)/urdf/lazo.urdf" 
/>
      <param name="robot_description" command="cat $(arg urdf_robot_file)" />

      <arg name="robot_name" default="lazo" />

      <arg name="pos_x" default="0.0" />
      <arg name="pos_y" default="0.0" />
      <arg name="pos_z" default="0.0" />

    <group ns="$(arg robot_name)"> <!-- lo ponemos en grupo para distinguir 
cuando tengamos +1 colector -->


      <!-- carga de config. de controladores -->
      <rosparam file="$(find ptcplant)/config/lazo_control.yaml" 
command="load" ns="/$(arg robot_name)" />

      <param name="/$(arg robot_name)/robot_description" command="cat $(arg 
urdf_robot_file)" />

      <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" 
respawn="false" output="screen"
                  args="-urdf -x $(arg pos_x) -y $(arg pos_y) -z $(arg pos_z) 
-model $(arg robot_name) -param /$(arg robot_name)/robot_description"/>


      <!-- carga (y ejecuta) los controladores. Elegimos los que queramos de 
los especificados en el config -->
      <node name="controller_spawner" pkg="controller_manager" type="spawner" 
respawn="false" output="screen"
            args="--namespace=/$(arg robot_name) /$(arg 
robot_name)/joint_state_controller
                                                                    /$(arg 
robot_name)/patas_parabola_in_position_controller
                                                                    /$(arg 
robot_name)/patas_parabola_out_position_controller">
      </node>


      <!-- robot state publisher. nos servira para visualizar el robot en 
rViz puntualmente-->

      <!--  <node name="robot_state_publisher" pkg="robot_state_publisher" 
type="robot_state_publisher" respawn="false" output="screen">
            <param name="publish_frequency" type="double" value="5.0" />
      </node>
      -->

    </group>

</launch>
```

A.3 Mundo de la simulación (SDF): simple.world

```
<?xml version="1.0" ?>
<sdf version="1.6">

  <world name="simple">

       <physics name="fisicas_agiles" type="ode">
         <max_step_size>0.005</max_step_size>
       <real_time_factor>1</real_time_factor>
       <real_time_update_rate>200</real_time_update_rate>
       <ode>
        <solver>
          <type>quick</type>
          <iters>50</iters>
          <sor>1.3</sor>
        </solver>
       </ode>
       </physics>

       <scene>
       <ambient>0.4 0.4 0.4 1</ambient>
       <background>0.7 0.7 0.7 1</background>
       <shadows>0</shadows>
         <sky>
            <clouds>
                  <speed>7</speed>
                  <humidity>5</humidity>
                  <mean_size>0.3</mean_size>
            </clouds>
      </sky>
      <grid>0</grid>
      <origin_visual>0</origin_visual>
    </scene>

    <!-- modelo de suelo personalizado con texturas -->
    <include>
      <uri>model://ground</uri>
      <name>my_ground</name>
    </include>

    <!-- modelo de suelo/carretera personalizado con texturas para las
tuberias -->
    <include>
      <uri>model://pipe_path</uri>
      <name>left_pipe_path</name>
      <pose>0 -233 0.02 0 0 0</pose>
    </include>

    <include>
      <uri>model://pipe_path</uri>
      <name>mid_left_pipe_path</name>
      <pose>0 -118.93 0.02 0 0 0</pose>
    </include>

    <include>
      <uri>model://pipe_path</uri>
      <name>mid_pipe_path</name>
      <pose>0 0 0.02 0 0 0</pose>
    </include>
```

```
<include>
  <uri>model://pipe_path</uri>
  <name>mid_right_pipe_path</name>
  <pose>0 118.93 0.02 0 0 0</pose>
</include>

<include>
  <uri>model://pipe_path</uri>
  <name>right_pipe_path</name>
  <pose>0 233 0.02 0 0 0</pose>
</include>

<!-- modelos de carretera personalizado con texturas para vehiculos -->
<include>
  <uri>model://road</uri>
  <name>low_road</name>
  <pose>125 0 0.04 0 0 0</pose>
</include>

<include>
  <uri>model://road</uri>
  <name>mid_road</name>
  <pose>0 0 0.04 0 0 0</pose>
</include>

<include>
  <uri>model://road</uri>
  <name>high_road</name>
  <pose>-125 0 0.04 0 0 0</pose>
</include>


<model name="zona_potencia">
  <pose>0 0 0 0 0 0</pose>
  <static>true</static>
  <link name="base">
        <self_collide>0</self_collide>
        <pose>0 0 10 0 0 0</pose>
          <inertial>
            <inertia>
              <ixx>1000</ixx>
              <ixy>0</ixy>
              <ixz>0</ixz>
              <iyy>1000</iyy>
              <iyz>0</iyz>
              <izz>1000</izz>
            </inertia>
            <mass>10000.0</mass>
          </inertial>

          <!-- diseño basico de central de potencia -->
          <collision name="base_collision">
              <pose>0 0 -10 0 0 0</pose>
              <geometry>
                <box>
                  <size>29.8 15 6</size>
                </box>
              </geometry>
            </collision>
          <visual name="base_visual">
              <pose>0 0 -10 0 0 0</pose>
              <geometry>
```

```
        <box>
          <size>30 15 6</size>
        </box>
      </geometry>
      <material>
            <ambient>0.2 0.2 0.2 1</ambient>
          <diffuse>0.1 0.1 0.1 1</diffuse>
          <specular>0.05 0.05 0.05 1</specular>
          <emissive>0 0 0 0</emissive>
        </material>
    </visual>
  <visual name="base_visual_2">
      <pose>0 0 -8.5 0 0 0</pose>
      <geometry>
        <box>
          <size>28 12 4</size>
        </box>
      </geometry>
      <material>
            <ambient>0.3 0.3 0.3 1</ambient>
          <diffuse>0.3 0.3 0.3 1</diffuse>
          <specular>0.1 0.1 0.1 1</specular>
          <emissive>0 0 0 0</emissive>
        </material>
    </visual>

  <collision name="cyl_collision">
      <pose>0 0 0 0 0 0</pose>
      <geometry>
        <cylinder>
          <radius>10</radius>
          <length>20</length>
        </cylinder>
      </geometry>
    </collision>
  <visual name="cyl_visual">
      <pose>0 0 0 0 0 0</pose>
      <geometry>
        <cylinder>
          <radius>10</radius>
          <length>20</length>
        </cylinder>
      </geometry>
      <material>
            <ambient>0.35 0.35 0.35 1</ambient>
          <diffuse>0.35 0.35 0.35 1</diffuse>
          <specular>0.15 0.15 0.15 1</specular>
          <emissive>0 0 0 0</emissive>
        </material>
    </visual>

    <collision name="cyl_collision_2">
      <pose>0 0 10 0 0 0</pose>
      <geometry>
        <cylinder>
          <radius>9.5</radius>
          <length>0.6</length>
        </cylinder>
      </geometry>
    </collision>
  <visual name="cyl_visual_2">
      <pose>0 0 10 0 0 0</pose>
      <geometry>
```

```
        <cylinder>
          <radius>9.5</radius>
          <length>0.6</length>
        </cylinder>
      </geometry>
      <material>
            <ambient>0.5 0.5 0.5 1</ambient>
          <diffuse>0.45 0.45 0.45 1</diffuse>
          <specular>0.15 0.15 0.15 1</specular>
          <emissive>0 0 0 0</emissive>
        </material>
    </visual>

    <collision name="aro_grande_collision">
      <pose>0 0 -6 0 0 0</pose>
      <geometry>
        <cylinder>
          <radius>10.2</radius>
          <length>8</length>
        </cylinder>
      </geometry>
    </collision>
  <visual name="aro_grande_visual">
      <pose>0 0 -6 0 0 0</pose>
      <geometry>
        <cylinder>
          <radius>10.2</radius>
          <length>8</length>
        </cylinder>
      </geometry>
      <material>
            <ambient>0.2 0.2 0.2 1</ambient>
          <diffuse>0.1 0.1 0.1 1</diffuse>
          <specular>0.05 0.05 0.05 1</specular>
          <emissive>0 0 0 0</emissive>
        </material>
    </visual>

  <visual name="aro_visual">
      <pose>0 0 -1 0 0 0</pose>
      <geometry>
        <cylinder>
          <radius>10.2</radius>
          <length>0.3</length>
        </cylinder>
      </geometry>
      <material>
            <ambient>0.2 0.2 0.2 1</ambient>
          <diffuse>0.1 0.1 0.1 1</diffuse>
          <specular>0.05 0.05 0.05 1</specular>
          <emissive>0 0 0 0</emissive>
        </material>
    </visual>
  <visual name="aro_visual_2">
      <pose>0 0 0 0 0 0</pose>
      <geometry>
        <cylinder>
          <radius>10.2</radius>
          <length>0.3</length>
        </cylinder>
      </geometry>
      <material>
```

```
                        <ambient>0.2 0.2 0.2 1</ambient>
                      <diffuse>0.1 0.1 0.1 1</diffuse>
                      <specular>0.05 0.05 0.05 1</specular>
                      <emissive>0 0 0 0</emissive>
                    </material>
                </visual>

          </link>
    </model>

      <light type="directional" name="sol">
          <cast_shadows>false</cast_shadows>
          <pose>0 0 10 0 0 0</pose>
          <diffuse>0.8 0.8 0.8 1</diffuse>
          <specular>0.2 0.2 0.2 1</specular>
          <attenuation>
            <range>1000</range>
            <constant>0.9</constant>
            <linear>0.01</linear>
            <quadratic>0.001</quadratic>
          </attenuation>
          <direction>-0.5 0.1 -0.9</direction>
    </light>

    <gui>
      <camera name="camara">
        <pose>100 -150 90 0 0.5 2</pose>
      </camera>
    </gui>

  </world>

</sdf>
```

A.4 Modelo de lazo (URDF): lazo.urdf

```
<robot name="lazo">

  <link name="link_patas">
      <inertial>
            <mass value="100" />
        <inertia ixx="4000" ixy="0.0" ixz="0.0" iyy="4000" iyz="0.0"
izz="4000"/>
      </inertial>
      <collision>
      <geometry>
        <mesh filename="package://ptcplant/urdf/meshes/lazo/patas_lazo.stl"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>
    <visual>
      <geometry>
        <mesh filename="package://ptcplant/urdf/meshes/lazo/patas_lazo.stl"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </visual>
  </link>


  <link name="link_parabola_in">
    <inertial>
        <mass value="30" />
        <inertia ixx="150" ixy="0.0" ixz="0.0" iyy="150" iyz="0.0" izz="150"
/>
    </inertial>
    <collision>
      <geometry>
        <mesh
filename="package://ptcplant/urdf/meshes/lazo/parabola_lazo_in.stl"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 -2.7"/>
    </collision>
    <visual>
      <geometry>
        <mesh
filename="package://ptcplant/urdf/meshes/lazo/parabola_lazo_in.stl"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 -2.7"/>
    </visual>
  </link>

  <joint name="joint_patas_a_parabola_in" type="revolute">
    <parent link="link_patas"/>
    <child link="link_parabola_in"/>
    <axis xyz="0 1 0"/>
    <origin rpy="0 0 0" xyz="0 0 2.7"/>
    <limit effort="1000.0" lower="-1.6" upper="1.6" velocity="0.1"/>
  </joint>


  <link name="link_parabola_out">
    <inertial>
        <mass value="30" />
        <inertia ixx="150" ixy="0.0" ixz="0.0" iyy="150" iyz="0.0" izz="150"
/>
```

```
    </inertial>
    <collision>
      <geometry>
        <mesh
filename="package://ptcplant/urdf/meshes/lazo/parabola_lazo_out.stl"/>
      </geometry>
      <origin rpy="0 0 0" xyz="-10 0 -2.7"/>
    </collision>
    <visual>
      <geometry>
        <mesh
filename="package://ptcplant/urdf/meshes/lazo/parabola_lazo_out.stl"/>
      </geometry>
      <origin rpy="0 0 0" xyz="-10 0 -2.7"/>
    </visual>
  </link>

  <joint name="joint_patas_a_parabola_out" type="revolute">
    <parent link="link_patas"/>
    <child link="link_parabola_out"/>
    <axis xyz="0 1 0"/>
    <origin rpy="0 0 0" xyz="10 0 2.7"/>
    <limit effort="1000.0" lower="-1.6" upper="1.6" velocity="0.1"/>
  </joint>


  <gazebo reference="link_patas">
    <material>Gazebo/FlatBlack</material>
  </gazebo>

  <gazebo reference="link_parabola_in">
    <material>Gazebo/Grey</material>
  </gazebo>

  <gazebo reference="link_parabola_out">
    <material>Gazebo/Grey</material>
  </gazebo>

  <gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    </plugin>
  </gazebo>


  <transmission name="trans_patas_a_parabola_in">
    <type>transmission_interface/SimpleTransmission</type>
    <actuator name="motor_rotacion">
      <mechanicalReduction>1</mechanicalReduction>
      <hardwareInterface>EffortJointInterface</hardwareInterface>
    </actuator>
    <joint name="joint_patas_a_parabola_in">

<hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface
>
    </joint>
  </transmission>

  <transmission name="trans_patas_a_parabola_out">
    <type>transmission_interface/SimpleTransmission</type>
    <actuator name="motor_rotacion">
      <mechanicalReduction>1</mechanicalReduction>
      <hardwareInterface>EffortJointInterface</hardwareInterface>
    </actuator>
```

```
    <joint name="joint_patas_a_parabola_out">
<hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface
>
    </joint>
  </transmission>

</robot>
```

A.5 Script para sensor en modo espera: home_listener.py

```
import rospy
from std_msgs.msg._Float64 import Float64
from std_msgs.msg import Bool
from sensor_msgs.msg import JointState
from control_msgs.msg._JointControllerState import JointControllerState
from topic_pub_generator import generator

def joint_states_callback(data, arg):

    topic = arg[0]
    sub_list = arg[1]

    for x in sub_list:
        if topic[1:7] in x and topic[0:40] == x[0:40]:
            if '_in_' in x:
                pub_topic = topic[0:7] + 'home_sens_in'
            elif '_out_' in x:
                pub_topic = topic[0:7] + 'home_sens_out'

    pub = rospy.Publisher(pub_topic, Bool, queue_size=100)

    if data.process_value >= 1.49:
        home = True
        #rospy.loginfo_once("Listener: home reached!")
        pub.publish(home)
    else:
        home = False
        pub.publish(home)


def listener():


    sub_str = generator()[1]

    rospy.init_node('listener', anonymous=True)


    # creacion de subscribers, uno por cada eje de giro (2 por lazo en
Gazebo)
    # escuchan la posicion real de cada eje de giro
    for y in sub_str:
        # y: /LazoXY/patas_parabola_(in-out)_position_controller/state
        rospy.Subscriber(y, JointControllerState, joint_states_callback, (y,
sub_str))


    # spin() evita una salida de python hasta que se finalice el nodo
    rospy.spin()



if __name__ == '__main__':
    try:
        listener()
    except rospy.ROSInterruptException:
        pass
```

A.6 Script para generación de topics: topic_pub_generator.py

```
import rospy

def generator():

    topic_list=rospy.get_published_topics()
    topic_list.sort()

    # topics contiene los lazos duplicados
    topics = []

    for x in topic_list:
        if "lazo" and "joint" in x[0]:
            topics.append(x[0][1:7])

    lazos = list(set(topics))
    lazos.sort()

    pub_str = []
    for x in topics:
        pub_str.append(x + '/patas_parabola_in_position_controller/command')
        pub_str.append(x + '/patas_parabola_out_position_controller/command')

    sub_str = []
    home_str = []
    home_comm = []
    names =[]

    for y in lazos:
        sub_str.append(y + '/patas_parabola_in_position_controller/state')
        sub_str.append(y + '/patas_parabola_out_position_controller/state')
        home_str.append(y + '/home_sens_in')
        home_str.append(y + '/home_sens_out')
        home_comm.append(y + '/home_comm_in')
        home_comm.append(y + '/home_comm_out')
        names.append(y + '_in')
        names.append(y + '_out')

    # strings con todos los topics necesarios

    return home_str, sub_str, pub_str, home_comm, names
```

A.7 Script para automatización de movimiento: move_auto.py

```
import rospy
from std_msgs.msg._Float64 import Float64
from std_msgs.msg import Bool
from sensor_msgs.msg import JointState
from control_msgs.msg._JointControllerState import JointControllerState
from topic_pub_generator import generator

class Lazo:
    def __init__(self, pub, pos_comm, home_comm, home_sens, home_going,
finish, names):
        self.pub        =   pub         # publisher
        self.pos_comm   =   pos_comm    # orden de posicion
        self.home_comm  =   home_comm   # orden de ir a home
        self.home_sens  =   home_sens   # sensor home
        self.home_going =   home_going  # lazo yendo a home
        self.finish     =   finish      # lazo ha finalizado ciclo
        self.names      =   names       # nombre del lazo


def home_callback(data, arg):
    topic = arg[0]
    home_list = arg[1]
    pubs = arg[2]
    found_index=False

    for i, x in enumerate(home_list):
        if topic in x:
            index=i
            found_index=True
            if data.data == True:
                pubs[index].home_sens = True
            else:
                pubs[index].home_sens = False

def home_comm_callback(data, arg):
    topic = arg[0]
    home_comm_list = arg[1]
    pubs = arg[2]
    found_index=False

    for i, x in enumerate(home_comm_list):
        if topic in x:
            index=i
            found_index=True
            if data.data == True:
                pubs[index].home_comm = True
            else:
                pubs[index].home_comm = False

def move():

    rospy.init_node('move', anonymous=True)

    pub_str = generator()[2]
    home_str = generator()[0]
    home_comm = generator()[3]
    names = generator()[4]
```

```
    # creacion lista de publishers de clase Lazo, una por eje de giro (2 por
lazo en Gazebo)
    pubs = []
    for i, x in enumerate(pub_str):
        pubs.append(Lazo(rospy.Publisher(x, Float64, queue_size=100) , 0.0,
False, False, False, False, names[i]))

    # creacion de subscribers al sensor home, uno por eje de giro (2 por lazo
en Gazebo)
    for y in home_str:
        rospy.Subscriber(y, Bool, home_callback, (y, home_str, pubs))

    # creacion de subscribers home_comm, uno por eje de giro (2 por lazo en
Gazebo)
    for z in home_comm:
        rospy.Subscriber(z, Bool, home_comm_callback, (z, home_comm, pubs))

    rate = rospy.Rate(0.5) # 0.5hz, cada 2 segundos

    num_lazos_fin = 0
    daycount = 1
    dormir = 0.5

    while not rospy.is_shutdown():
        for lazo in pubs:
            # casuistica para cada eje de lazo:

            # mvto automatico en condiciones normales, siguiendo al Sol
            if lazo.pos_comm>-1.5 and lazo.home_comm==False and
lazo.finish==False and lazo.home_going==False:
                lazo.pos_comm -= 0.05
                lazo.pub.publish(lazo.pos_comm)

            # vuelta a posicion Home tras acabar un dia
            elif lazo.pos_comm<-1.5 and lazo.home_comm==False and
lazo.finish==False:
                rospy.loginfo("%s -->\t Ciclo diario finalizado, volviendo a
home",lazo.names)
                lazo.home_going=True
                lazo.pos_comm=1.5
                lazo.pub.publish(lazo.pos_comm)

            # se ha mandado un lazo a reparar o ya está yendo a home
            if lazo.home_comm==True or lazo.home_going==True:
                if lazo.finish==False:
                    if lazo.home_sens==False:
                        rospy.loginfo("%s -->\t Mandando colector a
home",lazo.names)
                        lazo.pos_comm=1.5
                        lazo.pub.publish(lazo.pos_comm)
                    else:
                        lazo.finish=True
                        lazo.home_going=False
                        rospy.loginfo("%s -->\t Dia %d finalizado. Colector
reparando",lazo.names, daycount)
                        num_lazos_fin += 1

            # el lazo ha llegado al final
            if lazo.home_sens==True and lazo.home_going==True and
lazo.finish==False:
                lazo.finish=True
                lazo.home_going=False
```

```
                rospy.loginfo("%s -->\t Dia %d finalizado. 
Durmiendo."%(lazo.names, daycount))
                num_lazos_fin += 1

            # espera que lleguen todos los lazos al final, y resetea
            if num_lazos_fin==len(pubs):
                for x in pubs:
                    x.finish=False
                    x.home_going=False
                rospy.loginfo("Dia %d finalizado. Durmiendo."%(daycount))
                daycount += 1
                num_lazos_fin = 0
                rospy.sleep(dormir)
                rospy.loginfo("Comienza un nuevo día!")

            # publicación de info general
            rospy.loginfo("%s -->\t pos_comm: %0.2f,  home_comm: %d, 
home_going: %d,  finish: %d" %(lazo.names, lazo.pos_comm, lazo.home_comm, 
lazo.home_going, lazo.finish))
            rospy.loginfo("%s -->\t num_lazos_fin: %d,  daycount: %d" 
%(lazo.names, num_lazos_fin, daycount))
            print(" ")
        rate.sleep()

if __name__ == '__main__':
    try:
        move()
    except rospy.ROSInterruptException:
        pass
```

REFERENCIAS

Blender Reference Manual. (octubre de 2021). Obtenido de Blender Reference Manual: https://docs.blender.org/manual/es/latest/

Develop Paper. (2020). Obtenido de https://developpaper.com/study-notes-of-autolabor-2-5-3-working-space-and-compiling-system-of-ros/

Geek Gasteiz. (1 de noviembre de 2018). Obtenido de https://geekgasteiz.wordpress.com/2018/11/01/ros2-vs-ros-1-migramos/

Joseph, L. (2017). *ROS Robotics Projects.* Packt Publishing.

La Energía Solar. (2019). Obtenido de https://www.laenergiasolar.org/energia-termica-solar/

OGRE Wiki. (6 de julio de 2012). Obtenido de https://wiki.ogre3d.org/Materials

Protermosolar. (2020). Obtenido de https://helioscsp.com/concentrated-solar-power-skyfuel-completes-efficiency-testing-of-the-skytrough-dsp-collector/

Quigley, M., Gerkey, B., & D. Smart, W. (2016). *Programming Robots with ROS.* O'Reilly.

ROS Wiki. (8 de agosto de 2018). Obtenido de http://wiki.ros.org/

The Robotics Back-End. (2021). Obtenido de https://roboticsbackend.com/rqt-graph-visualize-and-debug-your-ros-graph/

Wikipedia. (21 de agosto de 2021). Obtenido de https://en.wikipedia.org/wiki/Gazebo_simulator

GLOSARIO

APIs: Application Programming Interface (Interfaz de Programación de Aplicaciones) 7

FPS: Frames Per Second (Imágenes Por Segundo) 20

LTS: Long Term Support (Soporte a Largo Plazo) 6

ODE: Open Dynamics Engine (Motor de Dinámicas Abierto) 15

OGRE: Object Oriented Graphics Rendering Engine 22

POSIX: Portable Operating System Interface 6

ROS: Robot Operating System 3

SDF: Simulation Description Format (Formato de Descripción de Simulación) 11

STL: STereo Lithography 21

TCP/IP: Transmission Control Protocol/Internet Protocol 8

UDP: User Datagram Protocol 8

URDF: Universal Robot Description File (Archivo de Descripción Universal de Robot) 11

YAML: YAML Ain't Markup Language 46