Proyecto Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Simulación en Gazebo de quadrotor para transporte de carga colgante

Autor: Pablo Gómez-Cambronero Martín
Tutor: Manuel Gil Ortega Linares
Manuel Vargas Villanueva

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

Sevilla, 2017

Proyecto Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Simulación en Gazebo de quadrotor para transporte de carga colgante

Autor:
Pablo Gómez-Cambronero Martín

Tutor:
Manuel Gil Ortega Linares
Profesor titular
Manuel Vargas Villanueva
Profesor titular

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Proyecto Fin de Carrera: Simulación en Gazebo de quadrotor para transporte de carga colgante

Autor: Pablo Gómez-Cambronero Martín

Tutor: Manuel Gil Ortega Linares
Manuel Vargas Villanueva

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis maestros

A mis amigos

Agradecimientos

En primer lugar, me gustaría agradecer a mis tutores, Manuel Vargas y Manuel Gil, la confianza depositada en mi, permitiendo mi incorporación a un proyecto como este, el cual ha ampliado en gran medida mis conocimientos. También agradezco toda la atención y los recursos prestados durante el desarrollo del mismo.

Gracias a compañeros como Mario Jimenez y Alfredo, por haber servido de apoyo en determinados aspectos del trabajo.

Sobre todo, gracias a Jesús Lozano, a quién considero un magnífico profesional en la materia, por la gran ayuda prestada, por su paciencia y por abrirme los ojos a un mundo tan extenso como es el de los vehículos aéreos no tripulados.

Gracias a mi familia, quienes desde el comienzo de estos duros cuatro años han sabido apoyarme, guiarme y ayudarme a tomar las decisiones que al final de este trayecto considero han sido las mejores. Gracias por siempre estar ahí y por enseñarme a nunca darme por vencido.

Finalmente, agradecer también a mis amigos, a los de siempre, por animarme como solo ellos saben hacer, y a los que han ido surgiendo en el transcurso de esta etapa universitaria, por ser tan fieles compañeros de viaje y hacer más llevaderos todos esos momentos complicados.

Pablo Gómez-Cambronero Martín

Escuela Técnica Superior de Ingeniería

Sevilla, 2017

Resumen

Los quadrotors han revolucionado el mundo de los vehículos aéreos no tripulados desde su aparición. Han transformado tareas que podían suponer algún riesgo para las personas en procesos sencillos, pudiendo llevarlos a cabo con gran eficiencia y precisión. Cada vez son más y más las investigaciones que se realizan con ellos y los estudios que ayudan a mejorarlos.

El presente documento refleja el procedimiento llevado a cabo para la creación de un mundo virtual donde se moverá la simulación de un drone con una carga suspendida, con el fin de utilizar el mismo para realizar estudios de su comportamiento, permitiendo así la implementación de controladores que estabilicen la carga e impidan su balanceo.

El formato adquirido por este escrito se asimilará a una guía para el usuario, con el objetivo de facilitar la compresión del trabajo realizado al retomarlo para futuras líneas de investigación.

Abstract

Quadrotors have revolutionized the world of unmanned aerial vehicles since its inception. They have transformed tasks that could pose some risk to people in simple activitites, being able to carry them out with great efficiency and precision. More and more researchs and studies are done with them in order to improve them.

The present document shows the procedure carried out for the creation of a virtual world where the simulation of a drone with a suspended load will be moved, in order to use it to conduct studies of its behaviour and also allowing the implementation of controllers that stabilize the load and prevent it from swinging.

The format acquired by this essay is similar to a guide for the user, with the aim of facilitating the understanding of the work done when resuming it for future lines of research.

Índice Abreviado

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice Abreviado **xv**

Índice **xvii**

Notación **xix**

1 Introducción **1**

2 Software **3**
2.1. Erle-Copter 3
2.2. Erle-Brain 4
2.3. Ardupilot 4
2.4. Software In The Loop (SITL) simulator 5
2.5. Robot Operating System (ROS) 5
2.6. Gazebo 6
2.7. MAVROS 7

3 Preparación del Entorno **9**
3.1. Proceso de instalación 9
3.2. Características del entorno montado 10
3.3. Primera prueba 11

4 Simulación del Optical Flow **15**
4.1. Optical Flow y LIDAR 15
4.2. Primer método 15
4.3. Segundo método 22

5 Creación del Mundo de Trabajo **29**
5.1. Archivos de una simulación 29
5.2. Plataforma 30
5.3. Carga 34
5.4. Modificaciones necesarias en el resto del código 38
5.5. Resultados y conclusiones 39

6 Comunicación **41**
6.1. Configuración inicial 41
6.2. Comunicación por terminal 42
6.3. Comunicación usando QGroundControl 43

7 Interfaz de Control Propia **47**
7.1. Componentes del programa *47*
7.2. Interfaz gráfica *49*

8 Control **53**
8.1. Configuración del código del programa *53*
8.2. Modo LOITER *55*
8.3. Detección de la carga *61*
8.4. PID para control de carga *66*

9 Conclusión **67**

Índice de Tablas **69**

Índice de Figuras **71**

Índice de Comandos **73**

Índice de Códigos **75**

Bibliografía **77**

Índice

Agradecimientos **ix**
Resumen **xi**
Abstract **xiii**
Índice Abreviado **xv**
Índice **xvii**
Notación **xix**
1 Introducción **1**
2 Software **3**
2.1. Erle-Copter *3*
2.2. Erle-Brain *4*
2.3. Ardupilot *4*
2.4. Software In The Loop (SITL) simulator *5*
2.5. Robot Operating System (ROS) *5*
2.5.1 Esquema de funcionamiento 5
2.6. Gazebo *6*
2.7. MAVROS *7*
2.7.1 MAVLink 7
3 Preparación del Entorno **9**
3.1. Proceso de instalación *9*
3.2. Características del entorno montado *10*
3.3. Primera prueba *11*
4 Simulación del Optical Flow **15**
4.1. Optical Flow y LIDAR *15*
4.2. Primer método *15*
4.2.1 PX4 de Pixhawk 16
4.2.2 XML 16
4.2.3 Formato SDF (Simulation Description Format) 17
4.2.4 Formato URDF (Universal Robotic Description Format) 19
4.2.5 Conclusiones del primer método 21
4.3. Segundo método *22*
4.3.1 Archivo *lidar_sensor.urdf.xacro* 22
4.3.2 Archivo *generic_camera.urdf.xacro* 24
4.3.3 Archivo *erlecopter.xacro* 25

4.3.4 Resultado y conclusiones del segundo método 26

5 Creación del Mundo de Trabajo 29

5.1. Archivos de una simulación 29

5.1.1 Archivos *.world* 29

5.1.2 Archivos *.urdf, .xacro* y *.sdf* 29

5.1.3 Archivos *.launch* 29

5.2. Plataforma 30

5.2.1 Modificación de las colisiones del drone 32

5.3. Carga 34

5.3.1 Primer bloque: modelado físico e inercial de la carga 34

5.3.2 Segundo bloque: modelado del comportamiento dinámico de la carga 36

5.4. Modificaciones necesarias en el resto del código 38

5.4.1 Modificacion del archivo *empty.world* 38

5.4.2 Modificacion del archivo *erlecopter.xacro* 38

5.4.3 Modificación del archivo *erlecopter_spawn.launch* 39

5.5. Resultados y conclusiones 39

6 Comunicación 41

6.1. Configuración inicial 41

6.2. Comunicación por terminal 42

6.3. Comunicación usando QGroundControl 43

7 Interfaz de Control Propia 47

7.1. Componentes del programa 47

7.2. Interfaz gráfica 49

8 Control 53

8.1. Configuración del código del programa 53

8.2. Modo LOITER 55

8.2.1 Estudio sobre el Pitch y el Roll 56

8.2.2 Conclusiones de los experimentos 57

8.3. Detección de la carga 61

8.4. PID para control de carga 66

9 Conclusión 67

Índice de Tablas 69

Índice de Figuras 71

Índice de Comandos 73

Índice de Códigos 75

Bibliografía 77

Notación

Kd	Amortiguamiento
Joint	Articulación
cm	Centímetros
τ	Constante de tiempo
Cfm	Constraint force mixing
ρ_{agua}	Densidad del agua
Link	Enlace
Erp	Error reduction parameter
XML	Extensible Markup Language
FCU	Flight Control Unit
FDM	Flight Dynamics Model
K	Ganancia del Sistema
K_d	Ganancia derivativa
K_i	Ganancia integral
K_P	Ganancia proporcional
g, g	Gramos
GUI	Graphical User Interface
GCS	Ground Control Station
HTML	HyperText Markup Language
ΔY	Incremento de la salida
ΔU	Incremento de la señal de control
Kg	Kilogramos
LIDAR	Light Imaging Detection and Ranging
L	Longitud del hilo
m_{agua}	Masa del agua
m	Metro
MAVLink	Micro Air Vehicle Communication Protocol
I	Momento de inercia
I_{CM}	Momento de inercia en el centro de masas
π	PI

R_{esf}	Radio de la esfera
RTL	Return To Load
kp	Rigidez de contacto
ROS	Robot Operating System
SDF	Simulation Description Format
SLAM	Simultaneous Location and Mapping
SITL	Sofware In The Loop
SGML	Standard Generalized Markup Language
$ts_{90\%}$	Tiempo de subida (al 90%)
T_d	Tiempo derivative
T_i	TIempo integral
URDF	Universal Robotic Description Format
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
V_{esf}	Volumen de la esfera

1 INTRODUCCIÓN

El mundo relacionado con los vehículos aéreos no tripulados ha sufrido una expansión espectacular en los últimos años, aunque sus comienzos se remontan incluso a los años 20. Este crecimiento no se ha producido únicamente en un mismo ámbito, sino que el abanico de actividades en los que se demandan actualmente los UAVs (Unmanned Aerial Vehicle) es inmenso. Aunque sus primeros usos fueron principalmente aplicaciones militares, posteriormente se comenzaron a utilizar en el entorno civil, comercial y cinematográfico. Dicho auge generó a su vez la búsqueda de componentes más optimizados, como baterías que ofreciesen una mayor autonomía, materiales más ligeros, a la vez que resistentes y la inclusión de sistemas de sensorización o de herramientas de manipulación, como brazos robóticos.

Dentro de la definición de UAV se pueden encontrar varias familias, pero el presente documento está enfocado a los drones de tipo quadrotor, que son uno de los vehículos aéreos no tripulados más novedosos en la actualidad. Constituyen una herramienta perfecta para ámbitos de vigilancia, detección e identificación de personas, objetos o incendios, gracias a su facilidad para incorporar cámaras y sensores de posicionamiento, además de su capacidad para mantenerse estables en pleno vuelo.

Figura 1.1 Drone con cámara incorporada

Otra de sus grandes ventajas es que pueden llegar a zonas de difícil acceso para las personas, de modo que no es necesario correr riesgos como los que se tienen en torres eléctricas, partes inferiores de puentes, zonas exteriores de edificios altos o zonas en las que ha ocurrido alguna catástrofe natural.

Figura 1.2 Drone destinado a salvamento

Además de estas aplicaciones, una de las más interesantes es el transporte de cargas y mercancías. Hoy en día ya existen empresas que realizan envíos de paquetes utilizando este sistema, como Amazon, cuyo UAV se

muestra en la Figura 1.3. Aunque, además de esta aplicación, más adelante se podría utilizar esta habilidad para el rescate de personas del mar o de zonas a las que solo se puede acceder por aire.

Figura 1.3 Drone de la empresa Amazon para reparto de paquetes

Frente a estos objetivos en los que existe una carga suspendida, el problema principal es el balanceo de la misma. Para su solución es necesaria la implementación de un control adecuado mediante el cual el drone consiga mantener estable la misma.

Esta última aplicación es la que ha inspirado el presente trabajo, que se incluye dentro de un grupo de proyectos de fin de carrera y de master, todos bajo la supervisión del Departamento de Sistemas y Automática de la Escuela Técnica Superior de Ingeniería y llevados a cabo por diferentes alumnos, que implican un estudio del modelado de un sistema compuesto por un drone más una carga suspendida, la simulación del mismo haciendo uso de la herramienta de simulación Gazebo y su implementación física utilizando para ello el drone que recibe el nombre de Erle-Copter, perteneciente a la empresa Erle Robotics, y el cual ha sido adquirido por el departamento antes mencionado.

Concretamente, este proyecto se centra en la segunda de las tareas nombradas. Para esto se utilizará el modelo del Erle-Copter en Gazebo proporcionado por Erle Robotics y se adaptará el mismo de manera que incluya la carga suspendida, así como todos los sensores de los que dispone el drone físico. Dicha simulación se implementará en un ordenador aportado por el departamento, y se tratará al mismo de la misma manera a como se trataría el sistema físico, por lo que se establecerá la conexión entre dicho ordenador y un portátil, desde el cuál se podrá controlar el UAV.

Una vez montado todo el entorno de simulación, se probará un modo de vuelo del drone y se estudiará su comportamiento con el fin de implementar un controlador para la carga, utilizando para ello un programa en C que permite el manejo del vehículo de manera remota y la implementación de los controladores pertinentes con los que mantener estable la carga suspendida.

Se conseguirá así un sistema de simulación y pruebas centrado en las actividades que se han mencionado previamente que permitirá futuros estudios en diferentes proyectos de investigación, así como la continuación de este trabajo como parte de trabajos de fin de grado o de fin de máster realizados por nuevos alumnos, quienes además podrán utilizar el presente documento como guía y punto de información detallada en lo referente al proyecto tratado.

2 SOFTWARE

Han sido numerosos los componentes software necesarios para la realización de este proyecto, es por esto por lo que se destinará el presente capítulo a realizar una breve presentación de los mismos, detallando sus características más importantes, así como el papel que han jugado en el trabajo.

Es necesario resaltar que toda la información incluida en los siguientes apartados se encuentra recogida en las páginas web oficiales tanto de Erle Robotics como en las de cada una de las herramientas en especifico.

2.1. Erle-Copter

Este es el nombre que recibe el quadrotor utilizado y constituye, por ello, el pilar fundamental de este proyecto. Aunque el hardware del mismo no estará presente en la elaboración del trabajo, es importante presentarlo ya que en la simulación llevada a cabo el drone que aparece es una réplica fiel de dicho UAV.

Figura 2.1 Erle-Copter

Se trata del primer drone inteligente basado en el Sistema Operativo Linux, lo que supone una gran novedad dentro de su campo. Utiliza marcos robóticos como ROS (Robot Operating System) y está dirigido por el cerebro artifical Erle-Brain 2, que será descrito posteriormente.

Se incluyen en la siguiente tabla algunas de sus características principales, ya que estas se verán reflejadas en el modelo virtual del dispositivo.

Tabla 2.1 Características comerciales del Erle-Copter

Característica	Descripción
Dimensiones	370 x 370 x 95 mm
Peso	878 g
Carga útil	1 kg
Hélices	10 x 4.5 o 9.4 x 4.3 (pulgadas X paso de la hélice)
Color	Negro y amarillo

El Erle-Copter se convierte así en un dispositivo ideal para operaciones en el exterior, y cuyas funcionalidades y controladores pueden ser modificados por el usuario de manera sencilla. Por ejemplo, destaca la posibilidad de instalar aplicaciones en el dron utilizando para ello simplemente un navegador. De esta manera, se puede

considerar como una herramienta perfecta con la que llevar a cabo estudios, investigaciones y experimentos diversos.

2.2. Erle-Brain

Otro de los dispositivos esenciales es este cerebro artificial implementado por la empresa Erle Robotics, ya que en el se encuentra todo el software y la sensorización necesaria para convertir el Erle-Copter en un dispositivo autónomo.

Proporciona una Unidad de Control de vuelo o FCU (Flight Control Unit), un ordenador encargado de aportar controles de vuelo básicos, también denominado autopiloto, y un ordenador complementario basado en la Raspberry PI 2. Además de esto, incluye numerosos sensores como son una IMU, un barómetro, GPS, una brújula digital, con posibilidad a su vez de añadirle cámaras. Permite conexiones por puerto I2C, UART, Etherhet y USB, aunque también es compatible con antenas Wifi. Por último, es muy importante comentar su compatibilidad con ROS, de manera que se pueden desarrollar todo tipo de aplicaciones robóticas.

Como se ha dicho anteriormente, aunque dicho dispositivo no aparecerá de manera física en este trabajo, se ha utilizado también un modelado virtual del mismo en la simulación.

Figura 2.2 Erle Brain 2

2.3. Ardupilot

Ardupilot es un controlador o autopiloto de código abierto que resulta de gran ayuda en la construcción de vehículos autónomos de todo tipo. En especial, la versión utilizada en este caso es el ArduCopter, ya que es específica para el uso en drones.

Figura 2.3 Logotipo de Ardupilot

Mediante este software se es capaz de realizar un control del quadrotor de manera manual o automática. Incluye, a su vez, numerosos modos de vuelo (Acro, Stabilize, Loiter, Guided, Return-to-Launch, Alt Holt, etc) con los que se pueden llevar a cabo diversas trayectorias y experimentos con el drone. Además, es un software totalmente flexible y personalizable, de manera que, aunque la gran mayoría de los parámetros y configuraciones están ya preprogramadas, estos se pueden modificar y ampliar si el usuario lo cree conveniente.

Otra de sus ventajas es que es compatible con la gran mayoría de programas de estación de tierra, como pueden ser el APM Planner, el Mission Planner o el QGroundControl, que consisten en una interfaz gráfica de gran utilidad para la ejecución de misiones autónomas complejas, a la vez que permiten la visualización de los diferentes valores de los sensores, parámetros y configuraciones del drone.

Para su simulación, resulta de gran ayuda el uso del simulador SITL, que se expondrá a continuación.

2.4. Software In The Loop (SITL) simulator

Este simulador permite probar el comportamiento del quadrotot sin necesidad de utilizar el sistema físico correspondiente. Se trata de una construcción del código del autopiloto en C++ que proporciona un ejecutable con el cual es posible probar el comportamiento del código sin hardware.

En definitiva, mediante SITL se puede ejecutar ArduPilot directamente en un ordenador. De este modo, se saca provecho de la gran portabilidad del autopiloto ArduPilot, que puede implementarse en diferentes plataformas, siendo el ordenador una de ellas.

Cuando se ejecuta SITL, los datos del sensor provienen de un modelo de dinámica de vuelo en un simulador de vuelo. ArduPilot tiene una amplia gama de simuladores de vehículos incorporados, y puede interactuar con varios simuladores externos. Además, es fácil añadir nuevos tipos de sensores y vehículos simulados.

Otra de las ventajas de disponer de ArduPilot en SITL es que este le da acceso a una gran variedad de herramientas de desarrollo como depuradores interactivos, analizadores estáticos y herramientas de análisis dinámico, por lo que desarrollar y probar nuevas características en ArduPilot es mucho más sencillo.

2.5. Robot Operating System (ROS)

ROS consiste en un framework flexible para escribir aplicaciones software destinadas a ser implementadas sobre sistemas robotizados. En concreto, se trata de una colección de herramientas y librerias cuyo principal objetivo es simplificar la tarea de crear un comportamiento robótico complejo y robusto a través de una amplia variedad de plataformas robóticas. En definitiva, provee los servicios estándar de un sistema operativo.

Figura 2.4 Logotipo de ROS

Se creó como método de desarrollo de software de robótica colaborativa, de modo que diferentes grupos de personas pueden compartir entre ellos las diferentes herramientas que cada uno de ellos desarrolle.

Es así por lo que ROS posee dos partes básicas, una de sistema operativo, ROS, y otra que es un conjunto de paquetes aportados por la contribución de usuarios (*stacks*) que implementan funcionalidades diversas como SLAM (Simultaneous Location and Mapping), planificación, percepción, simulación, etc.

2.5.1 Esquema de funcionamiento

ROS está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos, los cuales pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores.

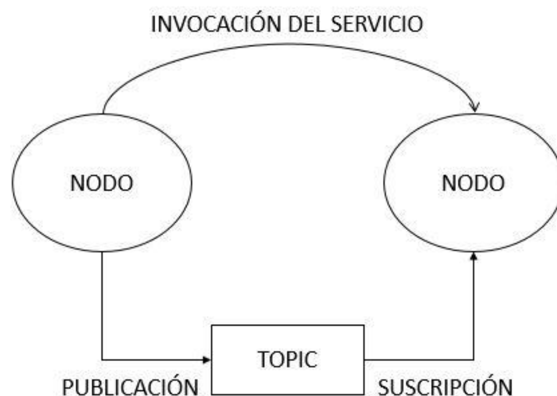

Figura 2.5 Esquema simplificado del funcionamiento de ROS

Los conceptos más importantes de este software son los siguientes:

- Nodos: son los procesos que llevan a cabo todos los cálculos. Así es como se consigue que el diseño de ROS posea un gran modularidad, ya que, por ejemplo, un nodo se ocuparía de la visión, otro nodo del movimiento de los rotores, otro nodo de interpretar los valores de los sensores etc.
- Master: el ROS Master proporciona registros de nombres y búsqueda para que los nodos puedan encontrarse entre ellos, intercambiar mensajes o invocar servicios.
- Mensajes: es el método por el que se comunican los nodos. Son simplemente estructuras de datos. Pueden incluir estructuras complejas como fragmentos de código C.
- Topics: los mensajes se enrutan a través de un sistema de publicación/suscripción. Un nodo publicará un mensaje en un topic determinado. Dicho topic es un nombre que se utiliza para identificar el contenido del mensaje. Un nodo que esté interesado en los datos que son recogidos por estos topics se suscribirá a aquel que le corresponda.

  Pueden existir varios editores y suscriptores simultáneos para un único topic, aunque ninguno de ellos es consciente de la presencia de los demás.
- Servicios: la petición/respuesta se lleva a cabo a través de servicios, definidos por un par de estructuras de mensajes, una para solicitudes y otra para respuestas. De este modo, un nodo ofrece un servicio bajo un determinado nombre y un cliente utilizará dicho servicio enviando el mensaje de solicitud y esperando la respuesta.

Esta modularidad permite que ROS se convierta en un sistema muy ampliable.

2.6. Gazebo

Gazebo es la principal herramienta de simulación utilizada en este trabajo. Se trata de un simulador dinámico en 3D con la habilidad de crear virtualmente sistemas robóticos de manera eficiente y precisa en entornos complejos de interior o exterior.

Mediante la definición de articulaciones (*joints*), vínculos (*links*) con colisiones (*collisions*), etcétera, esta plataforma ofrece la posibilidad de simular tanto la cinemática como la dinámica de los robots o sistemas no robotizados que se implementen, con un alto grado de fidelidad, además de numerosos sensores e interfaces con el usuario.

Tal y como se comentará en el capítulo 5, con este programa se puede trabajar desde su mismo editor de modelos (Model Editor), utilizando para ello la interfaz gráfica del simulador, o, por otro lado, se puede optar por modificar directamente los distintos códigos en los que se divide una simulación completa para realizar los modelos de la simulación.

Mediante el uso de esta plataforma, se conseguirá una simulación bastante fiel del UAV físico, por lo que, en definitiva, su comportamiento en el mundo virtual debería ser un claro reflejo de cómo actuará el mismo en el mundo real.

Figura 2.6 Logotipo de Gazebo

2.7. MAVROS

MAVROS es un paquete de ROS compatible con varios autopilotos y que proporciona los drivers necesarios que permiten utilizar el protocolo de comunicaciones llamado MAVLink (Micro Air Vehicle Communication Protocol). Adicionalmente proporciona un puente UDP MAVLink para estaciones de control terrestre como puede ser QGroundControl.

2.7.1 MAVLink

Como se ha comentado, MAVLink es un protocolo de comunicaciones capaz de empaquetar programas en lenguaje C sobre canales seriales con alta eficiencia, y enviar dichos paquetes a la estación de control terrestre. Es compatible con numerosos autopilotos y soporta además varios tipos de datos.

Es un protocolo orientado hacia dos propiedades prioritarias, que son la velocidad de transmisión y la seguridad. Con el es posible comprobar el contenido de un mensaje y detectar paquetes perdidos.

3 PREPARACIÓN DEL ENTORNO

La base de este proyecto es la simulación del Erle-Copter junto con su dinámica, su sensorización y el software correspondiente. Erle Robotics se ha preocupado de implementar y ofrecer de manera pública todos los archivos necesarios para esto, de manera que todo aquel que lo desee pueda llegar a probar el funcionamiento del dron y llevar a cabo diferentes experimentos con el. Adicionalmente, una de las ventajas de disponer de este entorno es que posibilita el hecho de poder probar el UAV sin correr el riesgo de que al estrellarse se dañe alguno de sus componentes, riesgo que se corre si se prueba el mismo directamente en la realidad.

Es por esto por lo que se dedicará un capítulo a explicar los distintos elementos que componen el entorno. Al seguir los pasos de instalación de la página de Erle Robotics, es probable que surjan algunos errores en el proceso de instalación, de modo que se presentarán las soluciones a los mismos junto con la dirección donde se puede encontrar.

3.1. Proceso de instalación

El Sistema operativo utilizado es Ubuntu 14.04. Es importante no actualizar a Ubuntu 16 ya que los resultados de este proyecto no han sido probados en dicha plataforma.

El proceso a seguir para la instalación del entorno de trabajo se encuentra definido paso a paso en la página de Erle Robotics, concretamente en el apartado de *Support-Docs-Simulation-Configuring your environment*. Los encargados de la misma se preocupan por mantener actualizadas dichas instrucciones a seguir, pero, en el presente proyecto, aparecieron algunos errores durante este proceso, los cuales se indican a continuación. De todos modos, lo ideal es intentar instalar todo el software siguiendo los pasos de la anterior referencia, por si en el momento en que el lector lo haga ya han solucionado dichos errores.

En el momento de instalar ROS Indigo, tras instalar las dependencias del ROS Base, es decir, tras ejecutar los comandos 3.1, se debe ejecutar el comando 3.2.

```
sudo apt-get    install python-rosinstall          \
                ros-indigo-octomap-msgs    \
                ros-indigo-joy             \
                ros-indigo-geodesy         \
                ros-indigo-octomap-ros     \
                ros-indigo-mavlink         \
                ros-indigo-control-toolbox \
                unzip
```

Comando 3.1 Comandos para la instalación de las dependencias del ROS Base sacado de [3]

```
sudo apt-get install ros-indigo-ros-control ros-indigo-ros-controllers -y
```

Comando 3.2 Comando para solucionar error 1

Con esta línea se elimina un error posterior relacionado con los mensajes de comunicación entre ROS y Gazebo, el cual dice así:

Could not find a package configuration file provided by “transmission_interface”

Esta solución se encontró en [11].

En la instalación de dichos mensajes, la última línea dicta como se muestra en el comando 3.3.

```
…
catkin_make -j 4
```

Comando 3.3 Último comando de la instalación de mensajes de comunicación entre ROS y Gazebo

Si al utilizar este comando, falla la compilación, se debe probar a cambiar el 4 por un 1[1].

Estos son los errores que surgieron. Una vez completada la instalación, se tendrá montado en el ordenador un entorno como el que se describe en el siguiente apartado.

3.2. Características del entorno montado

En la figura 3.1 se muestra un esquema de todo el entorno obtenido tras realizar los pasos anteriores.

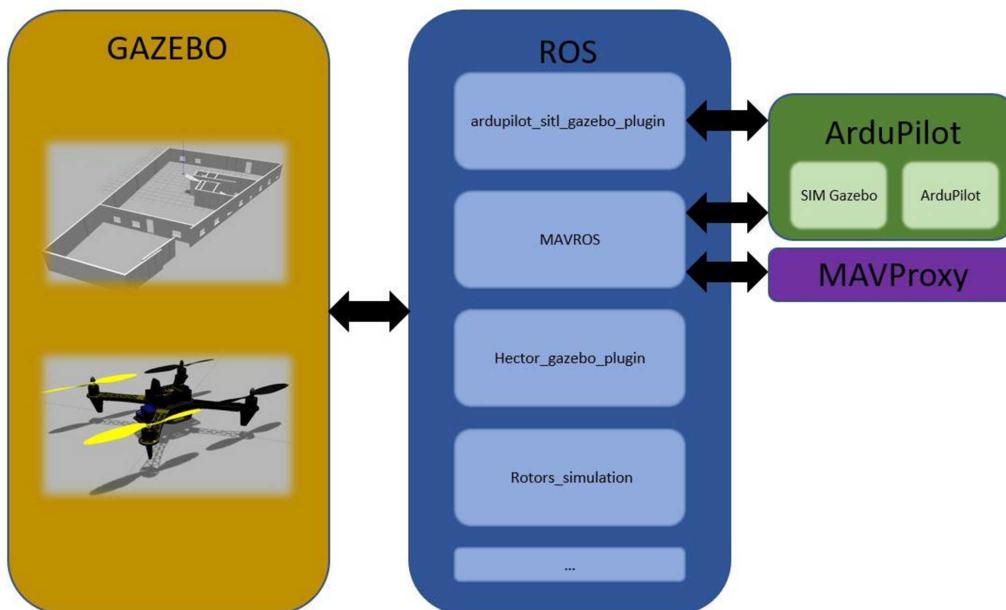

Figura 3.1 Esquema del entorno instalado

La versión instalada de Gazebo es compatible con la versión ROS indigo, que se corresponde con la utilizada en el drone real. También incluye MAVROS, lo cual permite la interacción con la versión virtual tal y como se haría con la versión real. Como se explicó en el apartado 2.7, para efectuar dicha comunicación se utiliza el protocolo MAVLink, el cual, además, permite acceder a todos los parámetros del drone virtual y del drone real.

Además de esto, el esquema del drone virtual será algo parecido a lo mostrado en la figura 3.2. En ella se puede apreciar como se hace uso de la simulación del autopiloto gracias al simulador SITL. Además, el autopiloto se comunica con un simulador de modelos dinámicos de vuelo o FDM (Flight Dynamics Model) llamado JSBSim, de código abierto, el cual, a su vez, se comunica con Gazebo, de modo que se consigue obtener un modelo dinámico muy fiel a la realidad.

El autopiloto también recibe todas las señales necesarias a través del protocolo MAVLink provenientes del MAVProxy, que se trata de una paquetería de estación de control en tierra (GCS) para UAVs basados en sistemas MAVLink. Es importante mencionar que además de con MAVProxy se puede conectar el autopiloto a otras estaciones de tierra utilizando de nuevo el mismo protocolo.

[1] Este valor representa el número de hilos utilizado para la compilación. Es posible que el ordenador utilizado carezca de un microprocesador con los recursos suficientes para utilizar 4 hilos.

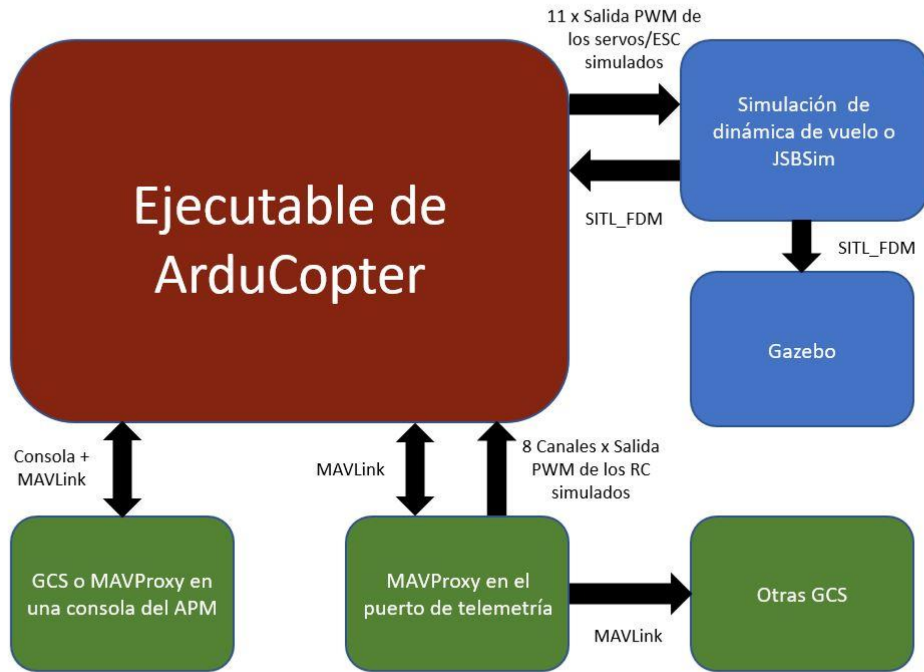

Figura 3.2 Esquema del drone simulado

3.3. Primera prueba

Tras el montaje completo del mundo, cabe la posibilidad de probar si todo se ha instalado de manera correcta. Los pasos que hay seguir para este testeo, aunque se exponen a continuación, también se pueden encontrar en la página oficial de Erle Robotics, en el apartado *Support/Docs/Simulation/Vehicles/Erle-Copter/Launching Erle-Copter simulation*. Este se trata del primero de cinco tutoriales que conviene realizar, para tener una primera toma de contacto con la simulación, aunque para el presente proyecto solo se probó el primero con el fin, como se ha explicado previamente, de comprobar que todo funciona según lo esperado.

Esta primera prueba trata simplemente de conseguir enviarle al drone una serie de comandos por terminal, haciendo uso del MAVProxy, y que este ascienda 2 metros, se estabilice, y finalmente aterrice.

Para lanzar la simulación se requieren dos terminales distintos, uno en el que se ejecutará MAVProxy, y otro desde el cual se lanza ROS, la simulación de ardupilot sobre SITL y Gazebo con el mundo del Erle-Copter incluido. Los pasos a seguir son los siguientes:

- En un terminal, lanzar MAVProxy de la siguiente manera:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
cd ~/simulation/ardupilot/ArduCopter
../Tools/autotest/sim_vehicle.sh -j 4 -f Gazebo
```

Comando 3.4 Comandos para lanzar MAVProxy

- En un segundo terminal se arranca la simulación:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
roslaunch ardupilot_sitl_gazebo_plugin erlecopter_spawn.launch
```

Comando 3.5 Comandos para lanzar la simulación

- Tras hacer lo anterior, se debe esperar a que el mundo cargue por completo, lo cual se sabrá porque, tras inicializarse todos los parámetros y sensores del Erle-Copter, en el terminal de MAVProxy aparecerá un mensaje de *Flight battery 100 percent*. Tras recibir este mensaje, se deben cargar los parámetros del Erle-Copter. Para ello, en el primer terminal se debe ejecutar lo siguiente:
-

```
param load /[ruta_al_directorio_personal]/simulation/ardupilot/Tools/Frame_params/Erle-Copter.param
# Ejemplo: param load /home/pablo/simulation/ardupilot/Tools/Frame_params/Erle-Copter.param
```

Comando 3.6 Comandos para cargar los parámetros de Erle-Copter

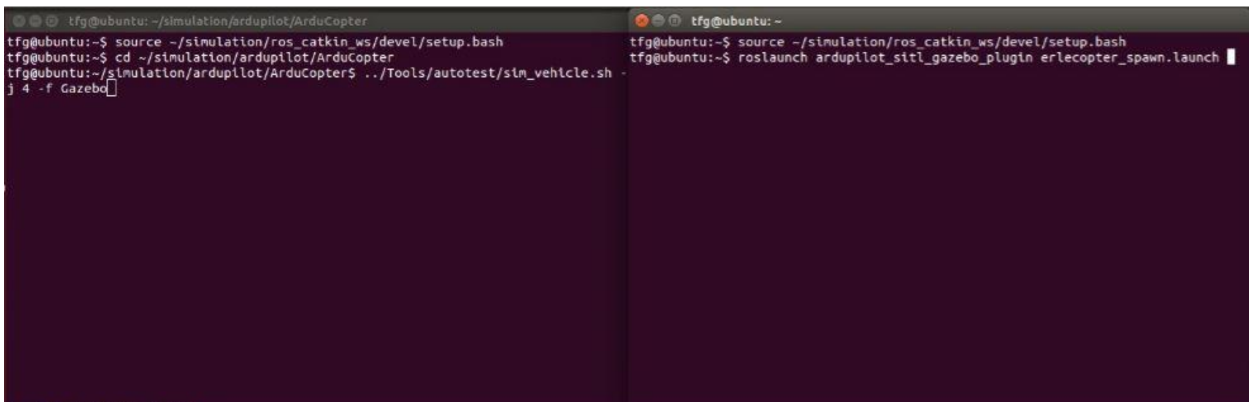

Figura 3.3 Terminales con los comandos para lanzar la simulación

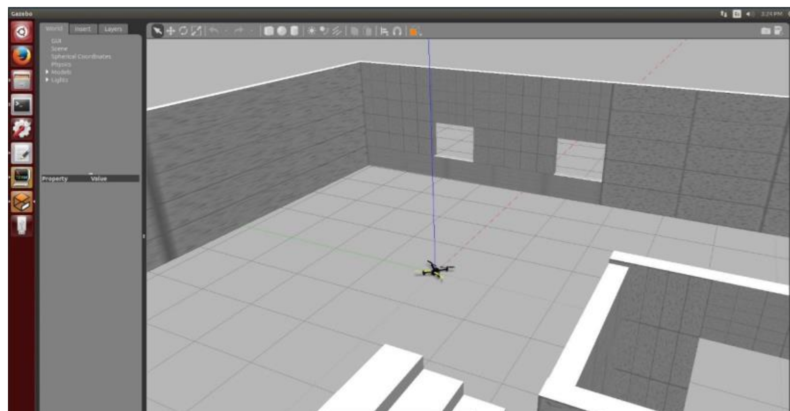

Figura 3.4 Inicio de la simulación

Una vez hecho esto sin que se haya producido ningún problema significará que ya está cargada la simulación por completo. En este momento se puede probar a levantar el drone haciendo uso de los diferentes modos que provee el autopiloto. Se muestran a continuación la forma de hacerlo volar mediante los modos *GUIDED* y

LOITER. Para ello, en el terminal del MAVProxy se escribe lo que aparece en la tabla de comandos 3.4.

```
mode GUIDED
arm throttle
takeoff 2
```

Comando 3.7 Comandos para armar y despegar el drone en modo GUIDED

O bien, utilizando el modo *LOITER*, con el cual se sobreescriben los distintos canales de *Roll*, *Pitch*, *Yaw* o *Throttle*.

```
mode LOITER
arm throttle
rc 3 1600
rc 3 1500
```

Comando 3.8 Comandos para armar y despegar el drone en modo LOITER

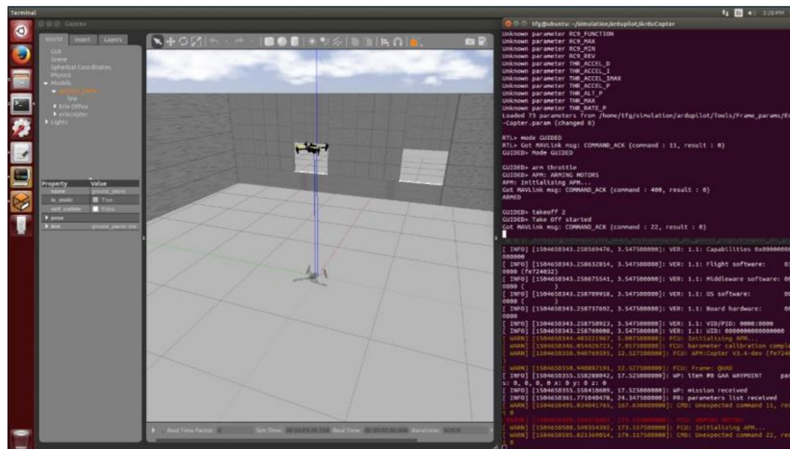

Figura 3.5 Erle-Copter volando

Si se produce algún fallo al armar los motores, se debe probar a poner a cero el parámetro *ARMING_CHECK* desde el terminal del MAVProxy (el primero de los dos), tal y como se indica en la tabla de comandos 3.9.

```
param set ARMING_CHECK 0
```

Comando 3.9 Comandos para establecer el parámetro *ARMING_CHECK* a cero

Una vez hecho esto, se debe reiniciar la simulación para que se guarde dicha actualización. Una vez reiniciada, probar de nuevo los comandos 3.7 o 3.8.

Para aterrizar el drone simplemente habría que cambiar el modo a *LAND*.

Además de lo anterior, también puede resultar interesante el mostrar las imágenes de las cámaras que lleva incorporadas el drone. Esto se conseguiría como se muestra en la tabla de comandos 3.10.

```
rosrun image_view image_view image:=/erlecopter/front/image_front_raw
rosrun image_view image_view image:=/erlecopter/bottom/image_raw
```

Comando 3.10 Comandos para visualizar las cámaras del Erle-Copter

Con estos últimos comandos, lo que se consigue en definitiva es recibir los datos de las cámaras que se están publicando en los topics de nombres *image_front_raw* y *image_raw*.

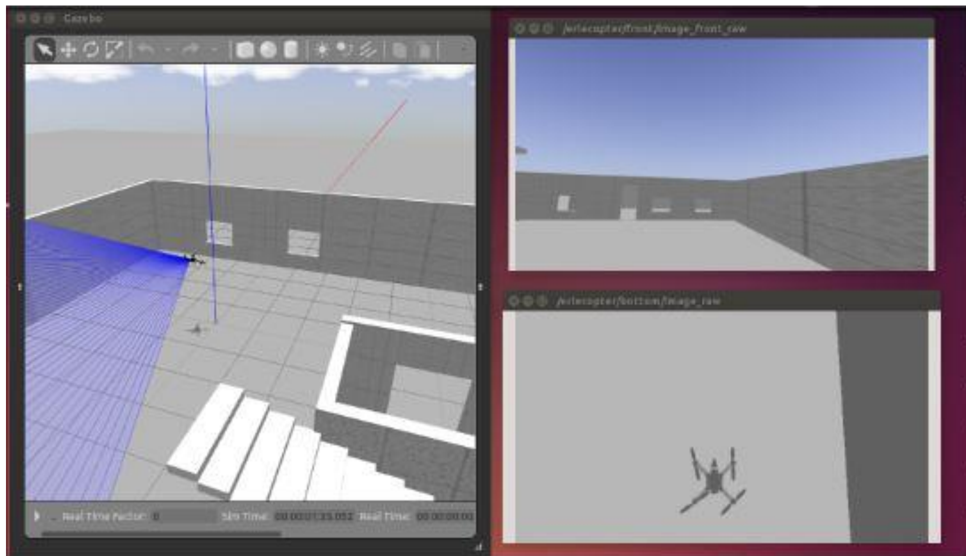

Figura 3.6 Simulación de Erle-Copter con cámaras activadas

4 SIMULACIÓN DEL OPTICAL FLOW

Una vez que se dispone de todo el entorno de simulación inicial, lo primero que hay que tener en cuenta es que se necesita una simulación de un sensor de flujo óptico similar al que se encuentra instalado en el drone físico, ya que con la ayuda de este se podrán conocer valores de posicionamiento y velocidad del Erle-Copter, tal y como se explicará posteriormente.

Es por esto por lo que se dedica este capítulo a detallar los dos procedimientos con los que se ha abordado este problema, siendo el segundo de los expuestos el que se ha optado por utilizar.

4.1. Optical Flow y LIDAR

Un optical flow o sensor de flujo óptico es un sensor que tiene la capacidad de medir el movimiento a través de una referencia visual y emitir una medición a partir de la misma. Existen varias configuraciones de este tipo de sensores, pero el que se ha utilizado tanto en el Erle-Copter físico como en el de la simulación se trata de un sensor de flujo óptico acompañado de un giróscopo de tres ejes y de un LIDAR o sonar, con el cual se complementan las medidas tomadas por el optical flow, disminuyendo el error de la medida total, y además se obtienen mediciones de altura.

Su funcionamiento se basa en el uso de la textura del suelo y otras características visibles para determinar la velocidad del vehículo, entre otras medidas. Además, dicho sensor será utilizado en el Erle-Copter para posicionar el drone junto con las medidas del GPS.

Un LIDAR tiene la misma finalidad que un sonar. Se trata de un dispositivo que permite determinar la distancia entre un emisor y una superficie, utilizando para ello un haz láser pulsado. Esta distancia es determinada mediante la medición del tiempo de retraso entre la emisión del pulso y su detección tras ser reflejada. Aún así, es necesario determinar la posición instantánea de dicho sensor, para lo cual se utiliza el GPS. Conocidos estos datos y la distancia del sensor a la superficie, se consigue obtener las coordenadas específicas del drone.

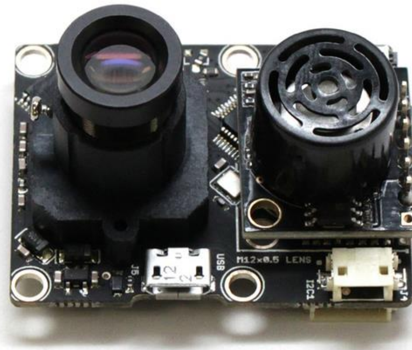

Figura 4.1 Optical Flow de Pixhawk

4.2. Primer método

El sensor de flujo óptico implantado en el drone físico se trata del optical flow de la empresa *Pixhawk*, otra de las grandes compañías destinadas a la fabricación de, en su mayor parte, autopilotos.

Es por esto por lo que el código que aparece implementado en Gazebo de su optical flow requiere de un proceso de adaptación para poder incluirlo en el modelo de Erle-Copter. De esto se trata el primer método presentado. En el, se intentará desacoplar el optical flow de PX4 implementado en el modelo del drone que recibe el nombre de Iris, para posteriormente incluir el mismo en el modelo de Erle-Copter.

4.2.1 PX4 de Pixhawk

El PX4 de Pixhawk se trata de otro de los grandes autopilotos que existen en la actualidad. Funciona practicamente de la misma manera que el ArduPilot y posee características similares como el ser un software de código abierto, el ofrecer distintos modos de vuelo para las aeronaves o su compatibilidad con diversas GCS. Además, también es compatible con multitud de plataformas como ROS, Gazebo o con el protocolo MAVLink, al igual que el ArduPilot.

Figura 4.2 Autopiloto PX4 de Pixhawk

En el ámbito de la simulación se comporta de la misma manera que el primer autopiloto presentado, ya que también se puede utilizar SITL con el fin de probar su funcionamiento sin requerimiento de hardware ninguno.

En especial, el modelo simulado del optical flow de esta misma marca es el que interesa para el proyecto. Como se ha comentado anteriormente, es necesario el desacople del mismo del modelo del drone que ofrece PX4, llamado Iris y, una vez separado de este, se necesita incluir dicho modelo del sensor al que ya se tiene montado del Erle-Copter.

Figura 4.3 Drone Iris

Frente a este objetivo, lo principal es conseguir transformar el modelo del sensor del formato SDF (Simulation Description Format) al formato URDF (Universal Robotic Description Format), los cuales se expondrán a continuación.

4.2.2 XML

Los dos formatos utilizados para definir los modelos de todos los robots que se deseen simular, que son SDF y URDF, se basan en el formato XML. Este es un meta-lenguaje que permite definir lenguajes de marcado (pero no es uno de ellos) adecuados a usos determinados, de manera que se pueden almacenar datos de forma legible.

Se trata de un subconjunto de SGML (Standard Generalized Markup Language), simplificado y adapatado a Internet. Es importante no confundirlo con HTML (HyperText Markup Language), el cual es un lenguaje definido por SGML, por lo tanto, una aplicación de dicho lenguaje.

Con su uso se consigue estructurar documentos de modo que la información que contienen pueda ser almacenada, transmitida o procesada por numerosas aplicaciones y dispositivos.

Posee numerosas ventajas, entre ellas su extensibilidad mediante la adición de nuevas etiquetas, su compatibilidad entre aplicaciones de distintas plataformas y su fácil comprensión debido a su organización y a la capacidad de transformar datos en información.

4.2.3 Formato SDF (Simulation Description Format)

SDF es, como se ha comentado, un formato de XML con el cual se describen objetos y entornos para simulaciones de robots, así como su visualización y control. Todos los archivos manejados por Gazebo tienen este formato. Haciendo uso del mismo se consigue describir todos los aspectos de un sistema robotizado, desde robots simples basados en un chasis con ruedas a robots humanoides.

Los documentos escritos en este formato se estructuran de la siguiente manera:

- Un archivo SDF puede incluir un elemento mundo (*<world>*) en el que se definen todos los elementos que aparecerán en el, un modelo (*<model>*) con el que se define un robot completo o cualquier otro objeto físico, o un elemento luz (*<light>*) con el que se describe la fuente de luz de una simulación. Es necesario indicar una versión de xml y una versión de sdf.

```
<?xml versión='1.0'?>
<sdf versión='1.6'>
  <world name='por_defecto'>
    . . .
  </world>
</sdf>
```

Código 4.1 Ejemplo de código de mundo en SDF

```
<?xml versión='1.0'?>
<sdf versión='1.6'>
  <model name='mi_modelo'>
    . . .
  </model>
</sdf>
```

Código 4.2 Ejemplo de código de modelo en SDF

```
<?xml versión='1.0'?>
<sdf versión='1.6'>
  <light name='mi_luz'>
    . . .
  </light>
```

```
</sdf>
```

Código 4.3 Ejemplo de modelo de luz en SDF

- Dentro de la definición del mundo pueden aparecer otros muchos atributos como viento (*<wind>*), inclusión de otros archivos desde una dirección (*<include>*), gravedad (*<gravity>*), atmósfera (*<atmosphere>*), modelos de robots u objetos físicos (*<model>*) y plugins (*<plugin>*) mediante los cuales se le pueden añadir códigos que aportan dinámica yc ontrol, entre otros.

```
<?xml versión='1.0'?>
<sdf versión='1.6'>
  <world name='por_defecto'>
    <include>
      . . .
    </include>
    <model name='mi_modelo_1'>
      . . .
    </model>
    <model name='mi_modelo_2'>
      . . .
    </model>
    . . .
  </world>
</sdf>
```

Código 4.4 Ejemplo de atributos dentro de un modelo de mundo en SDF

- En la descripción de un modelo, las partes más importantes, aunque existen otros muchos atributos, son la inclusión de archivos (*<include>*) , la posición en x, *y* y *z, y* la orientación en roll, pitch, yaw, del objeto respecto a un cuerpo de referencia (*<pose>*), que se puede indicar mediante su correspondiente atributo (*<frame>*), y los atributos más utilizados, ya que definen el robot por completo, que son los enlaces (*<link>*) y las articulaciones (*<joint>*).

```
<?xml versión='1.0'?>
<sdf versión='1.6'>
  <model name='mi_modelo'>
    <pose>0 0 0.5 0 0 0</pose>
    <link name='mi_enlace_1'>
      . . .
    </link>
    <joint type="revolute" name='mi_articulacion_1'>
      . . .
```

```
    </joint>
    . . .
  </model>
</sdf>
```

Código 4.5 Ejemplo de atributos dentro de un modelo en SDF

- En los links y joints se incluyen otros muchos atributos para cada uno de ellos, algunos de los cuales se verán los apartados 5.2 y 5.3. En lo referente al elemento link, se le deben indicar propiedades como su posición, su inercia, su masa, las colisiones, su visualización o si posee algún tipo de sensor, entre otras. Respecto al joint, se incluyen en su interior elementos como quienes son los links padre e hijo de dicha articulación, atributos imprescindibles en la correcta definición de un joint, la posición, los ejes de rotación para articulaciones de rotación o el eje de traslación para las articulaciones prismáticas, la existencia de sensores etcétera.

Se han explicado los componentes de este tipo de formato que más se han utilizado en el proyecto, aunque, como se ha comentado, existen muchos más además de estos, pero la explicación detenida de ellos no es objeto de este documento. De todos modos, si se desea conocer más sobre este tipo de lenguaje, se recomienda seguir la página oficial del formato SDF adjuntada en la bibliografía.

4.2.4 Formato URDF (Universal Robotic Description Format)

URDF es, al igual que SDF, un archivo con formato XML utilizado para describir robots. Este tipo de formato carece de muchas características que, por el contrario, posee SDF. Por ejemplo, URDF sólo puede especificar las propiedades cinemáticas y dinámicas de un solo robot de forma aislada, pero no se puede predefinir una posición del robot dentro de un mundo, tampoco se pueden especificar *joint loops* (articulaciones paralelas) y carece de algunas propiedades como la fricción. Tampoco permiten describir elementos que no sean robots, como luces, mapas con relieve, etc. Se trata de un formato menos flexible que SDF, ya que difiere un poco del formato corriente de XML y no tiene un mecanismo de compatibilidad con versiones anteriores.

Sin embargo, este es el lenguaje principal que utiliza ROS para la definición de robots, aunque si se desea utilizar Gazebo es necesario tener en cuenta ciertos aspectos especiales para que funcionen los programas correctamente, ya que Gazebo realiza una conversión de formatos de URDF a SDF de manera automática, y algunos de los atributos de un tipo de formato son incompatibles en el otro.

Debido a lo comentado previamente, en URDF existe un elemento llamado *<gazebo>* a través del cual se consiguen definir propiedades necesarias para la simulación en Gazebo. De este modo se pueden definir ciertos atributos que aparecen en el formato SDF, pero no en el formato URDF. Si no se incluye dicho elemento no habrá ningún problema, ya que se asignarán los valores por defecto. El elemento *<gazebo>* se puede aplicar a distintas etiquetas, que son *<robot>*, *<link>* y *<joint>*. Para cada uno de ellos se incluirán distintos atributos, los cuales se encuentran recogidos en las tablas 4.1, 4.2 y 4.3.

Tabla 4.1 Elemento *<gazebo>* para la etiqueta *<robot>*

Nombre	Tipo	Descripción
static	bool	Si es *true*, el modelo permanece inmóvil. En cualquier otro caso el modelo es simulado junto con su dinámica.

Los elementos de enlace y articulación (*<link>* y *<joint>*) funcionan de igual manera que para el formato SDF. De esta forma, en los enlaces se deben definir las colisiones, las visualizaciones de los mismos, un elemento *<gazebo>* y además un elemento *<inertial>* en el que se describe la masa y las inercias de dicho link.

Tabla 4.2 Elemento <*gazebo*> para la etiqueta <*link*>

Nombre	Tipo	Descripción
material	value	Material de un elemento visual, incluyendo textura y color
gravity	bool	Usar gravedad
dampingFactor	double	Factor de amortiguamiento
maxVel	double	Valor máximo de truncación en la corrección de la velocidad de contacto
minDepth	double	Minima profundidad permitida antes de aplicar el impulso de corrección de contacto
mu1 mu2	double	Coeficientes de frición
fdir1	string	Término que indica la dirección de mu1 en un marco de referencia local de colisiones
kp kd	double	Rigidez de contacto Kp y el amortiguamiento Kd
selfCollide	bool	Si es *true*, el link puede chocar con otros links del modelo
maxContacts	int	Número máximo de contactos permitidos entre dos entidades
laserRetro	double	Valor de intensidad devuelto por el sensor láser

Para el joint, se deben especificar los enlaces padre e hijo, el origen de la articulación respecto al padre, los ejes, si proceden, un término <*gazebo*> de manera opcional y términos de dinámica.

Tabla 4.3 Elemento <*gazebo*> para la etiqueta <*joint*>

Nombre	Tipo	Descripción
stopCfm stopErp	double	Fuerza de restricción de parada de la articulación (cfm) y parámetro de reducción de error (erp)
provideFeedback	bool	Permite que las articulaciones publiquen sus datos de fuerza-par a través de un plugin de Gazebo
implicitSpringDamper cfmDamping	bool	Si están a *true* se utilizará el *erp* y el *cfm* para simular la amortiguación.
fudgeFactor	double	Término de escalado para establecer los límites de un motor articular.

		Debe estar entre uno y cero.

Con el fin de trabajar correctamente con archivos URDF en Gazebo es conveniente saber lo siguiente:

- Es imprescindible que todos los elementos <*link*> posean un elemento <*inertial*> con valores diferentes de cero para la masa y la inercia, ya que Gazebo ignorará dichos enlaces en caso contrario al hacer la conversión.
- Cada elemento <*link*> debe poseer opcionalmente un único elemento <*gazebo*> mediante el cual se pueden convertir colores y texturas al formato de Gazebo o añadir plugins de sensores.
- Igualmente, por cada elemento <*joint*> se debe añadir de manera opcional un elemento <*gazebo*> para establecer la dinámica de amortiguación de manera adecuada o para agregar plugins de control al actuador.
- También se debe utilizar un elemento <*gazebo*> por cada elemento <*robot*>.
- Si se desea que el robot se encuentre unido al mundo se debe añadir un enlace <*link name=”world”/*>.

4.2.5 Conclusiones del primer método

Una vez comprendidos todos los factores que influyen en el modelado de un elemento en Gazebo ya se puede proceder al objetivo que persigue el presente apartado, la implementación del optical flow de PX4 sobre el drone del Erle-Copter.

En pocas palabras, en la página web de PX4 que aparece en la bibliografía se puede encontrar una guía de PX4, también incluida en las referencias de este documento. En esta guía se explica todo el procedimiento necesario para conseguir implementar una simulación en Gazebo con soporte de SITL y ROS en la cual aparece el drone Iris de PX4. Es interesante, si se desea utilizar este método de implementación, seguir las distintas instrucciones que se muestran en esta guía.

Una vez obtenido el modelo del Iris se puede realizar un análisis de los archivos que de los que se compone dicha simulación. Entre todos los modelos que se adquieren aparecerán tres mediante los cuales se implementa el sensor deseado. Estos tres archivos, de formatos SDF, se encuentran ubicados bajo los nombres de *iris_opt_flow.sdf*, *flow_cam/model.sdf* y *LIDAR/model.sdf*.

El primero de ellos se encarga de unir los otros dos modelos al frame del iris mediante joints de tipo revolute, asignándoles además una posición de referencia a cada uno de ellos.

Por otro lado, en el modelo del optical flow (*flow_cam/model.sdf*) se describen, como es de esperar, todos los parámetros que definen la simulación de dicho sensor. Esta definición se lleva a cabo con dos etiquetas, una de tipo <*link*> y otra de tipo <*sensor*>. Tal y como se describió en el subapartado 4.2.3, en el <*link*> se definen la posición inicial, la masa y la inercia en el bloque <*inertial*> y las propiedades visuales del sensor en el bloque <*visual*>, en el cual se indica que se trata de un cubo. Respecto a la etiqueta de tipo <*sensor*>, en ella se describen el tipo y las propiedades del sensor, indicando que será de tipo *camera*. Dentro de este bloque se indican especificaciones tales como si se mantendrá siempre el sensor actualizado (*Always_on*), la tasa de actualización de los datos (*Update_rate*), si se visualizará el sensor en el GUI (*Visualize*), el topic de ROS en el que se publicarán los datos recogidos por el mismo (*Topic*), así como los distintos parámetros de la cámara como campo de vista horizontal, el ancho y la altura de la imagen, las distancias mínimas y máximas que serán traducidas y el ruido gaussiano introducido. Por último, se incluye un plugin que le incluirá la dinámica al sensor bajo el nombre de *libgazebo_opticalFlow_plugin.so*.

Respecto al modelo del LIDAR (*LIDAR/model.sdf*) se describen del mismo modo el bloque <*link*> donde igualmente en <*inertial*> se definen su posición, masa e inercias y a continuación se encuentra su descripción visual como un cilindro de color negro. A continuación, aparece de nuevo la descripción del tipo de sensor del que se trata, que en este caso es un láser, y en el cual se especifican atributos como el número de rayos simulados por cada ciclo del láser (*samples*), la resolución (*resolution*), los ángulos máximo y mínimo (*min_angle* y *max_angle*), el rango de los rayos (*range*) y el ruido gaussiano incluido en el sensor. Finalmente aparecen de nuevo los plugins incluidos y se indica que se mantendrá siempre actualizado, la tasa de actualización de los datos y se activa la visualización.

Una vez analizados y comprendidos estos archivos, lo único que falta es transformar del formato SDF en el que están a URDF e incluirlo en el modelo de Erle.

No se continuará describiendo el proceso a seguir ya que no se continuó con el mismo en el proyecto más allá del análisis anterior de los archivos que componen el sensor, sino que se optó por realizar el segundo método que se expondrá a continuación.

4.3. Segundo método

Este segundo método, que ha sido el que se ha implementado finalmente, consiste en hacer uso del optical flow que trae incorporado la simulación de Erle-Copter, que, aunque no es el mismo que el utilizado en el drone físico, posee sus mismas características y sus dimensiones y peso no difieren mucho respecto a las del sensor real.

Antes de nada, aclarar que el formato utilizado por Erle-Copter para la descripción de modelos es URDF y Xacro. Este último se trata de un lenguaje macro, que no es más que un tipo de lenguaje de programación mediante el cual se pueden desarrollar o programar pequeñas aplicaciones o automatizar tareas que de otro modo llevaría mucho tiempo y serían muy laboriosas. Mediante este formato se consigue reducir el tamaño general que tendrían los diferentes archivos que componen la simulación si estos se trataran de modelos en URDF, consiguiendo hacerlos más legibles y fáciles de mantener.

Los archivos en los que se encuentra descrito el sensor son *erlecopter.xacro*, *generic_camera.urdf.xacro*, *lidar_sensor.urdf.xacro* y funcionan de manera similar a la descrita anteriormente para el caso del primer método. Se describirán sus códigos a continuación más detalladamente que en el apartado anterior ya que este es el método que se ha llevado implementado.

4.3.1 Archivo *lidar_sensor.urdf.xacro*

En este archivo se define como será el sensor, tanto físicamente como sus propiedades, además de los plugins necesarios para su correcto comportamiento.

En el código, lo primero que se puede apreciar es la versión de XML utilizada, la habilitación del formato Xacro y el nombre del archivo junto con las variables que se le han de pasar al mismo.

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

<robot xmlns:xacro="http://ros.org/wiki/xacro">

  <xacro:macro name="lidar_sensor" params="name parent *origin
ros_topic update_rate min_range max_range field_of_view_horizontal
field_of_view_vertical ray_count_horizontal ray_count_vertical
sensor_mesh">
```

Código 4.6 Archivo *lidar_sensor.urdf.xacro*: Primeras líneas

A continuación, aparecen la articulación y el enlace con el que se describe dicho sensor. La articulación es fija, no permitiendo así ningún movimiento entre el link padre y el hijo. Además, se le indica a través de los parámetros de entrada la posición del origen, el padre de la articulación y el hijo.

El link posee unos valores concretos de masa e inercia, mientras que tanto la visualización del mismo y sus colisiones están desactivadas, ya que se ha preferido utilizar el modelo visual del sonar que se encuentra implementado en el drone. Cabe mencionar que este modelo visual y estas colisiones son descritas por su origen

y la geometría que poseen, que puede ir desde un cubo o una esfera a modelos realizados con herramientas más potentes como pueden ser AutoCAD o CATIA. La etiqueta <*visual*> define el como se verá el sensor en la simulación mientras que la etiqueta <*collision*> describe la zona donde puede chocar el mismo con otros elementos del mundo.

```
<joint name="${name}_joint" type="fixed">
  <xacro:insert_block name="origin" />
  <parent link="${parent}"/>
  <child link="${name}_link"/>
</joint>

<link name="${name}_link">

  <inertial>
    <mass value="0.001" />
    <origin xyz="0 0 0" rpy="0 0 0" />
    <inertia  ixx="0.000000017"  ixy="0"  ixz="0"  iyy="0.000000017"
iyz="0" izz="0.000000017" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <!-- COMENTADO -->[2]
    </geometry>
  </visual>

   <!-- <collision>
     <origin xyz="0 0 0" rpy="0 0 0" />
     <geometry>
       <box size="0.01 0.01 0.01" />
     </geometry>
   </collision> -->

</link>
```

Código 4.7 Archivo *lidar_sensor.urdf.xacro*: Joint y link

Finalmente se define un elemento <*gazebo*> donde aparecerá el tipo de sensor que se está creando y los plugins adicionales. La mayoría de los parámetros de este bloque se reciben como entrada. Comentados en el código se pueden encontrar las definiciones de cada uno de los atributos.

```
<gazebo reference="${name}_link">
  <sensor type="ray" name="${name}"> <!--Sensor tipo “ray”-->
    <always_on>true</always_on> <!--Sensor siempre actualizado-->
    <update_rate>${update_rate}</update_rate> <!--Tasa de
actualización-->
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize> <!--Visualizado en el GUI-->
    <ray>
      <scan>
        <horizontal>
          <samples>${ray_count_horizontal}</samples> <!--Número de
rayos simulados por ciclo-->
```

[2] Los comentarios en URDF se establecen de la forma <*!- - COMENTARIO - -*>.

```
          <resolution>0.5</resolution> <!--Resolución-->
          <min_angle>-${field_of_view_horizontal/2}</min_angle> <!--
Ángulo mínimo-->
          <max_angle> ${field_of_view_horizontal/2}</max_angle> <!--
Ángulo máximo-->
        </horizontal>
        <vertical>
          <samples>${ray_count_vertical}</samples>
          <resolution>1</resolution>
          <min_angle>-${field_of_view_vertical/2}</min_angle>
          <max_angle> ${field_of_view_vertical/2}</max_angle>
        </vertical>
      </scan>
      <range>
        <min>${min_range}</min> <!--Mínima distancia del rayo-->
        <max>${max_range}</max> <!-- Máxima distancia del rayo -->
        <resolution>0.01</resolution>
      </range>
    </ray>

    <plugin name="gazebo_ros_${name}_controller"
filename="librotors_gazebo_sonar_plugin.so">
      <gaussianNoise>0.005</gaussianNoise> <!--Ruido Gaussiano-->
      <topicName>${ros_topic}</topicName> <!--Nombre del topic donde
publicar los datos-->
      <frameId>${name}_link</frameId>
    </plugin>
    <plugin name="gazebo_ros_${name}_controller"
filename="libgazebo_ros_laser.so">
      <topicName>/scan</topicName>
      <frameName>${name}_link</frameName>
    </plugin>
  </sensor>
</gazebo>
</xacro:macro>
</robot>
```

Código 4.8 Archivo *lidar_sensor.urdf.xacro*: Sensor y plugins

4.3.2 Archivo *generic_camera.urdf.xacro*

En este archivo se encuentra definido un sensor tipo cámara para simular el optical flow. En este caso no se incluirá el código debido a que, excepto en el caso de la etiqueta *<sensor>*, el resto son idénticas a las anteriores, simplemente cambiándole los valores de cada uno de sus elementos.

En este caso hay dos joints y dos links definidos, diferenciando así entre el cuerpo completo del sensor y la lente del mismo, ambos sujetos al padre por articulaciones fijas.

En este caso, el sensor es de tipo *camera*, y de nuevo, la mayoría de sus parámetros se reciben como entrada.

```
<sensor type="camera" name="${name}_camera_sensor">
  <update_rate>${update_rate}</update_rate>
  <camera>
    <horizontal_fov>${hfov * M_PI/180.0}</horizontal_fov> <!-- Campo
de vista horizontal -->
    <image>
      <format>${image_format}</format> <!-- Formato de la imagen -->
      <width>${res_x}</width> <!-- Ancho de la imagen -->
      <height>${res_y}</height> <!-- Altura de la imagen -->
```

```
    </image>
    <clip> <!-- Distancias que se traducen -->
      <near>0.01</near>
      <far>100</far>
    </clip>
  </camera>
```

Código 4.9 Archivo *generic_camera.urdf.xacro*: Sensor

4.3.3 Archivo *erlecopter.xacro*

En definitiva, este es el bloque encargado de aunar en el modelo de Erle-Copter todos aquellos sensores o actuadores que se deseen, los cuales están descritos a parte como es el caso del LIDAR y la cámara anteriormente expuestos.

La forma en que se añaden dichos sensores es mediante etiquetas *<xacro:include>*, y enviándole a cada modelo los parámetros de entrada que se requieran. En la tabla 4.10 se puede apreciar el como están incluidos los modelos de los subapartados 4.3.1 y 4.3.2, y los parámetros que se les proporciona como entrada a cada uno de ellos.

```
  <xacro:include filename="$(find
ardupilot_sitl_gazebo_plugin)/urdf/sensors/lidar_sensor.urdf.xacro" />
   <xacro:lidar_sensor
    name="sonar2"
    parent="base_link"
    ros_topic="sonar_front"
    update_rate="10"
    min_range="0.01"
    max_range="10.0"
    field_of_view_horizontal="${0}"
    field_of_view_vertical="${0}"
    ray_count_horizontal="140"
    ray_count_vertical="1"

sensor_mesh="lidar_lite_v2_withRay/meshes/lidar_lite_v2_withRay.dae">
    <origin xyz="-0.071 0.0 -0.049" rpy="0 ${90*M_PI/180} 0"/>
  </xacro:lidar_sensor>


  <!-- Downward facing camera -->
  <xacro:include filename="$(find
ardupilot_sitl_gazebo_plugin)/urdf/sensors/generic_camera.urdf.xacro"
/>
  <xacro:generic_camera
    name="erlecopter/bottom"
    parent="base_link"
    ros_topic="image_raw"
    cam_info_topic="camera_info"
    update_rate="60"
    res_x="640"
    res_y="360"
    image_format="R8G8B8"
    hfov="81"
    framename="erlecopter_bottomcam">
    <origin xyz="-0.051 0.0 -0.056" rpy="0 ${(M_PI/2)-0.122173} 0"/>
  </xacro:generic_camera>
```

Código 4.10 Archivo *erlecopter.xacro*: Inclusión de los modelos del LIDAR y el optical flow

4.3.4 Resultado y conclusiones del segundo método

Una vez analizados los archivos anteriores se consigue entender por completo el funcionamiento del formato URDF junto con Xacro. Es importante decir que algunos de los valores iniciales que poseen dichos archivos respecto a los que aparecen en las tablas anteriores no son los mismos. El que más interesa cambiar es la posición de origen de ambos sensores en el archivo *erlecopter.xacro*, ya que inicialmente ambos sensores se encuentran en la parte delantera del drone, mientras que en la realidad están situados en la parte inferior trasera, con el fin de utilizar la cámara para la detección de la carga. Además, se gira el ángulo de la cámara 7º para conseguir centrar la carga en la imagen.

Tras actualizar todos los modelos, se puede llevar a cabo la simulación para comprobar que todo funciona como se esperaba, para lo cual se han de seguir los comandos 3.4, 3.5 y 3.6.

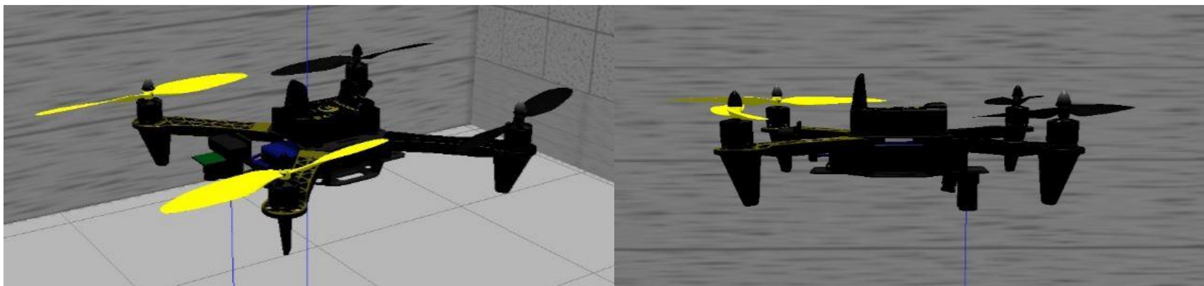

Figura 4.4 Drone con optical flow delante (izquierda) vs drone con optical flow detrás (derecha)

Cuando se ha cargado la simulación se han de modificar una serie de parámetros del drone, preferiblemente antes de comenzar el vuelo, para poder utilizar el optical flow. Esto se lleva a cabo desde el terminal del MAVProxy como se muestra en la tabla de comandos 4.1.

```
param set SIM_FLOW_ENABLE 1
param set FLOW_ENABLE 1
```

Comando 4.1 Comandos para activación del optical flow

Una vez cambiados estos parámetros, se reinicia la simulación para que se carguen los mismos. Se pueden comprobar sus nuevos valores mediante el comando *param show*.

```
param show SIM_FLOW_ENABLE
param show FLOW_ENABLE
```

Comando 4.2 Comandos para la comprobación de los valores de los parámetros

Tras esto, se cargan los gráficos donde aparecerán en uno de ellos los valores en los ejes x e y recogidos por el optical flow y en el otro la altura a la que se encuentra el drone.

```
module load graph
graph OPTICAL_FLOW.flox_x OPTICAL_FLOW.flox_y
graph RANGEFINDER.distance
```

Comando 4.3 Comandos para graficar valores del optical flow y de altura

Finalmente, para comprobar que se obtienen valores coherentes, se puede realizar un vuelo tal y como se indica en los comandos 3.7 o 3.8. En la figura 4.5 se muestra un ejemplo de esto, con asfalto en el suelo, ya que es como el optical flow funciona correctamente. Más adelante se explicará la forma de añadir este tipo de textura al suelo de la simulación.

Además, se adjunta una segunda figura 4.6 en la que aparecen mejor representados los valores del optical flow. Esto tiene una explicación sencilla, y es que en la primera imagen simplemente se despega el drone, sin traslación ninguna en los ejes *x* e *y*, correspondiéndose ese pico en los valores del optical flow al momento del despegue, en el que puede producirse alguna perturbación. Es decir, en dicha imagen lo que mejor se puede apreciar es la lectura del LIDAR en la esquina superior derecha, en la que tras ordenar al drone que realice un despegue de 4 metros, se ve como se van recogiendo dichos valores con ayuda del sensor.

Respecto a la segunda imagen, en ella se le ha aplicado una velocidad de referencia máxima a los canales 1 y 2 correspondientes al roll y al pitch respectivamente, todo esto utilizando el modo LOITER. De esta forma, se puede apreciar como ahora si que se recogen valores razonables con el optical flow.

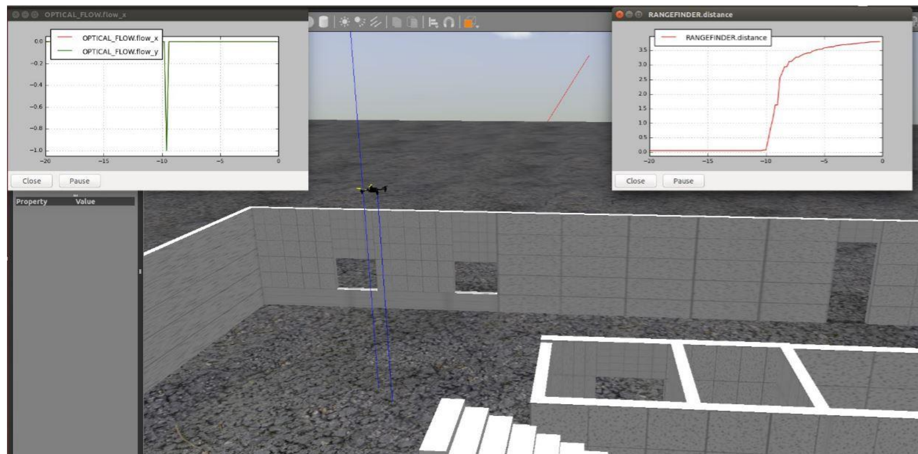

Figura 4.5 Despegue del Erle-Copter con medidas del optical flow y el LIDAR

Figura 4.6 Traslación en horizontal del Erle-Copter con medidas del optical flow y el LIDAR

5 CREACIÓN DEL MUNDO DE TRABAJO

Tras realizar todos los pasos de los capítulos anteriores ya se tiene la simulación principal del proyecto, la que se compone del Erle-Copter, con todos sus sensores, dentro de un mundo en el que realizar experimentos.

Aún así, aún faltan algunos retoques por hacer sobre dicho entorno, con los cuales finalizarían las tareas de modelado. Estos son la adición de la carga unida al drone y, a causa de dicha unión, se debe modificar la posición inicial del drone para que se encuentre sobre un soporte de un metro de altura, con el fin de que la pelota parta de una posición de suspensión.

En este quinto capítulo se describirán a modo introducción los distintos archivos que componen una simulación de Gazebo para, a continuación, presentar los códigos creados para el mencionado soporte y para la carga. Finalmente, se expondrán los resultados de estas actualizaciones tras ser probadas.

5.1. Archivos de una simulación

Muchos son los archivos que influyen en la simulación de un entorno en Gazebo y, ya que a continuación se pretende describir el procedimiento que se ha seguido a la hora de crear o modificar algunos de ellos, se presentará en este apartado una breve introducción de los mismos.

5.1.1 Archivos *.world*

Estos se tratan de archivos en formato SDF que, como su propio nombre indica, incluyen, tal y como se explicó en el subapartado 4.2.3, la descripción de un mundo. Esto implica que en su interior se definen todos los componentes que aparecerán en un determinado mundo y las propiedades y plugins del mismo. Comentar que este tipo de archivos se encuentran ubicados dentro del entorno instalado en el apartado 3.1, en el directorio *~/simulation/ros_catkin_ws/src/ardupilot_sitl_gazebo_plugin/ardupilot_sitl_gazebo_plugin/worlds*.

Dentro de estas propiedades aparecen especificaciones como el cielo del mundo, la gravedad, el viento, especificaciones físicas y de tiempo de la simualción o la luz que incidirá sobre la misma.

Para indicar los elementos que se desea que aparezcan en el entorno que se cree, se deben incluir los mismos mediante etiquetas *<include>*, o creando dentro de estos archivos bloques tipo *<model>*.

5.1.2 Archivos *.urdf*, *.xacro* y *.sdf*

Como se presentó en el apartado 4.2, los distintos modelos que describen cualquier sistema físico en Gazebo pueden estar en formato SDF o en URDF, pudiéndose a su vez utilizar Xacro en este segundo caso.

Estos modelos se encuentran ubicados, dentro del entorno instalado en el apartado 3.1, tanto en la dirección *~/simulation/ros_catkin_ws/src/ardupilot_sitl_gazebo_plugin/ardupilot_sitl_gazebo_plugin/urdf* como en *~/.gazebo/models*. Esta última se trata de una carpeta oculta dentro del directorio personal.

5.1.3 Archivos *.launch*

Este tipo de archivos también están descritos en formato XML y son aquellos que pueden ser ejecutados mediante el uso del paquete de ROS *roslaunch*.

Dentro de dichos documentos se encuentran, si se refieren a una simulación en Gazebo, los diferentes archivos que se desean ejecutar al lanzar una simulación, además de todas las condiciones iniciales de dicha simulación y argumentos específicos que deban adquirir determinados parámetros de la misma. En realidad, al ejecutar el comando 3.5, lo que se esta haciendo es lanzar varios archivos de este tipo, los cuales, a su vez, indican qué otros archivos se deben ejecutar, dentro de los cuales hay incluidos más documentos referentes a modelos, y así sucesivamente, de manera que se va creando un árbol de archivos que van ejecutándose y referenciándose entre ellos, generando finalmente la simulación deseada.

5.2. Plataforma

Frente a la necesidad de modelar una carga suspendida del drone a 60 cm, lo ideal sería disponer de una plataforma de aproximadamente un metro de altura, con el fin de que, al iniciar la simulación, el drone estuviese sobre ella, y la carga colgada, sin contacto ninguno con el suelo.

Dicho modelo se creará en SDF, debido a las facilidades que este formato ofrece en un proceso de modelado, y ya que no es necesario adjuntar el mismo al de Erle-Copter, sino que simplemente debe incluirse en el mismo mundo.

Este soporte será lo más básico posible, tratándose simplemente de cuatro patas cilíndricas de un metro de altura sobre las cuales se apoyará el drone. El código 5.1 se corresponde con el modelado de dicha plataforma.

```
<?xml version='1.0'?>
<sdf version='1.4'>
  <model name="soporte_drone">
    <static>false</static>
    <link name='first_leg'>
      <pose>0.141 -0.141 0.11 0 0 0</pose>
      <collision name='first_leg_collision'>
      <geometry>
        <cylinder>
          <radius>.7</radius>
          <length>1</length>
        </cylinder>
      </geometry>
      </collision>
      <visual name='first_leg_visual'>
      <geometry>
        <cylinder>
          <radius>.7</radius>
          <length>1</length>
        </cylinder>
      </geometry>
        <material>
          <script>
            <uri>file:Gazebo/media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Black</name>
          </script>
        </material>
      </visual>
    </link>
    <link name='second_leg'>
      <pose>-0.141 0.141 0.11 0 0 0</pose>
      <collision name='second_leg_collision'>
      <geometry>
        <cylinder>
          <radius>.7</radius>
          <length>1</length>
        </cylinder>
      </geometry>
      </collision>
      <visual name='second_leg_visual'>
      <geometry>
        <cylinder>
          <radius>.7</radius>
          <length>1</length>
        </cylinder>
      </geometry>
        <material>
```

```
        <script>
          <uri>file:Gazebo/media/materials/scripts/gazebo.material</uri>
          <name>Gazebo/Black</name>
        </script>
      </material>
    </visual>
  </link>
  <joint name="first-second_leg_joint" type="fixed">
    <parent>first_leg</parent>
    <child>second_leg</child>
  </joint>
  . . .
```

Código 5.1 Fragmento del código del soporte

Para que no ocupe demasiado, solo se incluye el fragmento del código para dos de las patas del soporte, ya que se puede extrapolar dicho formato al resto de patas.

Como se puede apreciar en el código, se define un link para cada pata de la plataforma, llamados *first_leg*, *second_leg*, *third_leg* y *fourth_leg*. La posición de cada una de ellas se define de manera que entre las cuatro formen un cuadrado con centro en el origen de coordenadas del mundo de Gazebo.

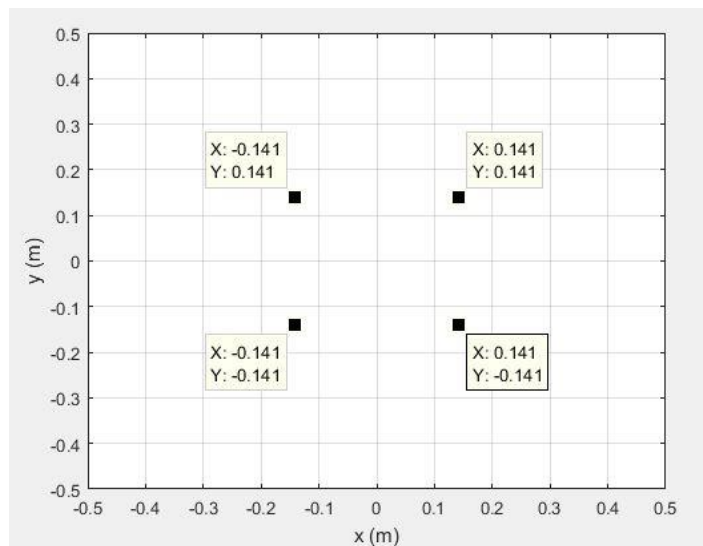

Figura 5.1 Posiciones de las patas de la plataforma

Por otro lado, tanto la parte de colisión como la parte visual tienen los mismos componentes para todas las patas, de manera que estas sean iguales entre ellas. Dichos bloques describen cilindros de un radio de 7 cm y de una altura de un metro. Además, en la parte visual se especifica el color que se quiere que adquieran las patas. La carpeta indicada donde se encuentra la definición del color debe estar ubicada en la misma dirección que el modelo del soporte para poder utilizarlo.

Finalmente, se puede observar como están descritas las articulaciones. Estas son de tipo fijo y van uniendo la primera pata con la segunda, la segunda con la tercera, la tercera con la cuarta y esta de nuevo con la primera. Así se consigue que todas ellas se comporten como un conjunto, proporcionando una mayor estabilidad al soporte.

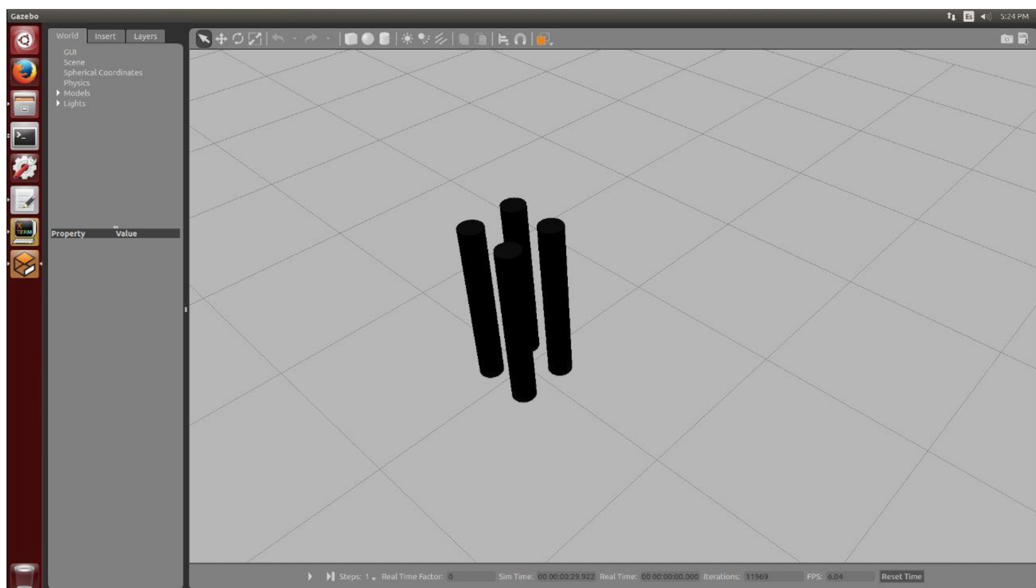

Figura 5.2 Modelo de la plataforma

5.2.1 Modificación de las colisiones del drone

Tras conseguir una simulación válida del soporte, es necesario situar el drone encima del mismo. Lo referente a el como se ha modificado su posición inicial se comentará en el apartado 5.4, pero hay que tener otro aspecto en cuenta, las colisiones.

Tal y como están definidas en el modelo de Erle-Copter, el mismo que se ha instalado y montado, las colisiones que posee el drone son una para cada hélice de forma cilíndrica y un cubo para el frame, cuya altura va desde el límite inferior de las patas, sin incluir las mismas, hasta la parte superior del drone. Se incluye una imagen a continuación para aclarar dicho concepto.

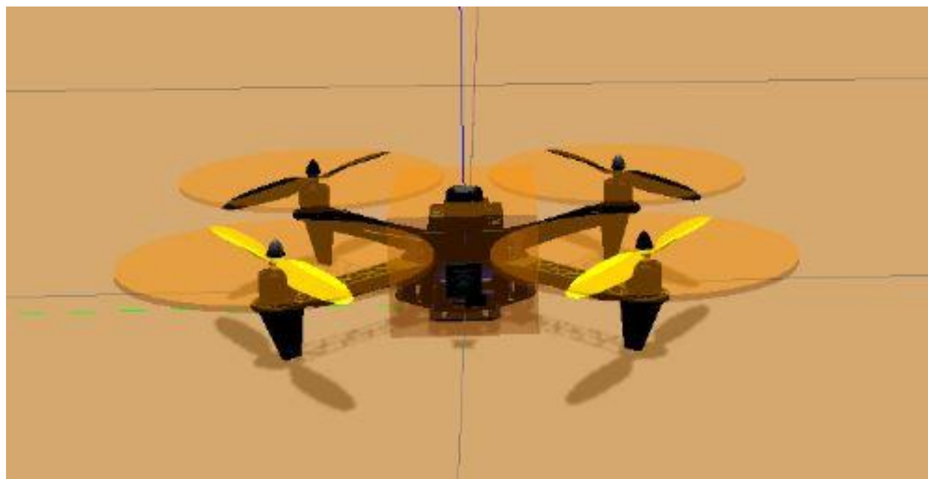

Figura 5.3 Colisiones iniciales del Erle-Copter

De este modo el drone no se sostendrá sobre el soporte, que es lo que se pretende, ya que no posee colisiones en la base de las patas. Es por esto por lo que se ha de realizar un archivo que defina dichas colisiones, y es el que recibe el nombre de *legs_bases.urdf.xacro*. En este, tal y como se muestra en el código 5.2, se incluyen al final de cada pata unos links compuestos únicamente por colisiones de geometría cúbica y muy pequeñas pero suficientes para hacer que el Erle-Copter se mantenga sobre el soporte. Estos cuatro links, uno por cada pata, se unen al frame del drone mediante articulaciones fijas.

```
<?xml version="1.0" ?>
<robot name="legs_bases">
  <link name= "leg1_link">
    <collision name="leg1_base_collision">
      <origin xyz="-0.134 0.134 -0.0495" rpy="0 0 0" />
      <geometry>
        <box size="0.01 0.01 0.0005"/>
      </geometry>
    </collision>
  </link>
  <joint name="leg1_joint" type="fixed">
    <origin xyz="0 0 0" rpy="0.0 0.0 0.0" />
    <parent link="base_link" />
    <child link="leg1_link"/>
  </joint>
  <link name= "leg2_link">
    <collision name="leg2_base_collision">
      <origin xyz="0.134 0.134 -0.0495" rpy="0 0 0" />
      <geometry>
        <box size="0.01 0.01 0.0005"/>
      </geometry>
    </collision>
  </link>
  <joint name="leg2_joint" type="fixed">
    <origin xyz="0 0 0" rpy="0.0 0.0 0.0" />
    <parent link="base_link" />
    <child link="leg2_link"/>
  </joint>
  ...
```

Código 5.2 Fragmento del código *legs_bases.urdf.xacro*

Solo se incluyen las colisiones de dos de las patas ya que el resto se realiza de igual manera, evitando así una extensión innecesaria de dicho código en el presente documento.

Una vez realizado esto, e incluyendo este modelo en el general, se obtiene el resultado mostrado en la Figura 5.4.

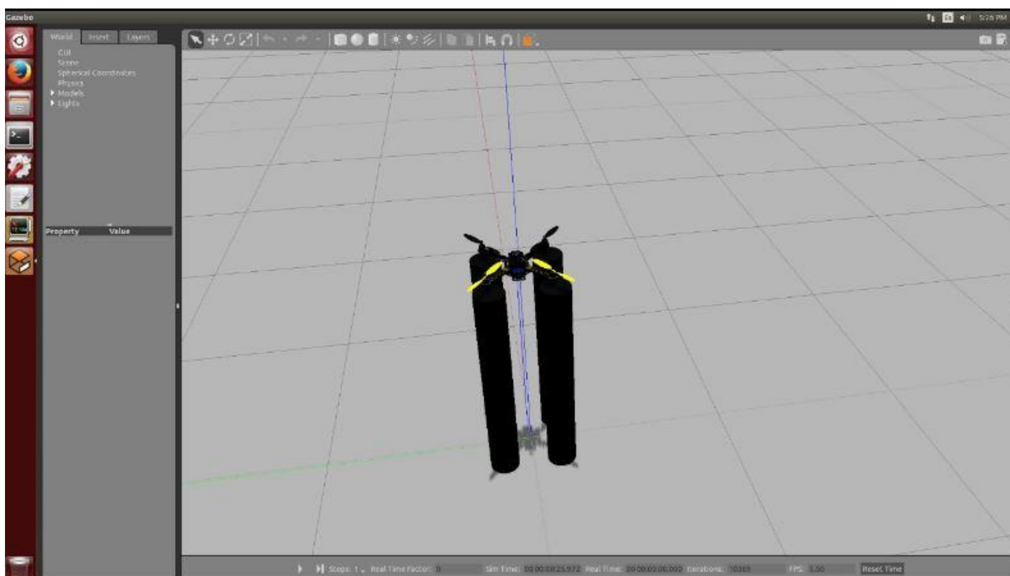

Figura 5.4 Erle-Copter sobre plataforma

5.3. Carga

El modelado de la carga es otro de los puntos más importantes de este proyecto. En definitiva, se ha de conseguir simular una pelota naranja colgada a 60 cm del drone y que posea una dinámica lo más parecida posible a la que tendría una carga suspendida de un hilo.

El modelo de la carga se encuentra definido en formato URDF en un archivo independiente llamado *charge.urdf.xacro*, el cual se incluye en *erlecopter.xacro* tal y como se indicará en el apartado 5.4.

Se puede dividir su código en dos bloques diferentes: uno en el que se define el modelo físico de la carga y otro mediante el cual se consigue simular la dinámica de la misma como si se encontrase colgada de un hilo.

5.3.1 Primer bloque: modelado físico e inercial de la carga

Tal y como se puede apreciar e el código 5.3, la carga esta compuesta simplemente por una esfera con unos determinados valores de masa y de inercia. El radio de la misma es de dos cm, el cual se especifica tanto para la visualización como para la colisión de la misma. Este valor se ha sacado tomando como referencia el radio de una pelota de ping-pong oficial, que es la que en definitiva se utilizará. Además, otra de las especificaciones es que dicha pelota estará en un principio llena de agua. Con estos datos, y conociendo la ecuación del volumen de una esfera, así como la densidad del agua, se puede calcular la masa total de agua y sumársela a la de la pelota. De este modo:

$$V_{esf} = \frac{4}{3}\pi R_{esf}^3, R_{esf} = 2\ cm \tag{5.1}$$

$$\rho_{agua} = \frac{m_{agua}}{V_{esf}}, \rho_{agua} = 1\frac{g}{cm^3} \tag{5.2}$$

$$\rho_{agua} = \frac{m_{agua}}{\frac{4}{3}\pi R_{esf}^3} \tag{5.3}$$

$$m_{agua} = \frac{4}{3}\rho_{agua}\pi R_{esf}^3 = 8.377\ g \tag{5.4}$$

Sabiendo que la masa de una pelota de ping-pong es aproximadamente 2.7 g, la masa total de la carga sería:

$$m_{total} = m_{pelota} + m_{agua} = 2.7 + 8.377 = 11\ g \tag{5.5}$$

Una vez se tiene esto, para el cálculo de las inercias se tiene en cuenta el Teorema de Steiner, el cual afirma que si se conoce el momento de inercia con respecto a un eje que pase por el centro de masas de un objeto, entonces se puede conocer el momento de inercia con respecto a cualquier otro eje paralelo a este primero que se encuentre a una distancia L de la siguiente manera:

$$I = I_{CM} + mL^2 \tag{5.6}$$

Donde I_{CM} representa el momento de inercia en el centro de masas del objeto y m se corresponde con la masa total del mismo.

En este caso, tenemos que la distancia del hilo es de 60 cm, mientras que los valores de radio y masa son los presentados anteriormente. Además, al ser la carga una esfera, el momento de inercia de su centro de masas sería:

$$I_{CM} = \frac{2}{5} m R^2 \tag{5.7}$$

Sabido esto, se puede calcular la matriz de inercias en los ejes de la carga, sabiendo que:

- Los términos cruzados de la matriz son nulos.
- El efecto del cable solo se tiene para los ejes *x* e *y*, siendo este nulo en *z*.

Siguiendo esas dos pautas, y aplicando el teorema de Steiner, se concluye que la matriz de inercias será la siguiente:

$$I = \begin{bmatrix} \left(\frac{2}{5} m_{total} R_{esf}^2\right) + m_{total} L^2 & 0 & 0 \\ 0 & \left(\frac{2}{5} m_{total} R_{esf}^2\right) + m_{total} L^2 & 0 \\ 0 & 0 & \frac{2}{5} m_{total} R_{esf}^2 \end{bmatrix} \tag{5.8}$$

Sustituyendo en la anterior matriz los distintos datos que se tienen, sabiendo que la masa ha de estar el kg y la distancia en metros:

$$I = \begin{bmatrix} 3.962 \times 10^{-3} & 0 & 0 \\ 0 & 3.962 \times 10^{-3} & 0 \\ 0 & 0 & 1.76 \times 10^{-6} \end{bmatrix} Kgm^2 \tag{5.9}$$

Estos son los correspondientes valores de inercia que se añadirán a la carga, tal y como se muestra en el código 5.3.

Se puede apreciar que tanto el origen del bloque <*inertial*> como en los de <*collision*> y <*visual*> poseen un valor de -0.6. Esto se debe a que dicho origen esta referenciado al link padre, que en este caso será el frame del drone, llamado *base_link* en el código.

También se indica al final del código el color que se desea que posea la carga, en este caso naranja.

```
<?xml version="1.0" ?>
<robot name="suspended_charge" xmlns:xacro="http://ros.org/wiki/xacro">
  <link name="suspended_charge_link">
    <inertial>
      <mass value="0.011"/>
      <origin rpy="0  0  0" xyz="0   0   -0.6"/>
      <inertia ixx="3.962e-3" ixy="0" ixz="0" iyy="3.962e-3" iyz="0"
izz="1.76e-6"/> <!-- [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] -
->
    </inertial>
    <collision>
      <origin rpy="0  0  0" xyz="0   0   -0.6"/>
      <geometry>
        <sphere radius="0.02"/>
      </geometry>
    </collision>
    <visual>
      <origin rpy="0  0  0" xyz="0   0   -0.6"/>
      <geometry>
        <sphere radius="0.02"/>
      </geometry>
    </visual>
  </link>
```

```
  ...

  <gazebo reference="suspended_charge_link">
    <material>Gazebo/Orange</material>
  </gazebo>
</robot>
```

Código 5.3 Fragmeto del código *charge.urdf.xacro* destinado al modelado físico de la carga

5.3.2 Segundo bloque: modelado del comportamiento dinámico de la carga

La segunda parte en la que se ha dividido la explicación del código consiste en como se ha simulado el movimiento de balanceo de la carga.

Más allá de la adición de inercias a la misma, se debía encontrar alguna forma de simular el movimiento que poseería una carga suspendida de un hilo, es decir, un movimiento de balanceo.

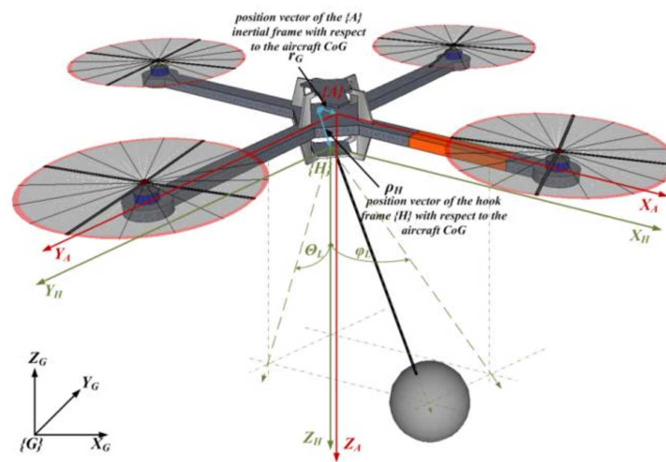

Figura 5.5 Drone con carga suspendida tomado de [20]

En formato SDF sería simple la simulación de dicho movimiento, ya que solo con incluir una articulación de tipo *ball* a la distancia deseada entre la carga y el drone, y aplicándole a la misma las inercias adecuadas, se obtendría un resultado bastante realista. El problema está en que en este caso se esta usando URDF con el fin de poder incluir dicho modelo de manera directa al modelo *erlecopter.xacro*, y en dicho formato no existe la articulación tipo *ball*.

La solución a este problema reside en la encadenación de tres articulaciones rotativas, una por cada eje. Se muestra en el código 5.4 la forma de implementar dicha solución.

```
...

<joint name="ChargePsi" type="continuous">
    <parent link="base_link"/>
    <child link="ChargePsi_link"/>
    <axis xyz="1 0 0"/>
  </joint>

  <link name="ChargePsi_link">
    <inertial>
      <mass value="5e-6"/>
```

```
      <inertia ixx="5.8083e-9" ixy="0" ixz="0" iyy="5.8083e-9" iyz="0"
izz="5.8083e-9"/>
    </inertial>
  </link>

  <joint name="ChargeTheta" type="continuous">
    <parent link="ChargePsi_link"/>
    <child link="ChargeTheta_link"/>
    <axis xyz="0 1 0"/>
  </joint>

  <link name="ChargeTheta_link">
    <inertial>
      <mass value="5e-6"/>
      <inertia ixx="5.8083e-9" ixy="0" ixz="0" iyy="5.8083e-9" iyz="0"
izz="5.8083e-9"/>
    </inertial>
  </link>

  <joint name="ChargePhi" type="continuous">
    <parent link="ChargeTheta_link"/>
    <child link="suspended_charge_link"/>
    <axis xyz="0 0 1"/>
  </joint>
...
```

Código 5.4 Fragmeto del código *charge.urdf.xacro* destinado al comportamiento dinámico de la carga

Se puede observer que a los enlaces *ChargePsi_link*, *ChargeTheta_link* y *ChargePhi_link* se les asigna masas e inercias muy pequeñas con el fin de que, frente a las masas e inercias del resto del modelo, sean despreciadas. El lector puede preguntarse el por que es necesario entonces añadir dichos valores, pudiendo establecerlos a cero. La respuesta a esta cuestión reside en un argumento realizado en el subapartado 4.2.4, el cual explica que, en el caso en el que un link posea valores de masa o de inercias nulos, Gazebo lo ignorará al realizar la conversión de formato URDF a SDF.

Con este segundo bloque se completa el código de la carga suspendida, obteniéndose el resultado mostrado en la figura 5.6.

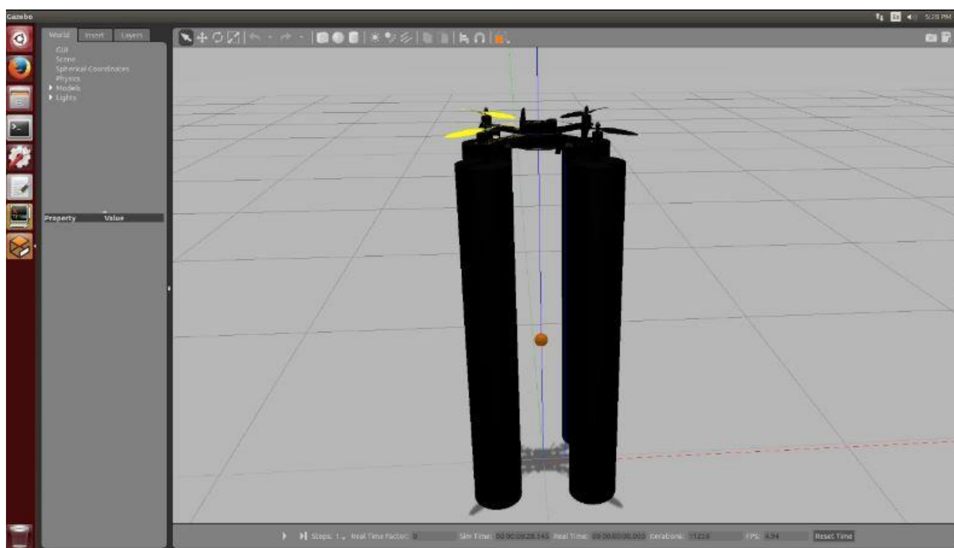

Figura 5.6 Erle-Copter con soporte y carga suspendida

5.4. Modificaciones necesarias en el resto del código

Ya se posee el modelo completo que se buscaba desde el inicio de este proyecto, una carga suspendida unida al Erle-Copter a 60 cm del mismo, con una dinámica similar a la que poseería la carga si estuviese colgada mediante un hilo y con un soporte de un metro de altura para que la posición inicial del drone y la carga sea la mostrada en la figura 5.6.

Sin embargo, no solo basta con las modificaciones llevadas a cabo en los apartados anteriores, sino que se requiere de la correcta inclusión de esos modelos creados al código general de la simulación. Además, resulta favorable cambiar la textura del suelo del entorno para que sea de asfalto, ya que de esta forma el óptical flow realiza mejores medidas. Por último, con el objetivo de facilitar los experimentos que se explicarán en el capítulo 8, se deben retirar las paredes del mundo ofrecido por Erle Robotics.

5.4.1 Modificacion del archivo *empty.world*

Este es el nombre del archivo *.world* de la simulación del Erle-Copter. Si se observa detenidamente su código, además de todas las especificaciones del entorno a simular, las cuales no son objeto de modificación en este proyecto, también aparecen, como se explica en el subapartado 5.1.1, los diferentes modelos que se deben incluir en este mundo. Inicialmente, estos modelos se tratan de *ground_plane*, que no es más que la colisión del suelo para que los diferentes objetos que se situen en el mundo se mantegan sobre el, y el modelo *erle*, que es el edificio que aparece en la simulación.

Tal y como se ha explicado previamente, el procedimiento a seguir para que en el mundo aparezcan el soporte y el suelo de asfalto y que desaparezcan las paredes consiste en comentar el modelo de *erle* e incluir los modelos *asphalt_plane* y *soporte*, ambos en formato SDF y los cuales deben estar en la ubicación *~/.gazebo/models*. El archivo *empty.world* quedaría entonces tal y como se muestra en el código 5.5.

```
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://asphalt_plane</uri>
    </include>

    <include>
      <uri>model://soporte</uri>
    </include>-->

<!--    <include>
      <uri>model://erle</uri>
      <pose>-10 -5 0 0 0 0</pose>
    </include>
-->
```

Código 5.5 Modificaciones en el código *empty.world*

5.4.2 Modificacion del archivo *erlecopter.xacro*

Tal y como se ha explicado en apartados anteriores, este es el archivo de la simulación donde se encuentran los distintos componentes montados sobre el Erle-Copter, por tanto, es aquí donde se deben incluir los modelos de las colisiones de las patas del subapartado 5.2.1, así como el de la carga. Recordar que ambos modelos están en formato URDF y se deben incluir, con el fin de poder añadirlos al modelo tal y como se pretende, en la ubicación *~/simulation/ros_catkin_ws/src/ardupilot_sitl_gazebo_plugin/ardupilot_sitl_gazebo_plugin/urdf*.

La inclusión de estos dos archivos se lleva a cabo en la parte del código encabezada como `<!-- Included URDF Files -->` y de la forma en que se muestra en el código 5.6.

```
  <!-- Included URDF Files -->

  <!-- <xacro:include filename="$(find
rotors_description)/urdf/multirotor_base.xacro" /> -->

  <xacro:include filename="$(find
ardupilot_sitl_gazebo_plugin)/urdf/multirotor_base.xacro" />

  <xacro:include filename="$(find
ardupilot_sitl_gazebo_plugin)/urdf/legs_bases/legs_bases.urdf.xacro" />

  <xacro:include filename="$(find
ardupilot_sitl_gazebo_plugin)/urdf/charge/charge.urdf.xacro" />
```

Código 5.6 Modificaciones en el código *erlecopter.xacro*

5.4.3 Modificación del archivo *erlecopter_spawn.launch*

Este es el archivo *.launch* de la simulación del Erle-Copter. En el, tal y como se comentó en el subapartado 5.1.3 se establecen todas las condiciones iniciales de la simulación y se indican todos los archivos que han de ser lanzados con la ejecución de este archivo *.launch*.

Es preciso modificar este código con el único fin de que el Erle-Copter adquiera como posición inicial aquella que lo sitúa sobre la plataforma creada. Dentro del archivo, hay una parte encabezada por el comentario `<!-- Initial pose for the drone -->` dentro de la cual se debe modificar la posición en metros de *z*, sabiendo que esta es positiva hacia arriba. Ese cambio se muestra en el código 5.7.

```
  <!-- Initial pose for the drone -->
  <arg name="x" default="0.0"/> <!-- [m], positive to the North -->
  <arg name="y" default="0.0"/> <!-- [m], negative to the East -->
  <arg name="z" default="1.21"/> <!-- [m], positive Up -->
  <arg name="roll" default="0"/> <!-- [rad] -->
  <arg name="pitch" default="0"/> <!-- [rad] -->
  <arg name="yaw" default="3.1415"/> <!-- [rad], negative clockwise -->
```

Código 5.7 Modificación del código *erlecopter_spawn.launch*

El valor de z = 1.21 metros se debe a 1 metro de la plataforma, 10 cm del plano del asfalto, 10 cm de altura de las patas del drone y 1 cm de seguridad, ya que esta posición en z se refiere al centro del Erle-Copter.

5.5. Resultados y conclusiones

Finalmente, se ha conseguido obtener un modelo del Erle-Copter más la carga bastante fiel a la realidad, cuyo resultado se muestra en la figura 5.7, en la que se puede ver el drone con la pelota sobre el soporte, y el suelo de asfalto.

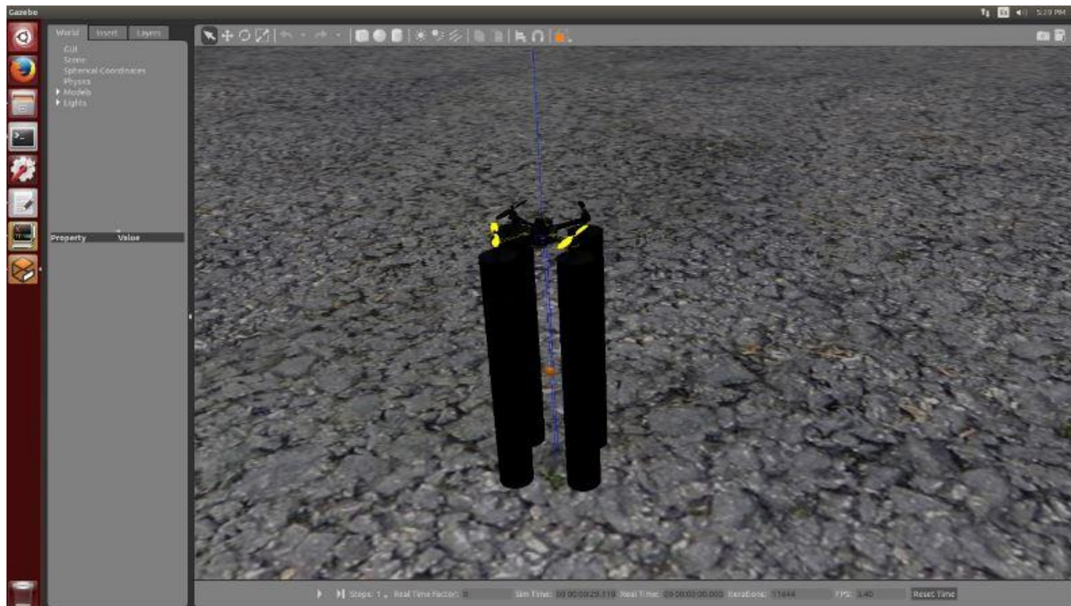

Figura 5.7 Simulación finalizada

Una vez acabado dicho modelado, se puede llevar a cabo una prueba del comportamiento del mismo, para ello se siguen los mismos pasos expuestos en el apartado 3.3. Cabe resaltar que debido a la variación de la posición inicial del drone, se produce una ligera pérdida de comunicación al inicio de la simulación, la cual dura unos segundos y posteriormente se recupera y se puede efectuar el armado y manejo del Erle-Copter de manera normal.

En la figura 5.8 aparece el drone una vez despegado y tras haber realizado una determinada trayectoria. En esta se puede apreciar, prestando un poco de atención, el balanceo de la carga durante dicha trayectoria, ya que esta aparece desviada un cierto ángulo hacia la izquierda respecto a la posición del drone.

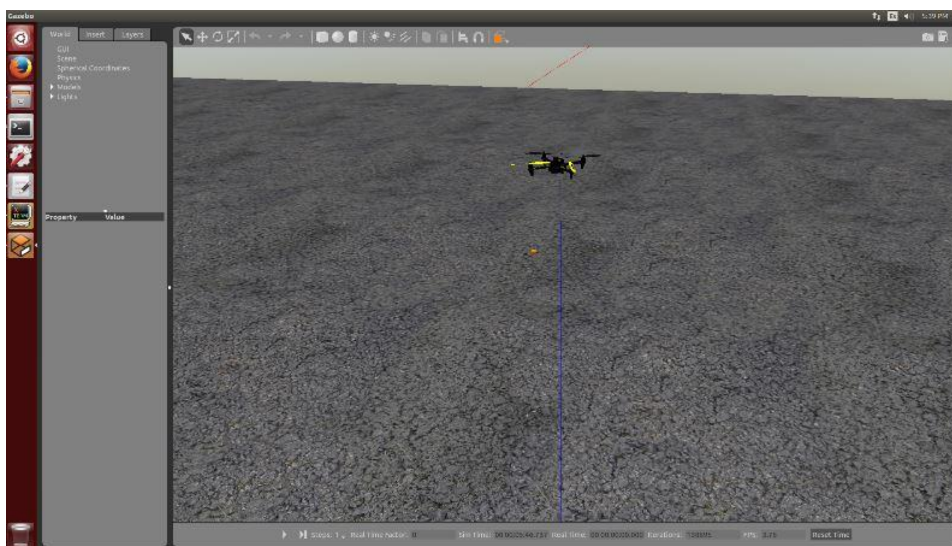

Figura 5.8 Erle-Copter volando con el modelado del entorno finalizado

6 COMUNICACIÓN

El proyecto se encuentra ahora en una nueva perspectiva. Ya se posee todo el entorno de simulación completo, con todos los elementos y comportamientos deseados. A partir de este momento el nuevo objetivo planteado se fija en conseguir una comunicación entre aquel ordenador en el que esta implementado todo el entorno, el cual desde este momento recibirá el nombre de PCDRONE, y aquel desde el cual se mandarán todos los comandos para el manejo de la simulación y el control de la misma, el cual desde ahora se le llamará PCUSER. De esta forma se consigue tratar al PCDRONE como si fuese un drone real, el cual únicamente recibe órdenes y actúa en consecuencia, mientras que el otro ordenador actuaría como estación de control de tierra o GCS.

A continuación, se describen dos tipos de comunicaciones que se han realizado, basadas en el mismo principio, mediante las cuales se ha conseguido el objetivo presentado. Además, ambos procesos de conexión entre los dos ordenadores servirán como introducción para el tercer método descrito en el capítulo 7.

6.1. Configuración inicial

Antes de comenzar se han de realizar algunas aclaraciones sobre las bases de la comunicación entre ambos ordenadores.

La idea principal consistía en llevar a cabo la comunicación vía wifi, utilizando para ello el dongle wifi de la empresa EDIMAX 802.11 de alta ganancia, pero ante esta propuesta apareció un problema. Al tratarse la máquina del PCDRONE de una máquina virtual de Ubuntu 14.04 sobre Windows 10, parece que existe algún tipo de incopatibilidad entre dicha versión de Linux y los drivers del dongle wifi. Tras estar un tiempo dedicado a la implementación de dicha conexión, finalmente se decidió abarcar la solución al problema de la comunicación por otros medios, para no dedicarle excesivo tiempo a este tema.

La alternativa al dongle wifi fue realizar la conexión de ambos ordenadores vía ethernet con la ayuda de un switch, una solución mucho más trivial que la anterior, ya que, al conectar ambos ordenadores a este dispositivo, el reparto de direcciones IP se lleva a cabo automáticamente. Lo único que hay que hacer es percatarse de las IPs que recibe cada uno de los ordenadores. Una vez conocidas estas, se puede probar que se ha establecido la comunicación a través de la ejecución de pings entre ambas máquinas. Por si resulta de ayuda al lector, el comando para conocer las características de las conexiones en Ubuntu desde el terminal, entre ellas la IP de un ordenador, se corresponde con el comando 6.1, mientras que para realizar un ping a otra máquina habría que ejecutar el comando 6.2, en el cual, en el campo IP se debe introducir la dirección IP del ordenador con el que se desea comprobar la conexión.

```
ifconfig
```

Comando 6.1 Comando para conocer las características de red de un ordenadoren Ubuntu

```
ping < IP >
```

Comando 6.2 Comando para comprobar la conexión entre dos ordenadores en Ubuntu

Una vez que se ha establecido la conexión entre las dos máquinas, lo primero que hay que hacer es cambiar las direcciones en ROS, para que se puedan comunicar los nodos a través de dicha conexión. Para esto, hay que modificar el archivo *.bashrc* de cada uno de los ordenadores.

```
pablo@pablo-EasyNote-TE11HC: ~
pablo@pablo-EasyNote-TE11HC:~$ ping 192.168.1.34
PING 192.168.1.34 (192.168.1.34) 56(84) bytes of data.
64 bytes from 192.168.1.34: icmp_seq=1 ttl=64 time=1.09 ms
64 bytes from 192.168.1.34: icmp_seq=2 ttl=64 time=1.00 ms
64 bytes from 192.168.1.34: icmp_seq=3 ttl=64 time=1.05 ms
64 bytes from 192.168.1.34: icmp_seq=4 ttl=64 time=1.04 ms
64 bytes from 192.168.1.34: icmp_seq=5 ttl=64 time=1.06 ms
64 bytes from 192.168.1.34: icmp_seq=6 ttl=64 time=1.11 ms
64 bytes from 192.168.1.34: icmp_seq=7 ttl=64 time=1.04 ms
64 bytes from 192.168.1.34: icmp_seq=8 ttl=64 time=1.07 ms
64 bytes from 192.168.1.34: icmp_seq=9 ttl=64 time=1.04 ms
64 bytes from 192.168.1.34: icmp_seq=10 ttl=64 time=1.02 ms
```

```
tfg@ubuntu: ~
tfg@ubuntu:~$ ping 192.168.1.35
PING 192.168.1.35 (192.168.1.35) 56(84) bytes of data.
64 bytes from 192.168.1.35: icmp_seq=1 ttl=64 time=0.794 ms
64 bytes from 192.168.1.35: icmp_seq=2 ttl=64 time=1.03 ms
64 bytes from 192.168.1.35: icmp_seq=3 ttl=64 time=0.986 ms
64 bytes from 192.168.1.35: icmp_seq=4 ttl=64 time=0.776 ms
64 bytes from 192.168.1.35: icmp_seq=5 ttl=64 time=1.00 ms
64 bytes from 192.168.1.35: icmp_seq=6 ttl=64 time=1.03 ms
64 bytes from 192.168.1.35: icmp_seq=7 ttl=64 time=0.904 ms
64 bytes from 192.168.1.35: icmp_seq=8 ttl=64 time=1.05 ms
```

Figura 6.1 Ping entre el PCUSER (izquierda) y el PCDRONE (derecha)

En el PCDRONE se ha de cambiar en este archivo la línea de ROS_HOSTNAME, y adjudicarle el nombre que se desee a dicha máquina, tal y como se muestra en el código 6.1.

```
export ROS_HOSTNAME=PCDRONE
```

Código 6.1 Modificación del archivo *.bashrc* en el PCDRONE

En lo referente al PCUSER, se debe incluir en su archivo *.bashrc* correspondiente una IP para el ROS Master, que será la del PCDRONE, una IP para el segundo nodo, que será la del PCUSER, y un nombre para este ordenador, tal y como se indica en el código 6.2.

```
export ROS_HOSTNAME=PCUSER
export ROS_MASTER_URI)http://< IP_PCDRONE >:11311
export ROS_IP=< IP_PCUSER >
```

Código 6.2 Modificación del archivo *.bashrc* en el PCUSER

De esta forma ya se tienen configurados ambos ordenadores para poder comenzar con la comunicación entre los mismos utilizando los métodos que se exponen a continuación.

6.2. Comunicación por terminal

La primera prueba de comunicación entre ambos ordenadores consistirá en lanzar un terminal de MAVProxy desde el PCUSER y manejar a través de este la simulación del PCDRONE.

Una vez configurado todo lo comentado en el apartado 6.1, dicha prueba no debe generar ningún contratiempo. Lo único que habría que hacer es lanzar la simulación de Gazebo en el PCDRONE tal y como se ha hecho siempre, es decir, como se indica en el apartado 3.3, y, una vez arrancada la misma, desde el PCUSER se lanza un nuevo terminal de MAVProxy utilizando para ello el mismo método.

Tras esto, la comunicación vía MAVROS se llevará a cabo entre los dos ordenadores, permitiendo así el envío de comandos desde el PCUSER, y consiguiendo que el Erle-Copter actúe frente a ellos. En este caso, los comandos se deben realizar publicando en los nodos. Además, existe la posibilidad de recibir la información de los distintos sensores del Erle-Copter a través de la suscripción a los nodos que se deseen.

A continuación, se presentará un segundo método que utiliza las mismas bases de comunicación entre los ordenadores, es decir, se comunican haciendo uso del switch y de MAVROS, pero en este caso, en vez de utilizar MAVROS desde el terminal, se hará desde una GCS.

6.3. Comunicación usando QGroundControl

Ya se presentó anteriormente la estación de control de tierra o GCS llamada QGroundControl. Como se sabe, se trata de un programa mediante el cual se puede controlar un drone a través de una interfaz gráfica como la que se muestra en la figura 6.2.

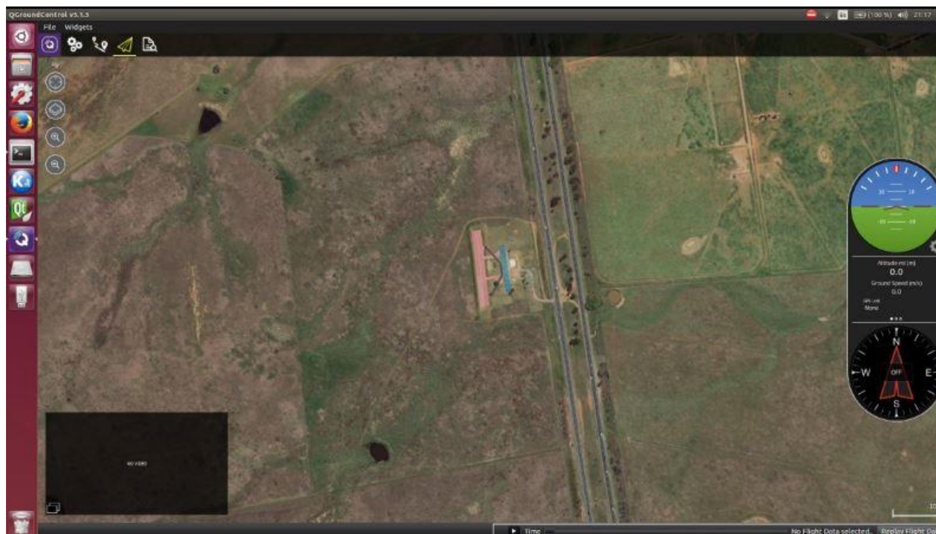

Figura 6.2 Interfaz gráfica de la aplicación QGroundControl

En la esquina superior izquierda se encuentran las diferentes herramientas de esta aplicación. En especial, al pulsar sobre el icono de QGroundControl se pasará a una ventana desde la cual se pueden cambiar todas las configuraciones del UAV, además de poder calibrar sus sensores, su emisora de radio-telemetría, etc. Es importante saber que mientras que no haya ningún un dispositivo conectado, dicha configuración no aparecerá en la pantalla.

En el caso de este trabajo, lo que se ha de enlazar no es el QGroundControl con un hardware, sino con la simulación que se ha creado, pero esto no resulta ningún problema, ya que esta aplicación permite el uso del protocolo MAVLink bajo UDP, de manera que se puede conectar la simulación tratando la misma como si de un UAV real se tratase.

Para indicar al programa el dispositivo con el que se quiere establecer la conexión, es necesario modificar, en el PCUSER, que es donde se ha lanzado el QGroundControl, el puerto de escucha de UDP y añadir la dirección IP del PCDRONE tal y como aparece en la figura 6.3. Esta configuración se encuentra en la pestaña de *Comm Links*.

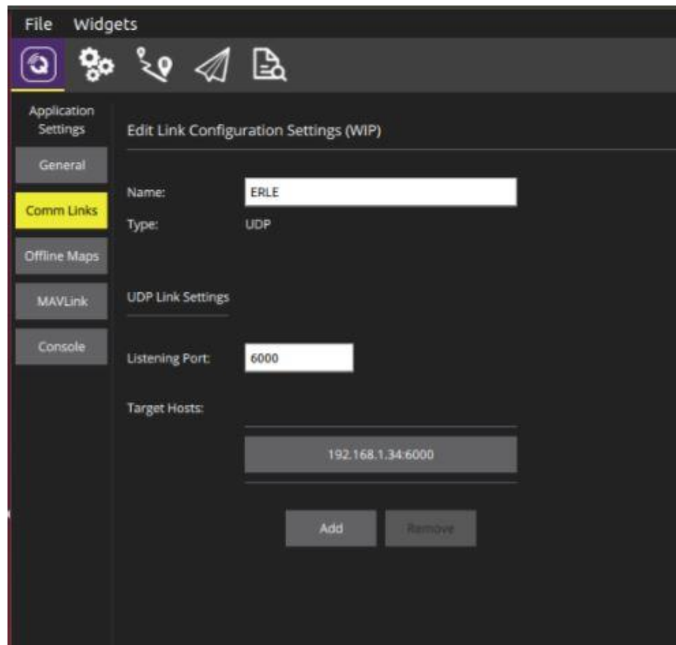

Figura 6.3 Configuración de comunicaciones de QGroundControl

No basta solo con esto para que la simulación del Erle-Copter reciba todas aquellas ordenes que se le den. Es requerida la creación de un terminal MAVProxy en el PCDRONE que utilice también el puerto que se le ha asignado a UDP en el QGroundControl. Para conseguir esto, se utiliza el comando 6.3, en cual se deben indicar la IP del PCUSER y el mismo puerto que el utilizado en la configuración del QGroundControl. Un ejemplo de este comando es el que aparece en la figura 6.4, que se correspondería con la configuración aportada en la figura 6.3.

```
mavproxy.py --master=127.0.0.1:14551 --out <IP PCUSER>:<PUERTO UTILIZADO>
```

Comando 6.3 Comando para la creación de un terminal MAVProxy

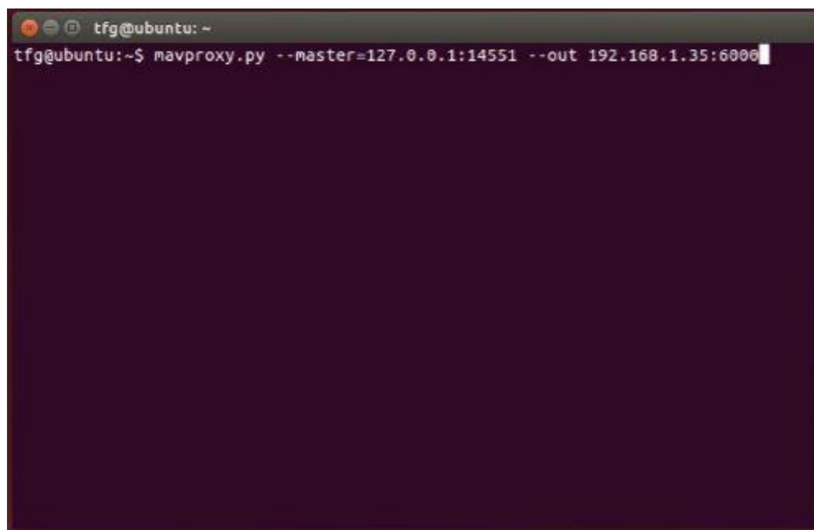

Figura 6.4 Creación de un terminal MAVProxy

El procedimiento a seguir para probar esto comenzaría con el lanzamiento de la simulación del Erle-Copter junto con el MAVProxy en el PCDRONE y, una vez cargado todo correctamente, se debe conectar el QGroundControl. Tras esto la conexión se habrá establecido y se podrá crear una misión de vuelo, definiendo las diferentes ordenes que debe cumplir el drone y que este las ejecute una tras otra, o bien indicándole al mismo paso a paso lo que realizar, lo cual se correspondería con el envío de comandos por terminal.

En la figura 6.5 se muestra a la izquierda la sucesión de posiciones a seguir que se le ha enviado desde QGroundControl, y a la derecha aparece el drone en la última de dichas posiciones. Antes de realizar todo esto, como es de esperar, es necesario armar y despegar el drone, utilizando para ello los pulsadores que aparecen en la parte inferior de la pantalla para cada uno de los casos.

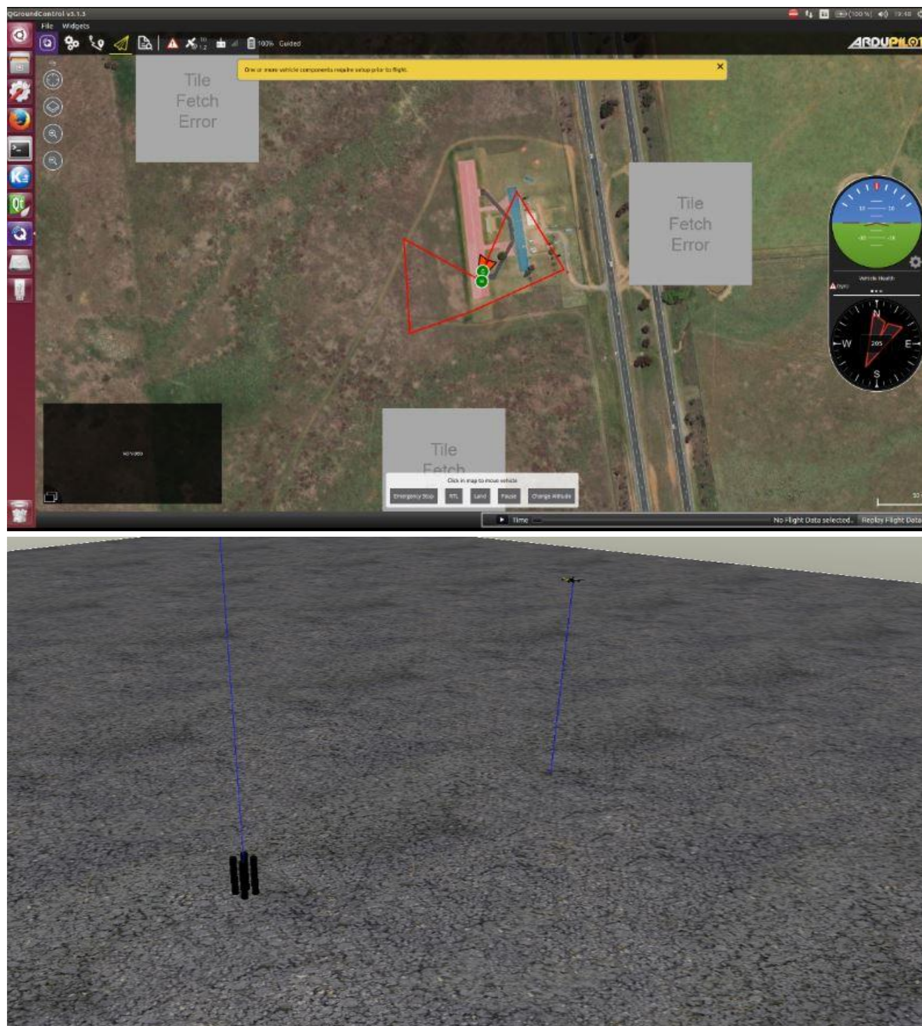

Figura 6.5 Sucesión de puntos en QGroundControl (arriba) y simulación (abajo)

7 Interfaz de Control Propia

Tras haber establecido la conexión y haber probado la misma según los dos métodos que se han presentado, es importante disponer de una plataforma basada en dicha comunicación que permita un control total de la simulación, en la cual el usuario tenga la posibilidad de implementar diversos bucles de control y aplicárselos al drone, mediante la que se pueda, también, llevar a cabo el control de la carga con el fin de que esta no se balancee constantemente, o simplemente, que permita aunar todos los conceptos vistos hasta ahora sobre los diferentes comandos aplicables en la simulación.

Esta interfaz ha sido desarrollada por Jesús Lozano Rodriguez, persona que ha permitido tanto su uso en este proyecto como la modificación de la misma para conseguir los objetivos que se persiguen.

7.1. Componentes del programa

La interfaz utilizada se trata de un programa en C++ compuesto por diferentes funciones e hilos (*threads*), donde cada uno de ellos lleva a cabo una función determinada. A continuación, se presenta una visión general de estos componentes, pero sin extenderse demasiado, ya que no es el objetivo de este documento llevar a cabo la descripción uno a uno de cada de los distintos elementos de este código y, además, porque sólo algunos de ellos han sido objeto de modificación durante el desarrollo del proyecto.

Los componentes del programa son los siguientes:

- **Carpeta CONTROL:** aquí se encuentran, como es de esperar, todos los archivos que influyen de alguna manera en los procedimientos de control del drone. Dentro de la misma se incluye el hilo de control, mediante el cual, como se expondrá en el apartado 8.1, aparecen las sucesiones de comandos que se le envían al drone con el fin de que lleve a cabo diferentes rutinas de vuelo. Además, es en dicho directorio donde se deben incluir todos aquellos controladores que se utilicen en el código.
- **Carpeta CVISION:** en esta carpeta están los códigos encargados de determinados procesos de visualización en la interfaz, los hilos principales se encargan de representar las imágenes captadas por las cámaras del Erle-Copter y, sobre ellas dibujar por ejemplo ciertas líneas de referencia para la carga, o, en el caso de que se ordene el detectar esta, dibujar una señal sobre la pelota. Esta detección esta implementada con un filtro de Kalman mediante el cual se consigue seguir la posición de la carga en cada momento, a través de la estimación de la misma para cada instante, calculando el error frente a la posición real y actualizando la posición del objetivo situado sobre la carga en función de este. Hay dos hilos de procesamiento, uno por cada una de las cámaras.

  Además de dichos hilos, también hay archivos con los que se consigue crear ventanas mediante las cuales ajustar los valores de detección de la carga o de la trayectoria, y archivos para mostrar los valores de la cámara o avisos de armado del drone.
- **Carpeta GUI:** incluye el hilo para iniciar y actualizar la interfaz gráfica o GUI (Graphical User Interface).
- **Carpeta PLOTS:** contiene las funciones encargadas de graficar los valores de roll, pitch y yaw tanto de la carga como del drone, además de los valores que adquiere cada uno de los canales del Erle-Copter. Cabe mencionar que, aunque dichas gráficas están disponibles, a la hora de graficar diferentes valores en este proyecto se ha preferido utilizar un método diferente que se explicará en el apartado 8.2.
- **Carpeta ROS:** aquí se encuentran todas las funciones que llevan a cabo labores relacionadas con ROS, es decir, realizan automáticamente muchas de las tareas que se hacían en apartados anteriores de forma manual, como el uso de los comandos necesarios las obtenciones de las imágenes o las conexiones vía MAVROS. Además, hay un hilo en el que se indican todas las suscripciones y publicaciones que se deben realizar sobre determinados topics de ROS, con el fin de sobreescribir dichos valores de la simualción o adquirir la información que se publica en ellos, como, por ejemplo, valores de sensores del drone.

- **Archivo *main.cpp*:** este programa es el encargado de ir ejecutando uno a uno los diferentes hilos del programa.
- **Archivo *mainwindow.cpp*:** programa encargado de incializar y configuración de todos los subsistemas y de sus variables, como pueden ser los valores booleanos que reciben las señales de activación de los pulsadores que posee la interfaz, el color que se detectará a través de la cámara para la carga y para la trayectoria.

  Asi mismo, se establecen los parámetros del drone tras lanzar la simulación, tal y como se hacía en con los comandos 3.6 y 4.1, además de otros muchos parámetros que se desea que tengan un valor específico durante la simulación.

  A continuación, se incluyen las instrucciones necesarias para atender a los cambios del GUI con el fin de actualizar las señales que se le envíen, como el pulsado de los botones o la actualización de las imágenes, de los sensores como la IMU, de los datos de la carga y del drone, o de los datos de MAVLink.

  Finalmente, se establecen las relaciones necesarias para realizar las tareas correctas ante la señalización de que se ejecuten las mismas a través de la interfaz gráfica.
- **Archivo *shared_memory.cpp*:** este último archivo se trata de un código encargado de organizar la memoria compartida de todo el programa mediante la apertura y el cierre de diferentes mutex.

Cada uno de estos archivos se encuentra acompañado de otro programa con el mismo nombre y extensión *.h* en el cual se definen variables y constantes que serán utilizados en los archivos de extensión *.cpp* correspondientes.

Se incluye a continuación, a modo de resumen, una tabla con la organización de dichos programas.

Tabla 7.1 Programas de la interfaz propia

<table>
<tr><td rowspan="15">…/src</td><td rowspan="2">CONTROL</td><td>pid.cpp</td></tr>
<tr><td>thread_processing_control.cpp</td></tr>
<tr><td rowspan="7">CVISION</td><td>Dialog_avisoARMED.cpp</td></tr>
<tr><td>Dialog_DetectarObjeto.cpp</td></tr>
<tr><td>Dialog_DetectarTrayectoria.cpp</td></tr>
<tr><td>Dialog_valorraspicam.cpp</td></tr>
<tr><td>kalman.cpp</td></tr>
<tr><td>thread_processing_cvision1.cpp</td></tr>
<tr><td>thread_processing_cvision2.cpp</td></tr>
<tr><td>GUI</td><td>thread_processing_gui.cpp</td></tr>
<tr><td rowspan="4">PLOTS</td><td>Dialog_Resultados_Angulos_Carga.cpp</td></tr>
<tr><td>Dialog_Resultados_Angulos_Drone.cpp</td></tr>
<tr><td>Dialog_Resultados_Canales_RC.cpp</td></tr>
<tr><td>qcustomplot.cpp</td></tr>
</table>

	ROS	*ros_image.cpp*
		ros_mavros_imu.cpp
		ros_mavros_set.cpp
		ros_raspicam_set.cpp
		subscribe_mavros_state.cpp
		thread_processing_ros.cpp
	main.cpp	
	mainwindow.cpp	
	shared_memory.cpp	

7.2. Interfaz gráfica

Como se ha explicado anteriormente, al ejecutar el código anterior, lo que se genera es una interfaz gráfica que actúa como centro de mando o estación de control de tierra y, desde la cual, se puede manejar tanto un drone real como, en este caso, la simulación. Su apariencia es la de la figura 7.1.

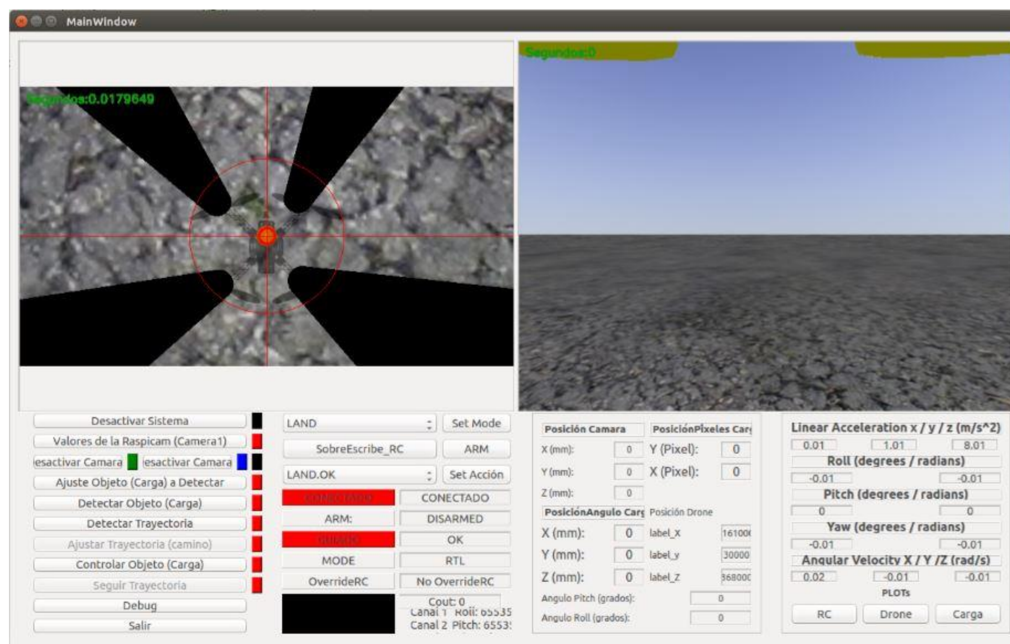

Figura 7.1 Apariencia de la interfaz gráfica propia

En la imagen se puede ver como, una vez lanzada la simuación, se pueden visualizar las imágenes de las cámaras en la parte superior.

Una vez que se ha lanzado una simulación, se debe pulsar el botón de *Activar Sistema*, hasta que comienza a parpadear el indicador negro inferior, mostrando que el autopiloto se encuentra conectado desde ese momento. Tras esto se pueden activar ambas cámaras en *Activar Cámara 1* y *Activar Cámara 2*, donde la cámara uno es

la inferior y la dos es la frontal. Los distintos parámetros utilizados para la cámara inferior se pueden visualizar con el botón *Valores de la Raspicam (Camara 1)*.

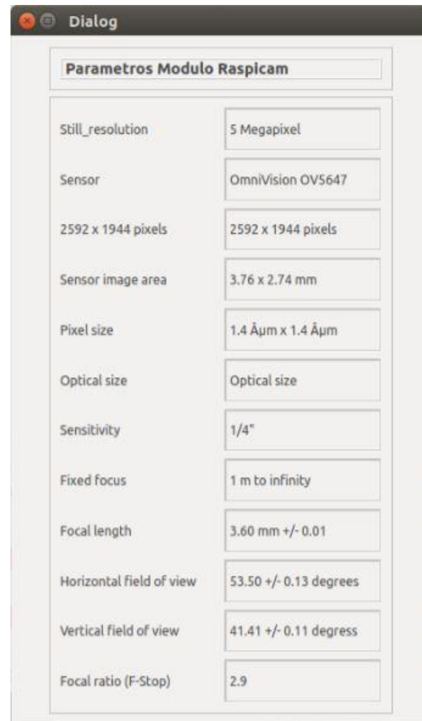

Figura 7.2 Parámetros raspicam

Al pulsar *Detectar Objeto (Carga)* comenzará dicho proceso y aparecerá una señal verde sobre la carga constantemente, y será cuando, una vez se haya implementado el controlador, se podrá pulsar *Controlar Objeto (Carga)* para comenzar dicha tarea. Sabiendo que la detección de la carga se lleva a cabo en función del color de la misma, con el botón *Ajuste Objeto (Carga) a Controlar* se podrá modificar el mismo en el caso de que se desee utilizar una tonalidad diferente. En este caso, se habrá de configurar dicha tonalidad mediante su código de colores, con ayuda de la ventana emergente.

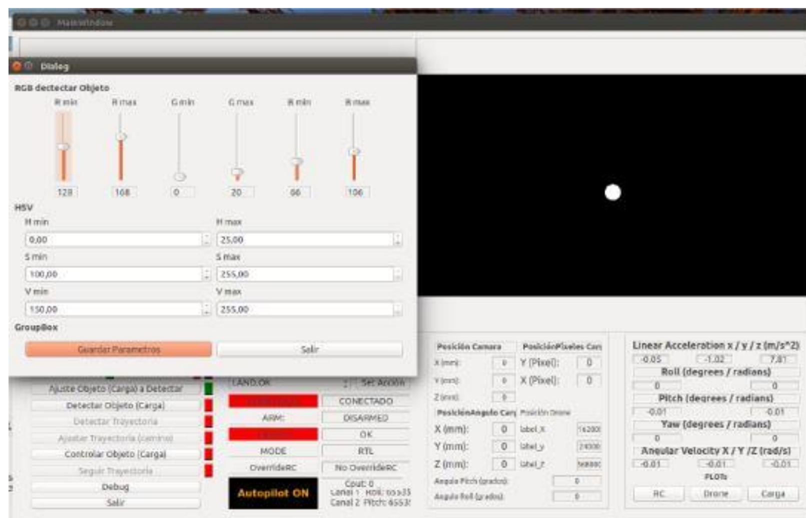

Figura 7.3 Ajuste Objeto (Carga) a Controlar

Además de esto, en el siguiente bloque de la interfaz se pueden apreciar dos pestañas diferentes. La primera de ellas sirve para indicar un modo de vuelo (*GUIDED*, *LOITER*, *LAND*, *ALT HOLD,* etc), el cual se establece pulsando el botón *Set Mode*, siempre y cuando la simulación se esté ejecutando. La segunda pestaña tiene un funcionamiento similar, pero en su caso se establecen misiones de vuelo, es decir, distintos modos de vuelo encadenados entre los cuales, por ejemplo, se puede realizar un control de la carga, o cualquier cosa que el usuario desee. También se dispone de un botón para armar el drone (*ARM*).

Lo siguiente que se puede ver es un gran número de indicadores, entre los que se incluyen, entre otros, el estado de la conexión, el modo de vuelo actual, los valores de los distintos canales, la posición de la cámara en milímetros y píxeles o las velocidades, aceleraciones y ángulos del drone.

Finalmente, en la esquina inferior derecha hay tres botones mediante los que se pueden graficar los canales, los ángulos del drone y los ángulos de la carga.

Figura 7.4 Gráfica de la carga obtenida por la interfaz

8 CONTROL

Una vez presentada la interfaz que se va a utilizar para los posteriores experimentos, se llega a este último capítulo. En el se explicarán las modificaciones del código del capítulo anterior que han sido necesarias para conseguir los objetivos que se plantean a continuación.

En este momento, se hará una simulación en la que se le aplicarán tres velocidades de referencia diferentes al Erle-Copter tanto en el eje x como en el eje y, y se analizarán los resultados de la misma. Además, se va a explicar el proceso mediante el cual se consigue conocer la posición de la carga respecto a determinados puntos de referencia especificados por el usuario con el fin de, a partir de ahí, obtener el error en posición de la misma y actuar al respecto utilizando para ello un controlador con el que intentar estabilizar la pelota.

8.1. Configuración del código del programa

Tal y como se introdujo en el capítulo 7, sobre el código adquirido se han realizado ciertas modificaciones con el fin de utilizar el mismo para fines específicos, la mayoría de los cuales se han basado en el código *thread_processing_control.cpp*.

Este archivo diferencia entre distintas rutinas o misiones de vuelo en función de aquella que se le indica desde el GUI creado. Esta distinción se implementa mediante bucles *if* dentro de los cuales se le adjudica un caso a cada una de las rutinas. De este modo, mediante un bucle *switch* se ejecutarán las acciones predefinidas dependiendo del caso que se haya indicado. Por ejemplo, hay un caso para despegar, otro para aterrizar, un tercero en el que se despega se espera un determinado tiempo y se aterriza, etc.

En este proyecto, se ha creado un caso en especifico en el cual, con la ayuda del uso de un contador, se realizan las siguientes tareas:

- Al iniciarse se establece el modo *ALT HOLD* y se ajustan los valores de cada uno de los canales a 1500, que es el valor en el que todos ellos se encuentran en reposo.
- A continuación, se cambia el modo a LOITER, se arma el Erle-Coper y se sobrescribe el canal del Throttle, introduciendo el valor 1600. Debido a esto, el drone comenzará a realizar un movimiento en el eje z positivo a una velocidad constante y no parará hasta el momento en que dicho valor vuelva a establecerse en 1500. Como el lector podrá percibir, este procedimiento es en realidad lo que se ejecuta mediante los comandos 3.8.
- Lo siguiente que se hace es sobreescribir el canal del Pitch siguiendo la serie 1500-1550-1600 en intervalos de 500 iteraciones. Estas iteraciones son las que se miden gracias al contador. Una vez que se realiza este proceso, se vuelve a establecer el valor del Pitch en 1500.
- Después de esto se realiza lo mismo, pero en este caso se sobrescribe el canal correspondiente al Roll, siguiendo el mismo patrón de valores en los mismos intervalos.
- Finalmente se aterriza el drone.

Se incluye a continuación el fragmento con el que se implementa dicho proceso.

```
case 10: //PARAMETRIZAR_DRONE

  if (accionContador == 1360)
   {
     ////////////////MODE////////////////////
     MAVROS_setAccion.MAVROS_setMode(0, "LOITER");
     share_memory->setOverride(true); // Señal de control indica si se habilita el
sobrescribir las canales de la emisora desde el PC.
     share_memory->setRoll(BASERC);
     share_memory->setPitch(BASERC);
     share_memory->setThrottle(BASERC);
     share_memory->setYaw(BASERC);}
  if (accionContador == 1360)
   {
     //////////////////ARM////////////////////
     MAVROS_setAccion.MAVROS_setCommandBool(true);
   }

  if (accionContador == 1320)
   {
     ///////////////MODE////////////////////
     MAVROS_setAccion.MAVROS_setMode(0, "LOITER");
     ////////////////TAKEOFF///////////////////
   }
  if (accionContador == 1280)
   {
     share_memory->setRoll(BASERC);
     share_memory->setPitch(BASERC);
     share_memory->setThrottle(1600);
     share_memory->setYaw(BASERC);
   }
  if (accionContador == 1240)
   {
     share_memory->setRoll(BASERC);
     share_memory->setPitch(BASERC);
     share_memory->setThrottle(BASERC);
     share_memory->setYaw(BASERC);
   }
  //Pitch
  if (accionContador == 1200)
   {
     share_memory->setRoll(BASERC);
     share_memory->setPitch(1550);
     share_memory->setThrottle(BASERC);
     share_memory->setYaw(BASERC);
   }
  if (accionContador == 950)
   {
     share_memory->setRoll(BASERC);
     share_memory->setPitch(1600);
     share_memory->setThrottle(BASERC);
     share_memory->setYaw(BASERC);
   }
  if (accionContador == 700)
   {
     share_memory->setRoll(BASERC);
     share_memory->setPitch(BASERC);
     share_memory->setThrottle(BASERC);
     share_memory->setYaw(BASERC);
   }
  //Roll
  if (accionContador == 620)
   {
     share_memory->setRoll(1550);
     share_memory->setPitch(BASERC);
     share_memory->setThrottle(BASERC);
     share_memory->setYaw(BASERC);
   }
```

```
 if (accionContador == 370)
  {
    share_memory->setRoll(1600);
    share_memory->setPitch(BASERC);
    share_memory->setThrottle(BASERC);
    share_memory->setYaw(BASERC);
  }

 if (accionContador == 120)
  {
    share_memory->setRoll(BASERC);
    share_memory->setPitch(BASERC);
    share_memory->setThrottle(BASERC);
    share_memory->setYaw(BASERC);
  }
 if (accionContador == 80)
  {
    MAVROS_set MAVROS_setMode(share_memory->getNodeHandle(),share_memory);
    MAVROS_setMode.MAVROS_setMode(0,"LAND");
    accionContador = accionContador-1;
    share_memory->setAccionContador(accionContador);
  }
 if (accionContador == 0)
  {
    //acciÃ³n terminada
    caso = 0;
    share_memory->setAccionContador(0);
    share_memory->setAccion("");
    share_memory->setOverride(false); // SeÃ±al de control indica si se habilita el
sobrescribir las canales de la emisora desde el PC.
    break;
   }
 accionContador = accionContador-1;
 //qDebug() << accionContador << endl;
 share_memory->setAccionContador(accionContador);
 break;
```

Código 8.1 Caso 10 del código *thread_processing_control.cpp*

Además de esto, se ha creado otro caso, el número 21, que actúa cuando se selecciona el botón *Controlar Objeto (Carga)*, y que se encarga de calcular la posición de la carga y sobrescribir los canales de Pitch y Roll en consecuencia, con el fin de intentar situar la pelota en el centro de la imagen. De todos modos, se estudiará su funcionamiento en el apartado 8.3.

8.2. Modo LOITER

Para el estudio que se lleva a cabo en este proyecto se hará uso del modo *LOITER* del drone. Este modo permite el control del drone mediante la aportación de velocidades de referencia lineales, es decir, en los ejes x, y, y z por cada uno de los canales. En el caso de desplazarse en los ejes x e y, el vehículo mantendrá los grados necesarios para adquirir dicha velocidad, además de la altitud que poseía antes de comenzar a moverse.

En el caso en que las velocidades de referencia sean nulas, se mantendrá la posición del Erle-Copter. Esto sería equivalente, en el caso del drone físico, a aplicarle al drone velocidades de referencia lineales mediante el movimiento de los sticks de la emisora de radio, con la condición de que, si se sueltan los mismos, el drone mantenga la posición que posea en ese mismo momento.

Para ello, según el modelo de Erle Robotics, son utilizados distintos lazos de control, entre los que se incluyen, por ejemplo, un controlador de posición y otro de velocidad. Además, el modo *LOITER* hace uso de los algoritmos de control utilizados en el modo *STABILIZE*, los cuales no serán estudiados en este proyecto, ya que se debe tratar el sistema del drone y la carga como una caja negra, y obtener a partir de experimentos realizados sobre el mismo la información deseada.

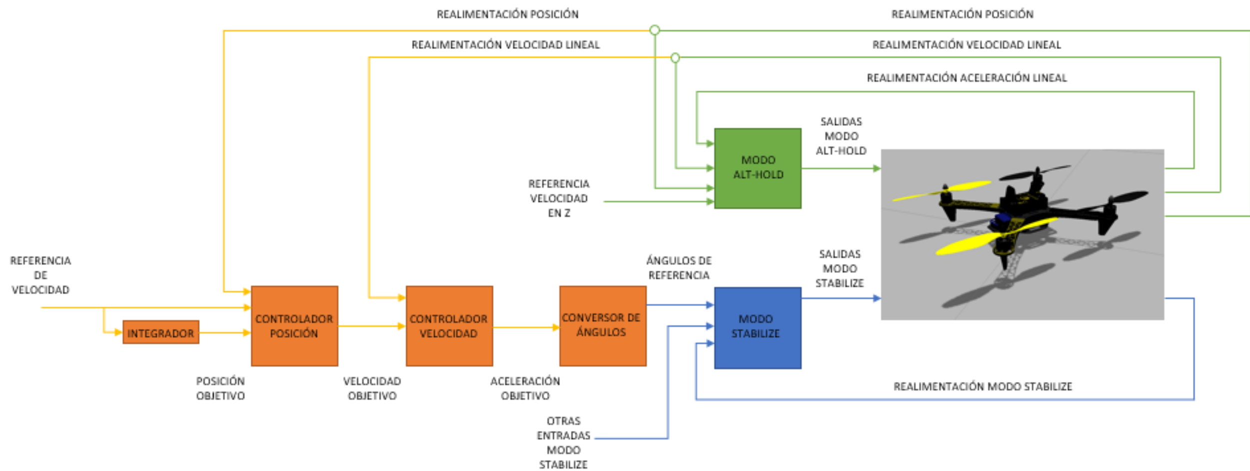

Figura 8.1 Esquema de funcionamiento del modo LOITER

8.2.1 Estudio sobre el Pitch y el Roll

En el apartado 8.1 se presentó el código implementado para realizar un experimento en modo *LOITER* variando para ello las velocidades de referencia del Erle-Copter en las direcciones del eje x e y. Estas velocidades de referencia se corresponden con un escalado que se les realiza a los valores introducidos en los canales.

El experimento, en resumen, consiste en, una vez situado el drone a una altura deseada, aplicar distintos escalones a dichas velocidades de referencia en las direcciones x e y. Una vez realizada dicha tarea, el drone aterrizará y se desarmará.

Una vez desarmado, la herramienta MAVROS ofrece la posibilidad de descargar un archivo *log* que recoge todos los datos de los sensores y actuadores del drone obtenidos durante una sesión de vuelo. Si se desean ver los archivos *logs* disponibles en el autopiloto se debe ejecutar el comando 8.1, mientras que, si se desea descargar uno de esos archivos en específico, basta con ejecutar el comando 8.2.

```
log list
```

Comando 8.1 Comando para mostrar la lista de logs en el APM

```
log download [numero_del_log] [nombre_tras_descarga]
```

Comando 8.2 Comando para descargar un log específico

Es importante recordar que solo se permite mostrar o descargar dichos archivos cuando el drone esta desarmado.

El archivo que se descarga tendrá una extensión del tipo *<nombre>.bin*, y se guardará en la siguiente ubicación *~/simulation/ardupilot/ArduCopter*. Una vez localizado, se puede utilizar otra herramienta de MAVROS con la cual se pasa de un archivo *.bin* a un archivo *.m*, el cual es compatible con el programa MATLAB. El comando utiizado para ello es el comando 8.4. Para ejecutar el mismo se ha de estar en el driectorio donde se encuentra el archivo a modificar.

```
mavtomfile.py <NOMBRE>.bin
```

Comando 8.3 Comando para convertir un archivo *.bin* a *.m*

Finalmente, mediante el uso de MATLAB, se pueden graficar aquellos valores del vuelo realizado con el Erle-Copter que se prefieran.

8.2.2 Conclusiones de los experimentos

Tras realizar la simulación programada, obtener el archivo *.bin* de la misma y graficar los datos deseados en MATLAB, los resultados son los siguientes.

Comentar antes de nada que, aunque únicamente se ha realizado un solo experimento, este se ha tratado como si se tratase de rutinas aisladas. Es por esto por lo que en los primeros segundos de los resultados que se muestran a continuación pueden existir ciertas discrepancias entre las medidas reales y las deseadas, ya que en realidad estos primeros valores se corresponden con las perturbaciones procedentes de otros procesos anteriores.

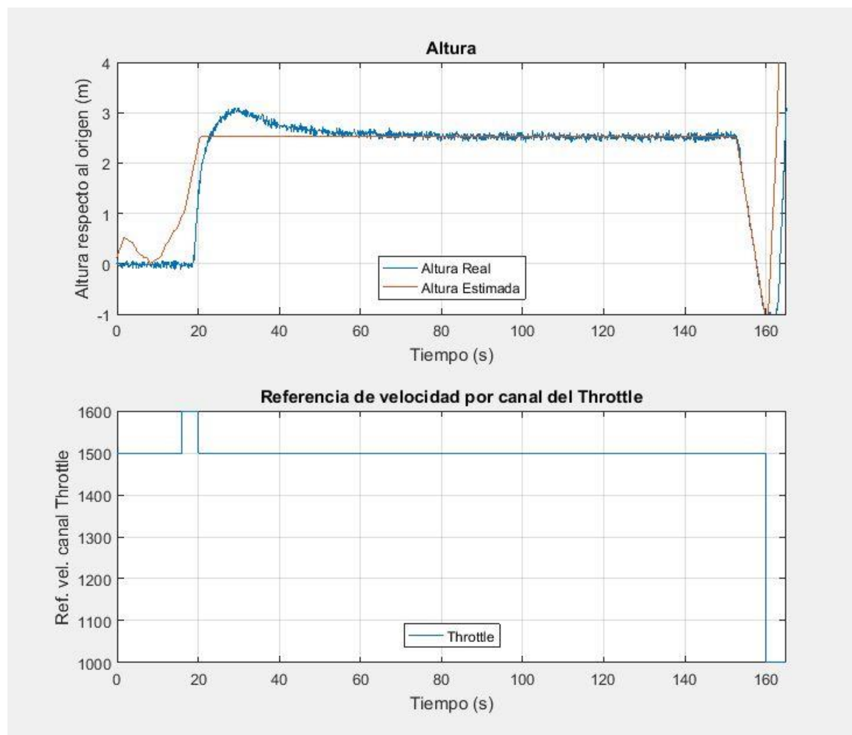

Figura 8.2 Estudio del movimiento en el eje *z*

Como se puede percibir en la gráfica 8.2, en los primeros 20 segundos aparece una perturbación debida al armado de los motores. La posición en el modo *LOITER* se rige por las medidas del GPS y del barómetro, las cuales se establecen como iniciales en el momento en el que se inicia la simulación. Debido a esto, se puede observar como la posición inicial del drone es de cero metros, cuando en realidad se encuentra a un metro del suelo por estar encima del soporte. Si en vez de graficar las medidas del GPS se representan las del LIDAR, el resultado sería el mismo, pero partiendo desde 1 metro de altura.

Se puede observar que, en el momento en que se varía la velocidad de referencia de 1500 (reposo) a 1600, el Erle-Copter comienza a ascender hasta que no se restablece dicho valor de nuevo a 1500. Como se puede observar, la altura real del drone sufre una sobreoscilación de un 20 % aproximadamente, para después alcanzar a la referencia correctamente. El tiempo de subida es de aproximadamente 3.8 segundos, y el tiempo de establecimiento de unos 30 segundos. Este valor en el tiempo de establecimiento se debe a que el drone, debido al empuje que tiene al estar sobre el soporte, sufre una pequeña traslación horizontal a la vez que asciende, y después intenta regresar al punto de origen y a la altura requerida a la vez.

Durante aproximadamente 100 segundos, en los cuales el drone se mantiene a una altura de 2.5 metros, se van realizando los desplazamientos horizontales detallados en el apartado 8.1. Se aprecia, por tanto, que aún siendo el valor de referencia introducido por el canal del Throttle el de reposo, el controlador de altura actúa correctamente manteniendo la misma. Finalmente, se le cambia el modo a *LAND*, por lo que el drone aterriza en un tiempo aproximado de 8 segundos. La variación de altura que se inicia al final de la gráfica se corresponde

con el modo *RTL* (Return To Load), en el cual el drone sube hasta los diez metros de altura y vuelve a la posición desde la que partió. Dicho método no será objeto de estudio en este proyecto, por lo que se ha optado por suprimir su comportamiento en los resultados.

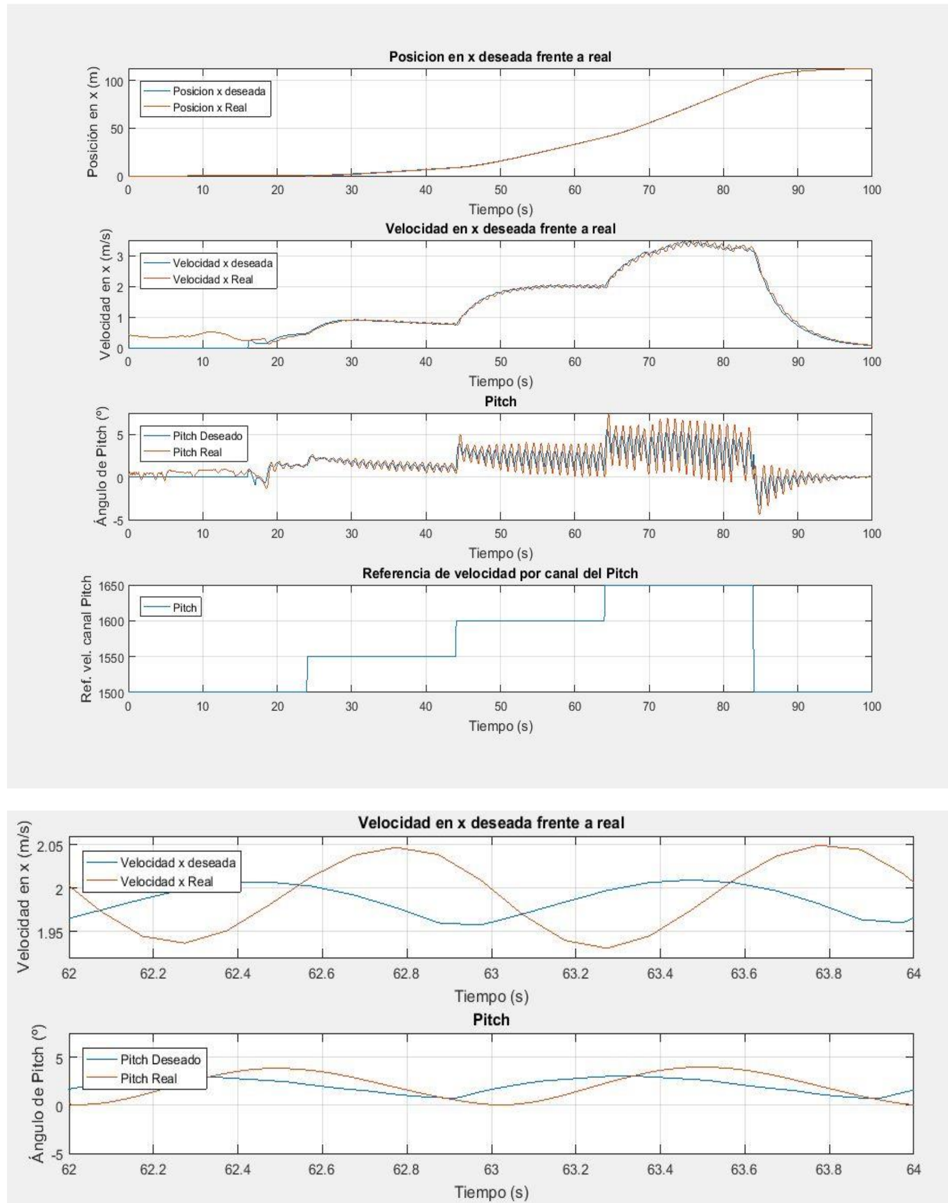

Figura 8.3 Estudio del movimiento en el eje *x*

En la figura 8.3 se muestra el estudio del movimiento del drone en el eje *x*, correspondiente a las variaciones de referencia por el canal del Pitch. Las dos gráficas inferiores se corresponden con ampliaciones realizadas sobre la segunda y la tercera gráfica respectivamente, con el fin de facilitar la visualización de los valores tomados. Reiterando en lo comentado anteriormente, las perturbaciones iniciales se deben a que, en este caso, el proceso previo al movimiento en *x* se correspondió con la variación de altura de la gráfica anterior.

En estas gráficas se puede apreciar que, ante un aumento de la referencia en velocidad por el canal del Pitch, la posición en el eje *x* aumenta de manera lineal y siguiendo a la posición objetivo continuamente, por lo que se comprueba el buen funcionamiento del controlador de posición. En lo referente a la velocidad, esta también sigue a la deseada en todo momento, pudiéndose observar mejor en la quinta gráfiEs importante no confundir la referencia en velocidad, representada en la cuarta gráfica, con la velocidad deseada, representada en la segunda y en la quinta gráfica en color azul. La primera hace referencia con el valor introducido por el canal correspondiente, el cual esta comprendido en un rango de 1100 a 1900. Dicha referencia de velocidad es traducida o escalada a una cierta velocidad objetivo, que es la que el drone debe mantener en todo momento, y

la cual recibe el nombre de velocidad deseada.

La velocidad deseada posee unos tiempos de subida de 6.4 segundos para el primer escalón, 8.8 segundos para el segundo y 9.2 segundos para el tercero, aproximadamente, alcanzando respectivamente para cada salto velocidades en torno a los 0.9 m/s, 2 m/s y 3.4 m/s. El primer tiempo de subida es relativamente menor que el resto debido a que en el proceso de ascensión previo ya se alcanza un cierto valor de velocidad en el eje x debido a esa traslación que sufre el drone al elevarse, que se comentó en el caso anterior.

En la tercera gráfica cuya ampliación entre los 62 y 64 segundos se corresponde con la última de ellas, aparece representado el ángulo de Pitch que adquiere el Erle-Copter con el fin de alcanzar y mantener la velocidad deseada, el cual, como es lógico, aumenta frente al incremento de dicha velocidad, comprobándose así el correcto funcionamiento del controlador de ángulo. Las oscilaciones que aparecen se pueden deber a un solapamiento entre las restricciones del controlador de altitud o posición y las del controlador de velocidad, ya que, mientras este último intenta variar el ángulo para alcanzar la velocidad deseada, el primero de los controladores mencionados intenta, a su vez, variar dicho ángulo para mantener constante la altitud, lo que provoca estas oscilaciones que, en el mayor de los casos, llega a ser de unos 5 grados aproximadamente. Aunque se encuentran dentro de un rango de valores normal, se podrían disminuir las mismas a través de un mejor ajuste de los controladores o mediante la aplicación de filtros.

Finalmente, se puede apreciar como al devolver el valor de referencia de velocidad al reposo (1500), se mantiene la posición actual del drone, anulando el giro en Pitch y, por tanto, su velocidad. De esta forma es más sencillo comprender el comportamiento del modo *LOITER* explicado al comienzo del apartado 8.2.

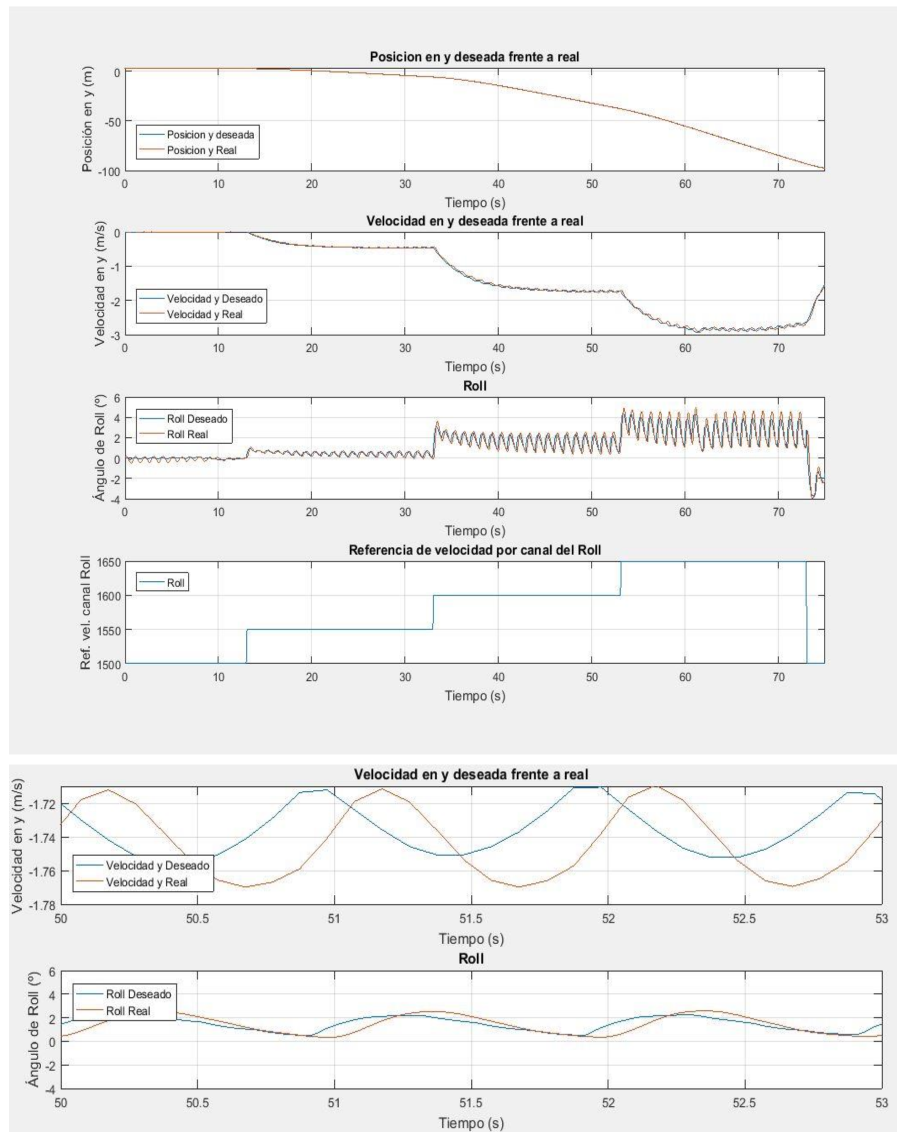

Figura 8.4 Estudio del movimiento en el eje *y*

Los resultados obtenidos para el movimiento en el eje *y* son muy parecidos a los del caso anterior. Comentar que, ante un incremento de la referencia en velocidad por el canal del Roll, el drone se desplaza en el sentido negativo del eje *y*, por lo que la posición referente al origen del sistema de referencia tomado decrece, tomando valores negativos. Debido a esto, el valor de la velocidad también aparecerá como negativo, lo que se traduce en una velocidad positiva en el sentido negativo del eje en cuestión. Con esto se consigue deducir que ante incrementos en la referencia se produce un giro en Roll positivo, como se aprecia en la gráfica tercera, y viceversa.

Se observa como tanto en posición como en velocidad y ángulo se sigue de nuevo correctamente a la referencia. Además, se vuelven a producir las oscilaciones comentadas anteriormente en este caso para el ángulo de Roll, adquiriendo de nuevo unos valores máximos de aproximadamente 5 grados.

Por último, especificar que, en términos de velocidad, los tiempos de subida para cada escalón son de 8.7 segundos, 8.8 segundos y 7.5 segundos, alcanzando tras las mismas velocidades en torno a los 0.45 m/s, 1.73 m/s y 2.9 m/s.

8.3. Detección de la carga

Con el objetivo de llevar a cabo un control de la carga, y ya que se dispone del método de detección de la carga, tal y como se puede ver en la figura 8.4, es necesario conocer la localización de la misma respecto a su posición de estabilidad, que se corresponde con el centro de la imagen que se presenta en la interfaz gráfica.

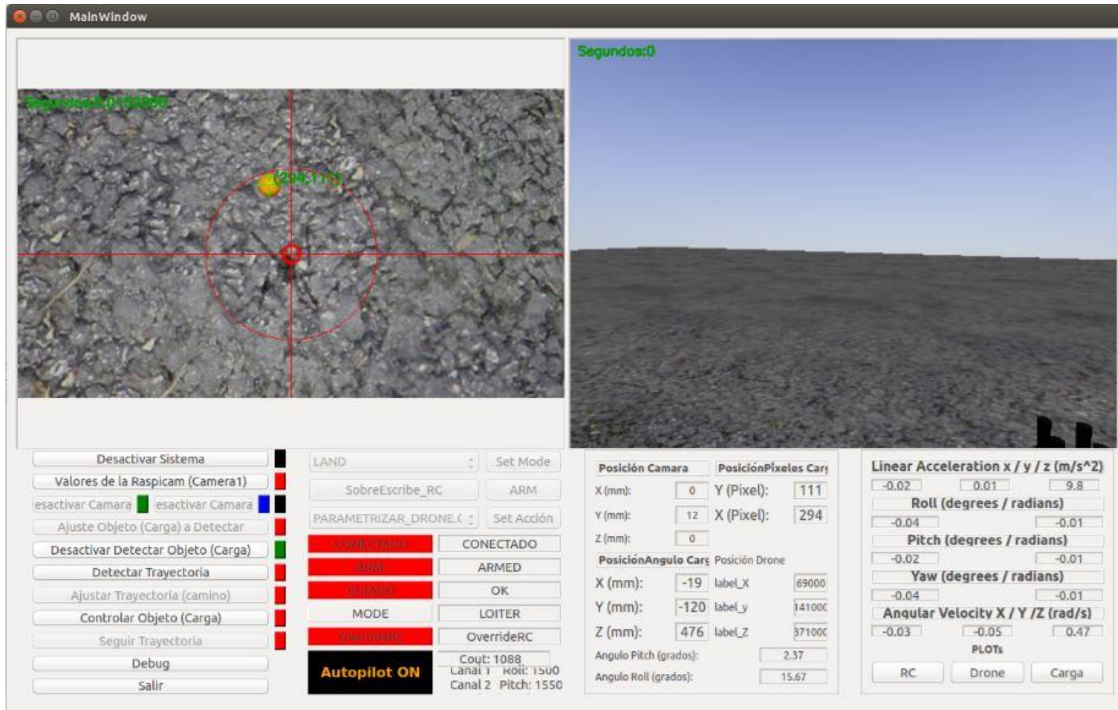

Figura 8.5 Interfaz gráfica con carga detectada

Dentro del caso 21 del archivo *thread_processing_control.cpp* se encuentra definido todo el procedimiento para llevar a cabo dicha detección además del control de la pelota. Se presenta en el código 8.2 un fragmento de dicho caso.

```
case 21: //TakeOFF1m_CONTROL_CARGA_LAND
  if  (share_memory->getControlarObjeto())
   {
      ...

      // Get the Image center
      ImageX = InImage.cols / 2;
      ImageY = InImage.rows / 2;
      // Detect Object
      ObjectoX = share_memory->getPuntoXImg();
      ObjectoY = share_memory->getPuntoYImg();
      // Move the origin to the center of the image
      Xval = ObjectoX - ImageX;
      Yval = -(ObjectoY - ImageY);
      // Change from cartesian coordinates to polar coordinates
      Radius = sqrt((Xval)^2+(Yval)^2); // Distance from the object to the origin
      Theta = atan2(Yval,Xval); // Angle of the object relative to the origin
      if(Theta > 0 && Theta < M_PI_2) // Cuadrante superior derecha
       {
         if(Radius <= 10) // Circulo pequeño
          {
            // Quieto
            max_roll = BASERC;
            // Quieto
            max_pitch = BASERC;
          } else if( Radius > 10 && Radius <= 100) // Circulo grande
          {
            // Hacia derecha
            max_roll = 1525;
            // Hacia adelante
            max_pitch = 1475;
          } else if(Radius > 100) // Fuera del Circulo grande
          {
            max_roll = 1550;
            // Hacia adelante
            max_pitch = 1450;
          }
       } else if(Theta > M_PI_2 && Theta < M_PI) // Cuadrante superior izquierda
       {
         if(Radius <= 10) // Circulo pequeño
          {
            // Quieto
            max_roll = BASERC;
            // Quieto
            max_pitch = BASERC;
          } else if( Radius > 10 && Radius <= 100) // Circulo grande
          {
            // Hacia izquierda
            max_roll = 1475;
            // Hacia adelante
            max_pitch = 1475;
          } else if(Radius > 100) // Fuera del Circulo grande
          {
            // Hacia izquierda
            max_roll = 1450;
            // Hacia adelante
            max_pitch = 1450;
          }
       }

     ...
      <CUADRANTES INFERIORES>
     ...

       // Calculate Roll and Pitch depending on the mode
       if (mode_control == "LOITER")
        {
          auto_Roll = max_roll;
          auto_Pitch = max_pitch;
```

```
    } else {
      auto_Roll = BASERC;
      auto_Pitch = BASERC;
    }
    // Limit the Roll
    if (auto_Roll > MAXRC)
     {
       auto_Roll = MAXRC;
     } else if (auto_Roll < MINRC)
     {
       auto_Roll = MINRC;
     }
    // Limit the Pitch
    if (auto_Pitch > MAXRC)
     {
       auto_Pitch = MAXRC;
     } else if (auto_Pitch < MINRC)
     {
       auto_Pitch = MINRC;
     }

            ...

      share_memory->setPitch(auto_Roll);
      share_memory->setRoll(auto_Pitch);
      share_memory->setThrottle(BASERC);
     }
     break;
  }
```

Código 8.2 Fragmento del caso 21 del código *thread_processing_control.cpp*

Lo primero que se realiza es la toma de la imagen y la definición de su centro. Se ha predefinido que dicha imagen será de 640x360, por lo que el centro se localiza en la posición (320,180), partiendo del punto (0,0) que se encuentra en la esquina superior izquierda. A continuación, se obtiene la posición de la carga, y se calcula el error de dicha posición respecto al centro de la imagen.

La imagen, tal y como se muestra en la figura 7.2, se divide en diferentes regiones con ayuda de unas marcas realizadas sobre la misma. Concretamente, se distinguen los cuatro cuadrantes y dos circunferencias que dividen cada cuadrante en tres regiones diferentes. Dentro de cada cuadrante, por tanto, se distinguirán tres posibles estados de la carga, que son, centrada, en cuyo caso se encontraría dentro del círculo de menor radio, a media distancia, si está posicionada entre el círculo pequeño y el grande, o lejos, si se encuentra fuera de ambos círculos.

Dependiendo de esta distancia y del cuadrante en el que este situada, el drone deberá llevar a cabo una alteración del valor el canal de Pitch y Roll de manera correcta con el objetivo de posicionar la carga en el centro.

Ahora bien, la detección de la posición de la carga se puede implementar de dos maneras diferentes. Una de ellas sería comprobando si los errores en x e y adquieren valores negativos o positivos, de modo que asi se identificaría el cuadrante, y a continuación, comprobar el valor absoluto de dichos errores para calcular la distancia que separa a la carga del centro, deduciendo así en que región se encuentra. La otra posibilidad es mediante el uso de coordenadas polares, que es lo implementado en el código 8.2. Lo primero que habría que hacer es trasladar el origen de la esquina superior izquierda al centro de la imagen y, a continuación, realizar la conversión de coordenadas cartesianas a polares. Tras esto, mediante el ángulo theta se identificaría el cuadrante mientras que con la ayuda del radio se calcularía la región donde esta situada la carga.

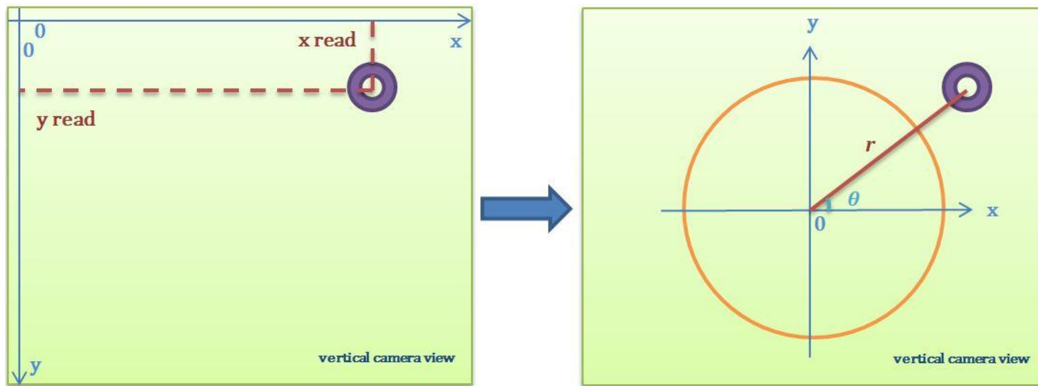

Figura 8.6 Posición de la carga en cartesianas (izquierda) y en polares (derecha) tomado de [19]

Entonces, como se ve en el código 8.2, mediante bucles se consigue realizar esta distinción de regiones, y dentro de cada caso específico se sobrescriben los canales de Pitch y de Roll con valores determinados. El movimiento del drone se rige de la siguiente manera:

- Canal Pitch > 1500: desplazamiento hacia atrás.
- Canal Pitch < 1500: desplazamiento hacia adelante.
- Canal Roll > 1500: desplazamiento hacia la derecha.
- Canal Roll < 1500: desplazamiento hacia la izquierda.

Entonces dependiendo de la posición de la carga, se aplicarán movimientos de la siguiente manera con el fin de centrar la carga:

- Carga en cuadrante superior izquierdo: movimiento hacia adelante y hacia la izquierda.
- Carga en cuadrante superior derecho: movimiento hacia adelante y hacia la derecha.
- Carga en cuadrante inferior izquierdo: movimiento hacia atrás y hacia la izquierda.
- Carga en cuadrante inferior derecho: movimiento hacia atrás y hacia la derecha.

Los valores aplicados en los correspondientes canales dependerán de la distancia a la que esté la pelota del centro, de este modo:

- Carga en círculo pequeño: 1500.
- Carga entre el círculo pequeño y el grande: escalón de 25 hacia arriba o hacia abajo partiendo de 1500.
- Carga fuera de ambos círculos: escalón de 75 hacia arriba o hacia abajo partiendo de 1500.

Este método se trataría de un control todo o nada, que es bastante impreciso. Una forma de mejorarlo sería la implementación de un control proporcional, con el cual, dependiendo del error para cada uno de los ejes, se le aplicaría una referencia de velocidad específica al canal del pitch o al del roll, en función también de la ganancia proporcional que se le aplique al controlador. El código 8.3 es el implementado para ello.

En el se puede apreciar como para el caso del error en el eje x se lleva a cabo un control proporcional para el roll, limitando la actuación a un mínimo de 1525, y de igual manera se hace para el caso del error en el eje y.

Una vez establecidos los límites entre los que trabajaría la ganancia proporcional mediante la ecuación 8.1, se ajustó la misma hasta obtener un comportamiento relativamente correcto.

$$K_P = \frac{\Delta U}{e} \tag{8.1}$$

Aún así, el resultado obtenido tras la ejecución de dicho programa sobre la simulación sigue sin ser el mejor de

todos. Esto se debe a que un control proporcional no consigue anular los errores en régimen permanente, por lo que la carga permanecerá oscilando, eso si, de manera menos agresiva, tras el control de la misma.

```
case 21: //TakeOFF1m_CONTROL_CARGA_LAND
   if  (share_memory->getControlarObjeto())
    {
       ...

       // Get the Image center
       ImageX = InImage.cols / 2;
       ImageY = InImage.rows / 2;
       // Detect Object
       ObjectoX = share_memory->getPuntoXImg();
       ObjectoY = share_memory->getPuntoYImg();
       // Calculate the error between Image center and Objecto center
       ErX = ObjectoX - ImageX;
       ErY = ObjectoY - ImageY;

       Kp = 0.45;
       if (abs(ErX) <= 10)
         Roll = BASERC;
       else
         {
           PROPX = Kp*ErX;
           if(abs(PROPX) < 25)
             PROPX = 25;
         Roll = PROPX + 1500;
         }
       if(abs(ErY) <= 10)
         Pitch = BASERC;
       else
         {
           PROPY = Kp*ErY;
           if(abs(PROPY) < 25)
             PROPY = 25;
           Pitch = PROPY + 1500;
         }

       // Calculate Roll and Pitch depending on the mode
       if (mode_control == "LOITER")
        {
          auto_Roll = Roll;
          auto_Pitch = Pitch;
        } else {
          auto_Roll = BASERC;
          auto_Pitch = BASERC;
        }
        // Limit the Roll
        if (auto_Roll > MAXRC)
         {
           auto_Roll = MAXRC;
         } else if (auto_Roll < MINRC)
         {
           auto_Roll = MINRC;
         }
        // Limit the Pitch
        if (auto_Pitch > MAXRC)
         {
           auto_Pitch = MAXRC;
         } else if (auto_Pitch < MINRC)
         {
           auto_Pitch = MINRC;
         }

                ...

                share_memory->setPitch(auto_Roll);
                share_memory->setRoll(auto_Pitch);
                share_memory->setThrottle(BASERC);
         }
       break;
    }
```

Código 8.3 Control proporcional de la carga

8.4. PID para control de carga

Tras el estudio anterior, se podría implementar un controlador PID para estabilizar la carga, para lo cual se tendría que realizar algún experimento.

En este caso, se tendría una caja negra basada en el sistema del drone más la carga suspendida, cuya entrada sería la referencia de velocidad por el canal del roll o por el canal del pitch, y la salida sería la posición en x o en y, respectivamente, de la carga respecto al centro de la imagen.

Debido a esto, el experimento podría consistir en la aplicación de un escalón en la referencia de velocidad del canal del pitch, esperando a continuación a que la carga se estabilice en un punto de la imagen. El comportamiento de la carga debería ser similar al de un sistema de segundo orden subamortiguado, por lo que graficando los valores tomados por la misma en el eje y (para el caso del pitch) y estudiando dicha gráfica, se podría obtener la función de transferencia del comportamiento de la carga y, a partir de esta, obtener los parámetros de un controlador PID adecuado para el control de la misma.

A continuación, se realizaría el mismo experimento para el caso del roll.

Ya que el comportamiento de la carga no es ideal, es probable que una entrada en escalón no se traduzca en un desplazamiento de la carga únicamente en el eje correspondiente, por lo que, si el resultado del experimento anterior no es del todo acertado, se podría intentar realizar el mismo, pero ante una entrada impulsional.

Una vez obtenidos los parámetros que definen un controlador PID, es decir, las ganancias proporcional, integral y derivativa (Kp, Ki y Kd), se procedería a aportar sus valores al código PID implementado.

Los estudios de estos comportamientos y sus resultados serán objeto de investigaciones posteriores a las que en este proyecto se abordan.

9 CONCLUSIÓN

Finalmente se ha conseguido disponer de un mundo virtual en el que aparece el Erle-Copter junto con una carga suspendida de el a 60 cm de distancia, y con un comportamiento dinámico muy similar al que adquiere el sistema real. Además, se ha conseguido convertir el ordenador en el que dicho entorno se encuentra montado en una máquina independiende de aquel desde el cual el usuario controla la simulación. De este modo, se utiliza dicha simulación como si se del drone real se tratase, con la ventaja de poder probar diferentes ejercicios sin correr el riesgo de que se estrelle y pueda dañarse algún componente.

Por último, se ha introducido el proyecto en el ámbito del control de la carga, realizando estudios del comportamiento de los controladores que utiliza el modo *LOITER* ofrecido por el autopiloto que se encuentra instalado, estableciendo un procedimiento mediante el cual se lleva a cabo un reconocimiento de la carga y una deducción de su posición respecto a su posición de estabilidad e indagando en la forma mediante la que se podrían establecer los parámetros de entrada de un supuesto controlador PID que se implementase con el objetivo de estabilizar la pelota.

En resumen, se piensa haber alcanzado satisfactoriamente los objetivos principales del proyecto, los cuales se trataban del modelado del entorno virtual, y haber ampliado dichos objetivos con la comunicación, los estudios de comportamiento y las bases para las actividades de control a realizar. Para todos estos objetivos se ha necesitado poner en práctica muchos conceptos aprendidos durante estos cuatro años, pero, además, se considera haber ampliado considerablemente el conocimiento en el ámbito del mundo de los UAVs.

Las futuras ampliaciones y líneas de investigación haciendo uso del sistema aportado podrían basarse en mejoras en el modelado de la simulación, haciendo la misma más precisa y realista, retoques sobre los controladores del Erle-Copter con el fin, por ejemplo, de eliminar esas oscilaciones que se producían en los ángulos de pitch y roll, la implementación de controladores para la carga, tanto para mantener estable la misma como para conseguir que se mantenga en un punto determinado mientras se describe una determinada trayectoria o utilizar el mismo sistema de reconocimiento de la carga para calcular el punto donde se debe depositar la carga, siendo dicho punto objetivo una plataforma fija o móvil, como por ejemplo, un vehículo terrestre.

ÍNDICE DE TABLAS

Tabla 2.1 Características comerciales del Erle-Copter 3
Tabla 4.1 Elemento <*gazebo*> para la etiqueta <*robot*> 19
Tabla 4.2 Elemento <*gazebo*> para la etiqueta <*link*> 20
Tabla 4.3 Elemento <*gazebo*> para la etiqueta <*joint*> 20
Tabla 7.1 Programas de la interfaz propia 48

ÍNDICE DE FIGURAS

Figura 1.1 Drone con cámara incorporada 1
Figura 1.2 Drone destinado a salvamento 1
Figura 1.3 Drone de la empresa Amazon para reparto de paquetes 2
Figura 2.1 Erle-Copter 3
Figura 2.2 Erle Brain 2 4
Figura 2.3 Logotipo de Ardupilot 4
Figura 2.4 Logotipo de ROS 5
Figura 2.5 Esquema simplificado del funcionamiento de ROS 6
Figura 2.6 Logotipo de Gazebo 7
Figura 3.1 Esquema del entorno instalado 10
Figura 3.2 Esquema del drone simulado 11
Figura 3.3 Terminales con los comandos para lanzar la simulación 12
Figura 3.4 Inicio de la simulación 12
Figura 3.5 Erle-Copter volando 13
Figura 3.6 Simulación de Erle-Copter con cámaras activadas 14
Figura 4.1 Optical Flow de Pixhawk 15
Figura 4.2 Autopiloto PX4 de Pixhawk 16
Figura 4.3 Drone Iris 16
Figura 4.4 Drone con optical flow delante (izquierda) vs drone con optical flow detrás (derecha) 26
Figura 4.5 Despegue del Erle-Copter con medidas del optical flow y el LIDAR 27
Figura 4.6 Traslación en horizontal del Erle-Copter con medidas del optical flow y el LIDAR 27
Figura 5.1 Posiciones de las patas de la plataforma 31
Figura 5.2 Modelo de la plataforma 32
Figura 5.3 Colisiones iniciales del Erle-Copter 32
Figura 5.4 Erle-Copter sobre plataforma 33
Figura 5.5 Drone con carga suspendida tomado de [20] 36
Figura 5.6 Erle-Copter con soporte y carga suspendida 37

Figura 5.7 Simulación finalizada 40
Figura 5.8 Erle-Copter volando con el modelado del entorno finalizado 40
Figura 6.1 Ping entre el PCUSER (izquierda) y el PCDRONE (derecha) 42
Figura 6.2 Interfaz gráfica de la aplicación QGroundControl 43
Figura 6.3 Configuración de comunicaciones de QGroundControl 44
Figura 6.4 Creación de un terminal MAVProxy 44
Figura 6.5 Sucesión de puntos en QGroundControl (arriba) y simulación (abajo) 45
Figura 7.1 Apariencia de la interfaz gráfica propia 49
Figura 7.2 Parámetros raspicam 50
Figura 7.3 Ajuste Objeto (Carga) a Controlar 50
Figura 7.4 Gráfica de la carga obtenida por la interfaz 51
Figura 8.1 Esquema de funcionamiento del modo LOITER 56
Figura 8.2 Estudio del movimiento en el eje z 57
Figura 8.3 Estudio del movimiento en el eje x 58
Figura 8.4 Estudio del movimiento en el eje y 60
Figura 8.5 Interfaz gráfica con carga detectada 61
Figura 8.6 Posición de la carga en cartesianas (izquierda) y en polares (derecha) tomado de [19] 64

ÍNDICE DE COMANDOS

Comando 3.1 Comandos para la instalación de las dependencias del ROS Base sacado de [3] 9
Comando 3.2 Comando para solucionar error 1 9
Comando 3.3 Último comando de la instalación de mensajes de comunicación entre ROS y Gazebo 10
Comando 3.4 Comandos para lanzar MAVProxy 11
Comando 3.5 Comandos para lanzar la simulación 12
Comando 3.6 Comandos para cargar los parámetros de Erle-Copter 12
Comando 3.7 Comandos para armar y despegar el drone en modo GUIDED 13
Comando 3.8 Comandos para armar y despegar el drone en modo LOITER 13
Comando 3.9 Comandos para establecer el parámetro *ARMING_CHECK* a cero 13
Comando 3.10 Comandos para visualizar las cámaras del Erle-Copter 14
Comando 4.1 Comandos para activación del optical flow 26
Comando 4.2 Comandos para la comprobación de los valores de los parámetros 26
Comando 4.3 Comandos para graficar valores del optical flow y de altura 26
Comando 6.1 Comando para conocer las características de red de un ordenadoren Ubuntu 41
Comando 6.2 Comando para comprobar la conexión entre dos ordenadores en Ubuntu 41
Comando 6.3 Comando para la creación de un terminal MAVProxy 44
Comando 8.1 Comando para mostrar la lista de logs en el APM 56
Comando 8.2 Comando para descargar un log específico 56
Comando 8.3 Comando para convertir un archivo *.bin* a *.m* 56

ÍNDICE DE CÓDIGOS

Código 4.1 Ejemplo de código de mundo en SDF 17

Código 4.2 Ejemplo de código de modelo en SDF 17

Código 4.3 Ejemplo de modelo de luz en SDF 18

Código 4.4 Ejemplo de atributos dentro de un modelo de mundo en SDF 18

Código 4.5 Ejemplo de atributos dentro de un modelo en SDF 19

Código 4.6 Archivo *lidar_sensor.urdf.xacro*: Primeras líneas 22

Código 4.7 Archivo *lidar_sensor.urdf.xacro*: Joint y link 23

Código 4.8 Archivo *lidar_sensor.urdf.xacro*: Sensor y plugins 24

Código 4.9 Archivo *generic_camera.urdf.xacro*: Sensor 25

Código 4.10 Archivo *erlecopter.xacro*: Inclusión de los modelos del LIDAR y el optical flow 25

Código 5.1 Fragmento del código del soporte 31

Código 5.2 Fragmento del código *legs_bases.urdf.xacro* 33

Código 5.3 Fragmeto del código *charge.urdf.xacro* destinado al modelado físico de la carga 36

Código 5.4 Fragmeto del código *charge.urdf.xacro* destinado al comportamiento dinámico de la carga 37

Código 5.5 Modificaciones en el código *empty.world* 38

Código 5.6 Modificaciones en el código *erlecopter.xacro* 39

Código 5.7 Modificación del código *erlecopter_spawn.launch* 39

Código 6.1 Modificación del archivo *.bashrc* en el PCDRONE 42

Código 6.2 Modificación del archivo *.bashrc* en el PCUSER 42

Código 8.1 Caso 10 del código *thread_processing_control.cpp* 55

Código 8.2 Fragmento del caso 21 del código *thread_processing_control.cpp* 63

Código 8.3 Control proporcional de la carga 65

BIBLIOGRAFÍA

[1] Docs de Erle Robotics. Disponible en http://docs.erlerobotics.com/

[2] Foro de Erle Robotics. Disponible en http://forum.erlerobotics.com/

[3] Instalación del entorno del Erle-Copter.
Disponible en http://docs.erlerobotics.com/simulation/configuring_your_environment

[4] Herramientas de MAVROS. Disponible en https://erlerobotics.gitbooks.io/erle-robotics-mav-tools-free/content/en/

[5] Modos de vuelo Erle-Copter. Disponible en https://erlerobotics.gitbooks.io/erle-robotics-erle-copter/content/es/flight_modes/index.html

[6] Página official Ardupilot. Disponible en http://ardupilot.org/ardupilot/

[7] Parámetros de Ardupilot. Disponible en http://ardupilot.org/copter/docs/parameters.html

[8] Guía PX4. Disponible en https://www.gitbook.com/book/px4/firmware-devguide/details

[9] Wiki de ROS. Disponible en http://wiki.ros.org/

[10] Foro de ROS. Disponible en https://discourse.ros.org/

[11] Solución al problema de instalación. Disponible en http://answers.ros.org/question/141151/building-ros_control-on-hydro/

[12] Formato URDF. Disponible en http://wiki.ros.org/urdf

[13] Formato SDF. Disponible en http://sdformat.org/spec

[14] Tutoriales de Gazebo. Disponible en http://gazebosim.org/tutorials

[15] Foro de Gazebo. Disponible en http://answers.gazebosim.org/questions/

[16] Guía MAVROS. Disponible en http://wiki.ros.org/mavros

[17] Guía MAVLink. Disponible en http://www.qgroundcontrol.org/mavlink/start

[18] Guía del usuario QGroundControl. Disponible en https://donlakeflyer.gitbooks.io/qgroundcontrol-user-guide/content/en/

[19] Localización de la carga. Disponible en http://www.ludep.com/drone-new-pid-with-polar-coordinates-and-howto-improve-reactivity-and-accuracy/

[20] Ivana Palunko, Rafael Fierro, and Patricio Cruz, *Trajectory generation for swing-free maneuvers of a quadrotor with suspended payload: A dynamic programming approach*, Robotics and Automation (ICRA), 2012 IEEE International Conference on.

[21] Franz Bahner, Modeling, *Simulation and control of a quadcopter carrying a slung load*

[22] Octavio Alfredo García Campos, *Modelado, simulación y control de un quadrotor para transporte de carga colgante. Extensión al caso tridimensional,* Proyecto de Fin de Carrera, Dep. Ingeniería de Sistemas y Automática. Escuela Técnica Superior de Ingeniería. Universidad de Sevilla, 2017.