Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Modelado y simulación de dron Erle-Copter con ROS/Gazebo

Autor: Miguel Ángel Guijarro Martínez

Tutores: Felipe Espinosa Zapata y Miguel Martínez Rey

2018

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Modelado y simulación de dron Erle-Copter con ROS/Gazebo

Autor: Miguel Ángel Guijarro Martínez

Directores: Felipe Espinosa Zapata y Miguel Martínez Rey

Tribunal:

Presidente: Elena López Guillén

Vocal 1º: Ernesto Martín Gorostiza

Vocal 2º: Felipe Espinosa Zapata

Calificación: .....

Fecha: .....

"Un viaje de mil millas comienza con un primer paso."
Lao-Tsé

Agradecimientos

En primer lugar, me gustaría agradecer a Felipe Espinosa, Miguel Martínez y Carlos Santos, por su gran ayuda siempre que ha sido necesaria y su guía durante este proyecto, además de facilitar todos los recursos necesarios durante la realización del mismo.

Gracias a mi familia, por el cariño y el apoyo incondicional que sé que siempre he tenido.

Gracias a Esther, por ser uno de los pilares fundamentales de este proyecto y de mi vida.

Finalmente, me gustaría agradecer también a mis amigos, por estar siempre ahí y hacer más llevaderos todos los momentos complicados.

Gracias, porque sin todos vosotros nada de esto hubiera sido posible.

Resumen

El mundo de los drones está revolucionando la actualidad debido a la gran cantidad de ámbitos en los que pueden implementarse. Han transformado tareas que podían resultar complejas con otro tipo de vehículos en algo sencillo, pudiendo realizarlas con eficiencia y precisión.

Este Trabajo Fin de Grado se enfoca en el dron *Erle-Copter*, en su modelo y los controladores que utiliza durante el vuelo. Lo simularemos en un entorno virtual, en el cual realizaremos pruebas con distintas trayectorias y parámetros, para finalmente analizar los datos post-vuelo.

Las herramientas principales para lograr la simulación han sido *ROS* y *Gazebo*, además de herramientas adicionales como *APMPlanner 2*, así como su enlace con *MATLAB* para el tratamiento de datos.

Palabras clave: *ROS, Gazebo*, modelo y simulación de *Erle-Copter*, análisis post-vuelo.

Abstract

Drones world is revolutionizing the present due to the great quantity of fields in which they can be implemented. They have transformed tasks that could be difficult with other types of vehicles in something really simple, being able to realize them with efficiency and precision.

This TFG focuses in the drone *Erle-Copter*, his model and controllers used during flight. We will simulate it in a virtual environment, in which we will realize some tests with different trajectories and parameters, to finally analize the post flight data.

The main tools needed to achieve the simulation have been *ROS* and *Gazebo*, in addition to some other tools such as *APMPlanner 2*, as well as the link with *MATLAB* to perform data analysis.

Keywords: *ROS, Gazebo, Erle-Copter* simulation and model, post flight analysis.

Índice general

Índice de figuras

Capítulo 1

Introducción

En la actualidad, el mundo relacionado con los vehículos aéreos no tripulados, también conocidos como drones o *UAVs*, ha cobrado una gran importancia debido a la enorme cantidad de aplicaciones en las que pueden ser de utilidad. Sus primeras apariciones fueron destinadas a las operaciones militares y de seguridad, aunque hoy en día son cada vez más usuales en supervisión del estado del tráfico, tareas de mantenimiento en el sector eléctrico y ferroviario, prevención y control de incendios, búsqueda de personas desaparecidas, en el ámbito de fotografía y vídeo, etc. Dentro del concepto de *UAV* podemos encontrar distintas familias, según la configuración del propio dron. En este proyecto, nos centraremos en los multirrotores, que son los que poseen tres o más rotores, y más concretamente en los quadrotor de tipo X. Podemos ver un ejemplo de esta configuración en la figura 1.1.

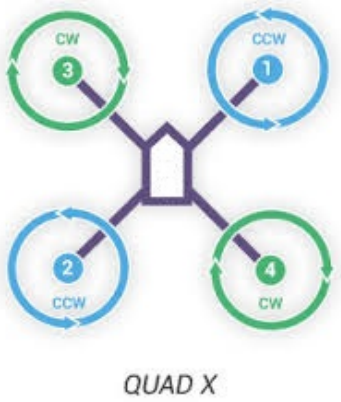


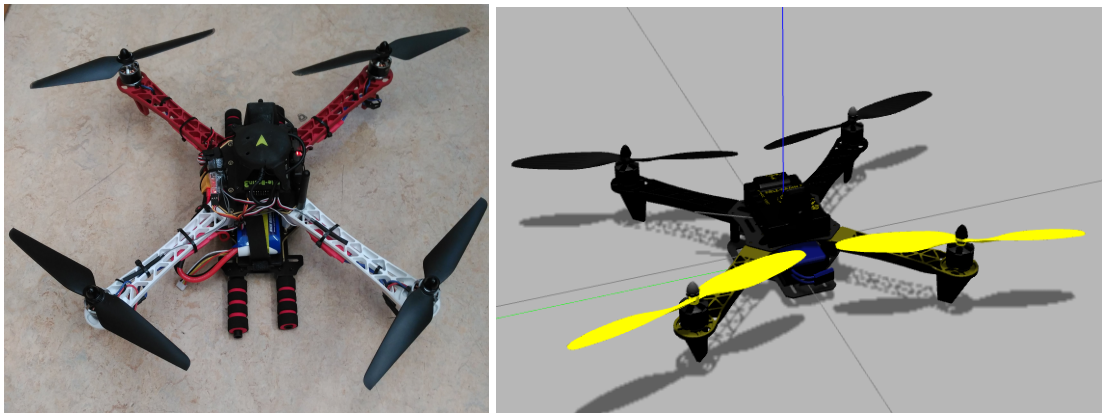
Figura 1.1: Configuración de quadrotor tipo X

Como podemos observar, estos drones poseen cuatro rotores, cuyas hélices son instaladas en sentidos de rotación opuestos de forma diametral, es decir, se alternan hélices de giro horario con hélices de giro anti-horario, resultando nula la suma de las fuerzas que generan. Si todos los rotores producen la misma fuerza de sustentación, el aparato se mantendrá en vuelo estacionario. Si por el contrario uno de los rotores presenta mayor o menor velocidad angular que el resto, se producirá el balanceo del aparato. Los quadrotor se han desarrollado enormemente en las últimas décadas, debido en gran parte a su diseño relativamente simple, pero siendo a su vez muy estables y fáciles de manejar.

El dron sobre el cual se enmarca este proyecto es precisamente del tipo mencionado en esta introducción. Su nombre es *Erle-Copter*[1], desarrollado por la compañía *Erle Robotics*[2]. Podemos ver el prototipo en la figura 1.2a.

El objetivo principal de este proyecto será simular el modelo del dron que proporciona *Erle Robotics*, en el entorno de simulación *Gazebo*. Además lo enlazaremos con la herramienta *ROS(Robot Operating System)* y veremos las posibilidades que nos ofrece para poder sacarle el máximo partido a las simulaciones. A partir de estas, podremos analizar los resultados del vuelo y estudiar su comportamiento ante diferentes circunstancias y parámetros.

Se conseguirá así un sistema de simulación completo y sus consiguientes pruebas que permitirán futuros estudios en diferentes líneas de trabajo dentro del Departamento de Electrónica de la UAH.



(a) Dron real (b) Modelo del dron simulado en Gazebo

Figura 1.2: Prototipos de Erle-Copter real y simulado

Capítulo 2

Herramientas utilizadas

En este capítulo se presenta de forma general el ya mencionado *Erle-Copter*. Es importante conocerlo ya que en la simulación llevada a cabo aparece una réplica del mismo. Así mismo, también se realiza una descripción detallada de cada una de las herramientas software utilizadas en el proyecto.

2.1 Erle-Copter

Se trata de un dron inteligente basado en el sistema operativo *Linux*, con soporte oficial para marcos robóticos como *ROS*. Su elemento más importante es el *Erle-Brain 3* [3], módulo electrónico que actúa como un pequeño ordenador Linux, el cual tiene *ROS* preinstalado. Posee todo el software y sensores necesarios para convertir al dron en un dispositivo autónomo. Proporciona una Unidad de Control de Vuelo (*FCU* o *Flight Control Unit*), encargada de aportar controles básicos de vuelo, también denominado autopiloto, el cual se tratará más adelante. Además de esto, tiene integrados una serie de sensores, tales como una IMU (acelerómetros y giroscopios), GPS, brújula y barómetro, los cuales le permiten tener una estimación de la posición y orientación.

Como ya se ha comentado, mediante herramientas de simulación podremos analizar el comportamiento del autopiloto y de los sensores que posee el *Erle-Brain 3* como si del dispositivo real se tratase.

Figura 2.1: Erle-Brain 3

2.2 Software

2.2.1 Autopiloto: ArduPilot

Un autopiloto es un software que permite controlar la trayectoria de un vehículo (en nuestro caso, el vuelo del dron) sin la necesidad del control manual constante de una persona.

Para el *Erle-Copter*, utilizaremos *ArduPilot* [4]. Se ha elegido este autopiloto dado que es de código abierto, lo cual hace que esté en constante desarrollo, y además es totalmente compatible con el *Erle-Brain 3*. *ArduPilot* posee distintos *firmwares* o códigos a elegir en función del vehículo que se use.

En nuestro caso, un quadrotor, el *firmware* que tenemos que usar será *Copter*, o también denominado *ArduCopter* [5].

Figura 2.2: Logotipo Arducopter

Posee una amplia variedad de modos de vuelo, los cuales analizaremos más adelante, entre los que encontramos algunos manuales y otros totalmente autónomos, que permiten hacer misiones preprogramadas sin intervención humana. Además, es capaz de conseguir que el dron mantenga la posición deseada en pleno vuelo gracias a los datos del GPS, acelerómetros y barómetro. Otra característica fundamental es que es totalmente flexible y personalizable, ya que el usuario tiene acceso a una gran cantidad de parámetros que permiten contolar el comportamiento del vehículo según la necesidad.

En este proyecto utilizaremos la versión 3.4 de *Copter*, que es una versión completamente estable del software. Para poder llegar a simularlo en nuestro ordenador, nos ayudaremos de la aplicación SITL, la cual se expone en detalle a continuación.

2.2.1.1 Software In The Loop (SITL)

Se trata de un simulador que permite ejecutar el autopiloto *Copter* directamente en el PC, sin la necesidad de tener ningún hardware. Es decir, nos permitirá probar el comportamiento del código sin la necesidad de utilizar el sistema físico correspondiente. Más adelante veremos como ejecutarlo en nuestro ordenador y cómo relacionar esta simulación del autopiloto con el *Erle-Copter*.

2.2.2 ROS

El sistema operativo robótico *ROS [6] [7] (Robot Operating System)* es un entorno de desarrollo de software robótico que provee librerías y herramientas destinadas a crear aplicaciones para robots. Proporciona los servicios estándar de un sistema operativo, tales como abstracción de hardware, control de dispositivos de bajo nivel, intercambio de mensajes entre procesos y gestión de paquetes.

Figura 2.3: Logotipo ROS

Una de sus principales ventajas es su código abierto (*open-source*), lo que brinda la oportunidad al desarrollador de reutilizar el código de aplicaciones ya existentes para mejorarlo y poder volverlo a compartir, creando así una gran cantidad de paquetes que están en continua mejora.

Una vez descrito esto, vamos a explicar brevemente cómo funciona *ROS* y cuáles son sus conceptos fundamentales.

Un sistema *ROS* está compuesto por una serie de nodos, los cuales se comunican entre ellos usando mensajes a través de un modelo de publicación/suscripción en un topic. A continuación se explica en qué consisten cada uno de estos conceptos.

Los **nodos** se encargan de la parte computacional del proyecto. *ROS* está diseñado para ser modular, de forma que el sistema de control de un robot está compuesto por varios nodos. Por ejemplo, en el caso de nuestro dron, un nodo controla las velocidades de los motores, otro nodo lleva a cabo la localización del robot, otro nodo controla la posición actual... Los nodos se pueden escribir usando una librería de ROS, que puede ser *roscpp* (lenguaje C++) o *rospy* (Python).

Existe un nodo al cual se le denomina el *Master*, que es fundamental para la comunicación. Se encarga del registro de nombres de todos los nodos en ejecución y los topic y servicios existentes. Sin él, los nodos no serían capaces de encontrarse unos a otros ni de intercambiar mensajes.

Los **mensajes** son la herramienta que tienen los nodos para comunicarse entre sí. Un mensaje es una simple estructura de datos, que pueden ser de diferentes tipos (*int, float, boolean...*)

La forma de intercambiar los mensajes entre nodos es realizada mediante un modelo de publicación/suscripción. Un nodo envía un mensaje publicándolo en un **topic**. El topic es usado para identificar el contenido del mensaje. Otro nodo que esté interesado en esos datos se suscribirá a ese topic, estableciendo así esa comunicación entre nodos. En definitiva, los topic son buses usados por los nodos para transmitir datos. Un topic puede tener múltiples publicadores y suscriptores, mientras que un nodo puede publicar/suscribirse a varios topic.

En la figura 2.4 podemos ver un ejemplo de lo descrito anteriormente. Se trata de un sistema de dos nodos, uno que registra las imágenes obtenidas por una cámara, y otro que se encarga del procesamiento de la imagen.

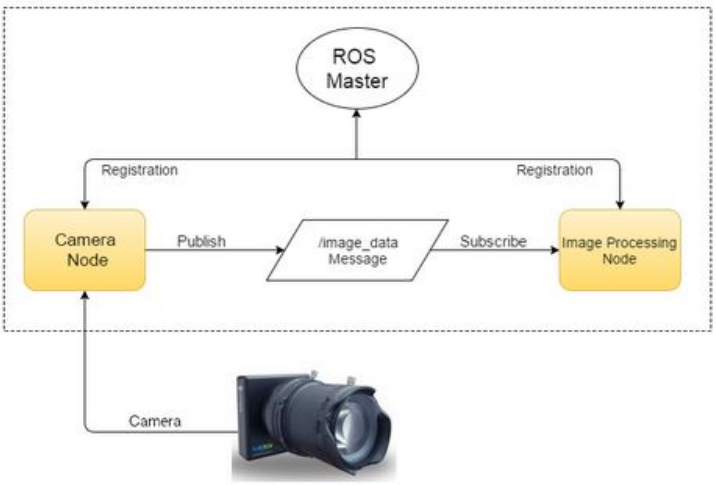


Figura 2.4: Ejemplo de funcionamiento de ROS

Vemos que el nodo llamado *Camera Node* recibe directamente la información de la cámara. Este nodo publica un mensaje con un tipo determinado en el topic */image_data*, tal y como vemos en la figura. El

nodo llamado *Image Processing Node* se suscribe a ese mismo topic porque le interesan esos datos. A su vez, podemos ver que ambos nodos están en contacto con el nodo *Master*, el cual es indispensable para esta comunicación como ya comentamos antes.

Expuesto el funcionamiento básico, vamos a explicar ahora algunos de los comandos más importantes para poder comenzar a trabajar con *ROS*. Para poder tener acceso a los comandos de *ROS*, es necesario ejecutar en la terminal la siguiente sentencia, que nos referencia a los archivos *setup.*sh* del sistema:

```
source /opt/ros/indigo/setup.bash
```

Una vez hecho esto, los siguientes comandos pueden ser lanzados directamente en una terminal de cualquier ordenador Linux que tenga *ROS* instalado.

- *roscore*: Es lo primero que se debe ejecutar para utilizar *ROS*. Es un conjunto de nodos y ejecutables que son imprescindibles en un sistema *ROS*. Es necesario para iniciar *ROS* y que los nodos puedan comunicarse entre ellos. El nodo *Master* mencionado anteriormente se inicializa con este comando.

  ```
  roscore
  ```

- *rosnode*: este comando nos permite visualizar información acerca de los nodos del sistema, incluyendo publicaciones, suscripciones y conexiones. Tiene varias opciones de uso, de las cuales se muestran las de mayor utilidad en nuestro caso.

  ```
  rosnode list --> imprime una lista de los nodos activos del sistema
  rosnode info /nombre_del_nodo --> imprime información del nodo en cuestión
  rosnode kill /nombre_del_nodo --> destruye un nodo en ejecución
  ```

- *rostopic*: este comando nos informa sobre los topic del sistema, sus publicadores, suscriptores y los mensajes publicados en un topic. Al igual que *rosnode*, también nos da varias posibilidades de uso, mostrando aquí las más interesantes para este proyecto.

  ```
  rostopic list --> imprime una lista de los topic activos del sistema
  rostopic info /nombre_del_topic --> imprime información de un topic
  rostopic echo /nombre_del_topic --> muestra los mensajes publicados en un topic
  ```

- *rosrun*: permite ejecutar un nodo de un paquete de *ROS* sin necesidad de saber el directorio donde se encuentra.

  ```
  rosrun nombre_paquete nombre_nodo
  ```

- *roslaunch*: permite lanzar múltiples nodos gracias a un archivo de tipo *.launch*, que contiene los parámetros necesarios y los nodos a ejecutar. El sistema buscará el archivo *.launch* dentro del paquete especificado y se ejecutará.

  ```
  roslaunch nombre_paquete nombre_archivo_launch
  ```

Para concluir este apartado, cabe destacar que *ROS* tiene múltiples distribuciones. Según el soporte de *Erle Robotics*, la recomendada para lograr la simulación del *Erle-Copter* es la versión ***ROS Indigo***, por tanto esta es la distribución que ha sido utilizada en la realización de este proyecto.

2.2.3 Gazebo

Gazebo [8] es un simulador 3D que permite evaluar el comportamiento de un sistema robótico en un entorno virtual. Esta aplicación nos ofrece la posibilidad de simular tanto la cinemática como la dinámica del robot con un alto grado de fidelidad: aceleraciones, fuerzas gravitatorias, inercias, masas, etc. Además de esto, también permite simular una gran cantidad de sensores.

Figura 2.5: Logotipo Gazebo

Este simulador es idóneo para la realización de este trabajo, ya que tiene una relación nativa con *ROS*. *Gazebo* está implementado en *ROS* mediante la utilización de paquetes, los cuales se denominan *gazebo_ros_pkgs*. Estos paquetes contienen plugins que interactúan con cualquier objeto de la escena del simulador y proporcionan la interfaz necesaria para usar *ROS* conjuntamente con *Gazebo*. Esta conexión entre ellos está basada en el funcionamiento de *ROS*, explicado anteriormente en 2.2.2.

Gazebo es considerado como un nodo para *ROS*. Permite lanzarlo mediante un archivo de tipo .launch, que tal y como vimos antes, permiten ejecutar varios nodos a la vez, y uno de ellos puede ser *Gazebo*. Así, se facilita mucho la comunicación entre ellos, ya que al ser como un nodo para *ROS*, este puede publicar y suscribirse a cualquier topic del sistema. Por ejemplo, las velocidades del robot que estén siendo publicadas en un topic de *ROS* podrán ser aplicadas a los motores del robot virtual en *Gazebo* gracias a esto.

En cuanto a la versión que usaremos para la simulación del *Erle-Copter*, será *Gazebo 7*, que es perfectamente compatible con *ROS Indigo*.

En definitiva, mediante el uso de esta plataforma podremos lograr una simulación bastante fiel del comportamiento del dron en 3D, de manera que su comportamiento en el mundo virtual de *Gazebo* sea similar al del mundo real.

2.2.4 MAVLink

Se trata de un protocolo de comunicación que se ha desarrollado especialmente para el intercambio de información entre la estación de control de tierra (se explica más adelante en 2.2.5) y el *UAV*. Consiste en una librería que posee la información necesaria para el intercambio de mensajes llevados a cabo en la comunicación con el *UAV*. Este protocolo está orientado hacia dos propiedades fundamentales: la velocidad y la seguridad en la transmisión. Permite comprobar el contenido del mensaje así como la pérdida de mensajes en el intercambio.
MAVLink [9] es muy fácil de integrar en cualquier sistema, dado que la definición de los mensajes se realiza mediante cabeceras en lenguaje C. Por este motivo, es el protocolo de comunicación que usan la gran mayoría de autopilotos, como es el caso de *ArduPilot*, el que vamos a usar nosotros como ya comentamos en 2.2.1.

Figura 2.6: Logotipo MAVLink

2.2.4.1 MAVROS

MAVROS [10] es un paquete de *ROS* que habilita el protocolo de comunicación *MAVLink* entre un ordenador ejecutando *ROS*, un autopiloto que soporte el protocolo *MAVLink* y una estación de control de tierra. En definitiva, permite comunicar *ROS* con el autopiloto y con la estación de control de tierra mediante *MAVLink*.

Para ejecutar *MAVROS*, basta con correr el nodo principal del paquete: *mavros_node*, el cual se puede ejecutar directamente con el comando *rosrun* mencionado en el apartado 2.2.2.
La gran utilidad de este paquete es que podremos crear nodos de *ROS* que se suscriban y publiquen a los topic de *MAVROS*, y este, gracias al protocolo MAVLink, transmitirá los mensajes al autopiloto.

En apartados posteriores veremos cómo podemos utilizar *MAVROS* en la simulación del *Erle-Copter* para realizar vuelos autónomos, enviándole posiciones a las que nuestro dron tendrá que desplazarse.

2.2.5 Estaciones de control de tierra (GCS)

Una estación de control de tierra, del inglés *Ground Control Station (GCS)*, es una aplicación software que permite comunicarnos con el *UAV* desde nuestro ordenador. Es una herramienta que sirve para monitorizar el estado del *UAV*, como si fuese un "cockpit virtual". Es capaz de mostrar datos de vuelo del dron y su posición, cambiar parámetros del autopiloto, su configuración, calibración de los sensores, etc. Además, también pueden enviarse comandos desde la *GCS* durante el vuelo, por ejemplo para cambiar el modo de vuelo o armado y desarmado de los motores del dron.

Hay una gran cantidad de *GCS* disponibles que son compatibles con *ArduPilot*. Las más comúnmente usadas e interesantes son *APMPlanner 2* [11] y *MAVProxy* [12]. Ambas han sido usadas en la realización de este trabajo, por tanto en las siguientes secciones vamos a describir algunas de sus características más relevantes.

2.2.5.1 MAVProxy

MAVProxy es una GCS para sistemas basados en el protocolo *MAVLink*, tratado en 2.2.4. Esta aplicación está basada en una interfaz de línea de comandos, en una consola. El usuario puede introducir comandos en la consola y de esa forma modificar y cargar parámetros, cambiar el modo de vuelo, o despegar y aterrizar el dron, entre otras cosas.
En apartados posteriores se verán ejemplos de uso de esta *GCS* con la simulación de nuestro dron.

2.2.5.2 APM Planner2

Es considerada como la mejor *GCS* para usar con plataformas Linux. Es de código abierto (*open-source*), lo que hace que esté en continuo desarrollo y mejora. Al igual que la anterior, también está diseñada para los sistemas basados en *MAVLink*, siendo compatible con todo sistema que use este protocolo.

Figura 2.7: Logotipo *APM Planner2*

Posee diversas pestañas con distintas utilidades, de las cuales vamos a destacar las más importantes y las de mayor utilidad para la simulación del *UAV*.

En la pestaña de *Flight Data* ofrece una primera pantalla con información útil para el vuelo del *UAV*, como su localización GPS actual, la altura a la que se encuentra, orientación de la brújula, ángulos Roll, Pitch y Yaw actuales o el número de satélites con los que está trabajando el GPS, entre otras cosas. También nos indica el modo de vuelo que actualmente está cargado en el autopiloto, y el estado de los motores (armados o desarmados). Podemos ver todo esto de forma gráfica en la figura 2.8.

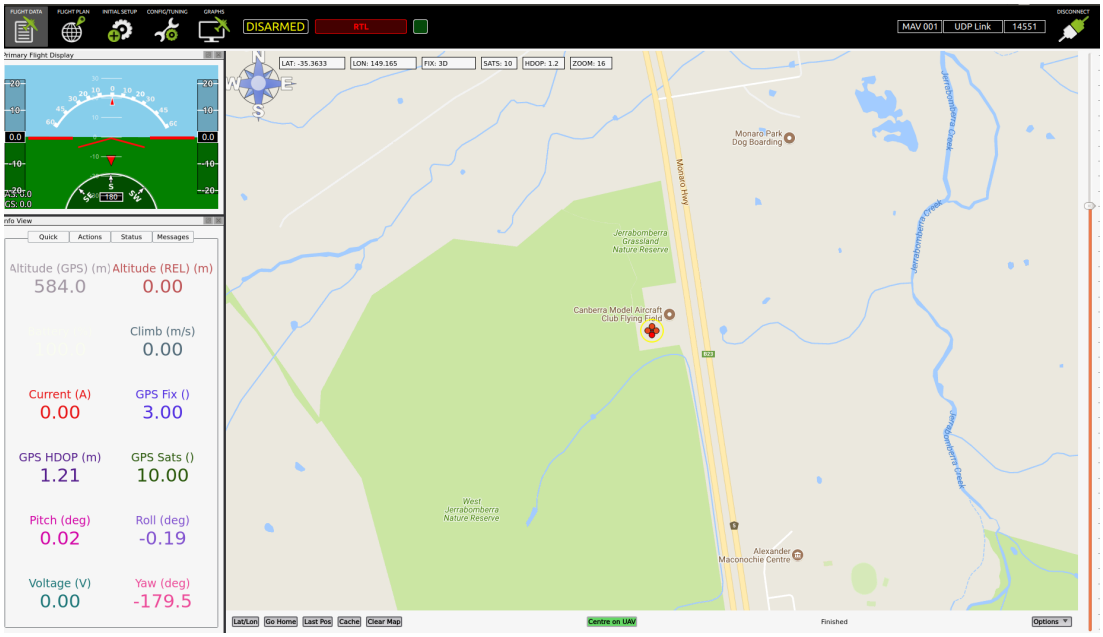


Figura 2.8: Pestaña *Flight Data* de *APM Planner 2*

En la pestaña *Config/Tuning*, en el apartado de *Full Parameter List*, tal y como vemos en la imagen 2.9, tenemos la posibilidad de ver y modificar todos los parámetros de funcionamiento del autopiloto, para poder personalizar el comportamiento del dron. Para entender el significado de cada parámetro podemos consultar la información de la página oficial de *ArduCopter*.

En esa misma pestaña de *Config/Tuning*, si vamos a la sección *Extended Tuning* (figura 2.10), vemos que aparecen una serie de valores PID. Estos valores pueden ser modificados para variar el comportamiento de los controladores PID del autopiloto, lo cual no es recomendable ya que con un valor inadecuado podríamos hacer el vuelo del dron inestable. De hecho, algunos de esos valores están restringidos, sin ni siquiera darnos la posibilidad de modificarlos.
Sin embargo, experimentalmente nosotros trataremos de cambiar alguno de esos valores que sí nos permiten modificar, para analizar cual es la influencia en el comportamiento del UAV, lo cual veremos en el capítulo 5.2.

Figura 2.9: *Full Parameter List* de *APM Planner 2*

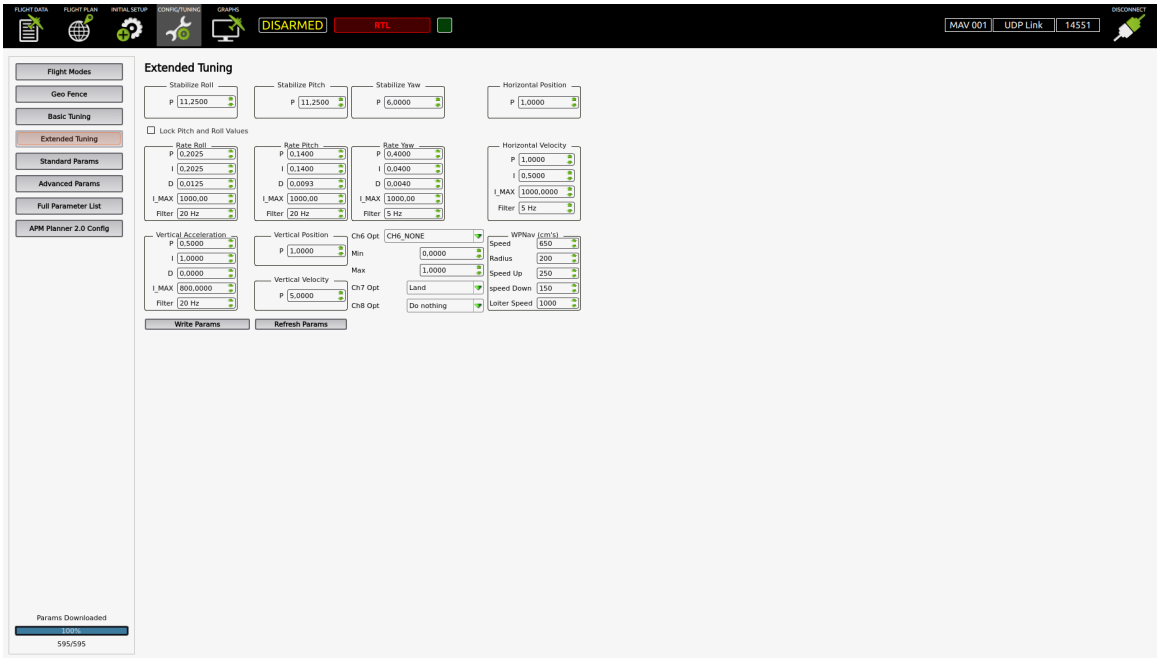


Figura 2.10: *Extended Tuning* de *APM Planner 2*

Capítulo 3

Estudio teórico

Un *quadcopter* se convierte en un *UAV* o dron cuando es capaz de realizar vuelos autónomos, para lo cual es indispensable un controlador de vuelo o autopiloto. Esto significa recopilar la información de los acelerómetros y giróscopos y combinarlo con los datos del GPS y del barómetro, de forma que el controlador de vuelo pueda estimar la orientación y posición del vehículo.

Para llevar a cabo la simulación del *Erle-Copter*, necesitamos tener su modelo, tanto físico como el del controlador de vuelo que utiliza, en nuestro caso *ArduCopter*. En este capítulo veremos los parámetros más importantes que definen el modelo físico del dron dentro del entorno de simulación, así como los distintos controladores y parámetros que utiliza el autopiloto para lograr estabilizar el vehículo. Finalmente, trataremos en detalle los modos de vuelo más importantes y veremos la importancia de los controladores del autopiloto en el funcionamiento de cada uno de ellos.

3.1 Parámetros del modelo físico del Erle-Copter

Los parámetros del modelo se definen en un archivo de nombre *erlecopter.xacro*, el cual se puede encontrar en el directorio de archivos una vez que hayamos instalado las herramientas necesarias descritas en el apéndice B.

El formato *Xacro* se trata de un lenguaje de programación *XML* basado en macros. El *XML (eXtensible Markup Language o Lenguaje de Marcado Extensible)* es un lenguaje cuya función principal es la descripción de datos. Xacro permite construir archivos *XML* de forma sencilla y visual mediante macros, mientras que usando largas expresiones de *XML* sería más laborioso. Es muy útil cuando se trabaja con documentos largos de *XML*, como es el caso de las descripciones de robots. En este fichero *Xacro*, se incluyen archivos *URDF* (Unified Robot Description Format o Formato Unificado de Descripción de Robots), formato con el cual trabaja *ROS*. Estos archivos definen los componentes del robot, como por ejemplo los rotores, con sus cinemáticas y dinámicas.

En definitiva, *erlecopter.xacro* es el bloque encargado de aunar en el modelo de *Erle-Copter* todas las propiedades y elementos físicos del modelo. A continuación, vamos a analizar de forma más detallada el contenido de este archivo.

Según vemos en la figura 3.1, en las primeras líneas se puede apreciar la versión de *XML* utilizada, la habilitación del formato *Xacro* y el nombre del archivo.

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

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

Figura 3.1: Primeras líneas de *erlecopter.xacro*

En las líneas siguientes, presentes en la figura 3.2, se pueden observar una serie de propiedades que caracterizan al *Erle-Copter*. Podemos ver la masa, longitud, anchura y altura del vehículo, la masa de los rotores, longitudes de los brazos, el radio de los rotores, la constante de los motores, constantes de tiempo, velocidad máxima del rotor en radianes por segundo, etc. Esta serie de parámetros definen las propiedades físicas del *Erle-Copter* en el entorno de simulación, y pueden ser cambiadas por el usuario.

```
  <!-- Properties -->
  <xacro:property name="namespace" value="erlecopter" />
  <xacro:property name="rotor_velocity_slowdown_sim" value="10" />
  <xacro:property name="mesh_file" value="erlecopter.dae" />
  <xacro:property name="mesh_scale" value="1 1 1"/> <!-- 1 1 1 -->
  <xacro:property name="mesh_scale_prop" value="1 1 1"/>
  <xacro:property name="mass" value="1.1" /> <!-- [kg] -->
  <xacro:property name="body_length" value="0.18" /> <!-- [m] 0.10 -->
  <xacro:property name="body_width" value="0.12" /> <!-- [m] 0.10 -->
  <xacro:property name="body_height" value="0.10" /> <!-- [m] -->
  <xacro:property name="mass_rotor" value="0.005" /> <!-- [kg] -->
  <xacro:property name="arm_length_front_x" value="0.141" /> <!-- [m] 0.1425  0.22 -->
  <xacro:property name="arm_length_back_x" value="0.141" /> <!-- [m] 0.154  0.22 -->
  <xacro:property name="arm_length_front_y" value="0.141" /> <!-- [m] 0.251  0.22 -->
  <xacro:property name="arm_length_back_y" value="0.141" /> <!-- [m] 0.234  0.22 -->
  <xacro:property name="rotor_offset_top" value="0.030" /> <!-- [m] 0.023-->
  <xacro:property name="radius_rotor" value="0.12" /> <!-- [m] -->
  <xacro:property name="motor_constant" value="8.54858e-06" /> <!-- [kg.m/s^2] -->
  <xacro:property name="moment_constant" value="0.016" /> <!-- [m] -->
  <xacro:property name="time_constant_up" value="0.0125" /> <!-- [s] -->
  <xacro:property name="time_constant_down" value="0.025" /> <!-- [s] -->
  <xacro:property name="max_rot_velocity" value="838" /> <!-- [rad/s] -->
  <xacro:property name="sin30" value="0.5" />
  <xacro:property name="cos30" value="0.866025403784" />
  <xacro:property name="sqrt2" value="1.4142135623730951" />
  <xacro:property name="rotor_drag_coefficient" value="8.06428e-05" />
  <xacro:property name="rolling_moment_coefficient" value="0.000001" />
  <xacro:property name="color" value="DarkGrey" />
```

Figura 3.2: Propiedades en *erlecopter.xacro*

A continuación, en la figura 3.3, se muestran los bloques de inercias, en los que aparecen las inercias del cuerpo del dron y de los rotores, asumiendo que son cuboides de base cuadrada de altura 3mm y ancho 15mm.

```
  <!-- Property Blocks -->
  <xacro:property name="body_inertia">
    <inertia ixx="0.0347563" ixy="0.0" ixz="0.0" iyy="0.0458929" iyz="0.0" izz="0.0977" /> <!-- [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2]
  </xacro:property>

  <!-- inertia of a single rotor, assuming it is a cuboid. Height=3mm, width=15mm -->
  <xacro:property name="rotor_inertia">
    <inertia
    ixx="${1/12 * mass_rotor * (0.015 * 0.015 + 0.003 * 0.003) * rotor_velocity_slowdown_sim}"
    iyy="${1/12 * mass_rotor * (4 * radius_rotor * radius_rotor + 0.003 * 0.003) * rotor_velocity_slowdown_sim}"
    izz="${1/12 * mass_rotor * (4 * radius_rotor * radius_rotor + 0.015 * 0.015) * rotor_velocity_slowdown_sim}"
    ixy="0.0" ixz="0.0"  iyz="0.0" /> <!-- [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] -->
  </xacro:property>
```

Figura 3.3: Inercias en *erlecopter.xacro*

En la siguiente sección del código se incluyen los anteriormente mencionados archivos *URDF*. En la figura 3.4 podemos observar la inclusión de la base del multirrotor, a la cual se le atribuyen las

propiedades definidas en las líneas anteriores, y también son insertadas las inercias del cuerpo que hemos visto anteriormente.

```
47  <!-- Included URDF Files -->
48  <!-- <xacro:include filename="$(find rotors_description)/urdf/multirotor_base.xacro" /> -->
49  <xacro:include filename="$(find ardupilot_sitl_gazebo_plugin)/urdf/multirotor_base.xacro" />
50
51  <!-- Instantiate multirotor_base_macro once -->
52  <xacro:multirotor_base_macro
53    robot_namespace="${namespace}"
54    mass="${mass}"
55    body_length="${body_length}"
56    body_width="${body_width}"
57    body_height="${body_height}"
58    mesh_file="${mesh_file}"
59    mesh_scale="${mesh_scale}"
60    color="${color}"
61    >
62    <xacro:insert_block name="body_inertia" />
63  </xacro:multirotor_base_macro>
```

Figura 3.4: Inclusión de la base en *erlecopter.xacro*

En las líneas posteriores del código son incluidos una serie de elementos y sensores, tales como una cámara o un sonar. Al no ser objeto principal de este trabajo, estas líneas serán omitidas.

En la sección final del código, se incluyen los cuatro rotores del modelo, asignándoles las propiedades y las inercias que estaban definidas en líneas anteriores. En la figura 3.5 vemos la definición de uno de ellos. Podemos apreciar que se puede elegir la posición que tomará cada rotor en el marco del dron, mediante la propiedad *suffix*. En este caso es atrás a la derecha (*back_right*), como podemos ver en el código. También es posible elegir el sentido de giro, mediante la propiedad *direction*. Este rotor en concreto está configurado en sentido horario (cw). En total, debe haber dos que giren en sentido horario y dos en sentido antihorario (ccw). También se tiene que asignar un número al motor, que está presente en la propiedad *motor_number*. En este caso es el 3, tal y como podemos observar. En total, habrá cuatro rotores definidos (0,1,2,3), mostrándose en el código de la figura el último de ellos.

```
285  <xacro:vertical_rotor robot_namespace="${namespace}"
286    suffix="back_right"
287    direction="cw"
288    motor_constant="${motor_constant}"
289    moment_constant="${moment_constant}"
290    parent="base_link"
291    mass_rotor="${mass_rotor}"
292    radius_rotor="${radius_rotor}"
293    time_constant_up="${time_constant_up}"
294    time_constant_down="${time_constant_down}"
295    max_rot_velocity="${max_rot_velocity}"
296    motor_number="3"
297    rotor_drag_coefficient="${rotor_drag_coefficient}"
298    rolling_moment_coefficient="${rolling_moment_coefficient}"
299    mesh="erlecopter_prop"
300    mesh_scale="${mesh_scale_prop}"
301    color="Black">
302    <origin xyz="-${arm_length_back_x} -${arm_length_back_y} ${rotor_offset_top}" rpy="0 0 0" />
303    <xacro:insert_block name="rotor_inertia" />
304  </xacro:vertical_rotor>
```

Figura 3.5: Inclusión de los rotores en *erlecopter.xacro*

3.2 Parámetros del controlador de vuelo

En esta sección presentaremos los distintos controladores que maneja el autopiloto *ArduCopter* para conseguir la estabilización del dron. Adicionalmente, conoceremos el filtro de Kalman extendido *(EKF)* [13] incorporado en el autopiloto, que fusiona todas las medidas de los sensores y logra una estimación de su posición y orientación.

Comenzaremos con una introducción sobre el funcionamiento básico de un PID, basándonos en el esquema de la imagen 3.6.

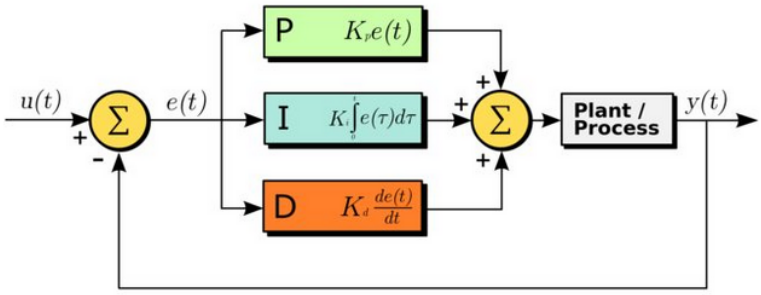


Figura 3.6: PID básico

Por un lado tenemos la referencia, *u(t)*, que es la consigna que queremos enviar. En el caso de un *UAV*, podría ser una consigna de ángulo *roll* deseado, por ejemplo. Por otro lado, tenemos la señal *y(t)*, que es el valor real de la magnitud a controlar, medido por los sensores del sistema. Siguiendo el ejemplo, esta señal sería la medida real del *roll* obtenida por el sensor. La resta de estos dos valores nos daría la señal de error, *e(t)*. El objetivo del controlador PID es corregir este error, tratando de que sea 0, de modo que la referencia constante sea igual al valor medido. Para lograr este propósito el controlador debe proporcionar una salida adecuada a través de los actuadores.

Los actuadores (que en este caso son los cuatro motores) podrán modificar hasta seis magnitudes diferentes para conseguir el control en posición y orientación del quadrotor: podremos controlar la posición en los ejes *x*, *y* y *z* y la orientación (*attitude*) del dron, mediante los ángulos *roll*, *pitch* y *yaw*. La figura 3.7 permite visualizar estos giros sobre los ejes del propio dron de forma gráfica.

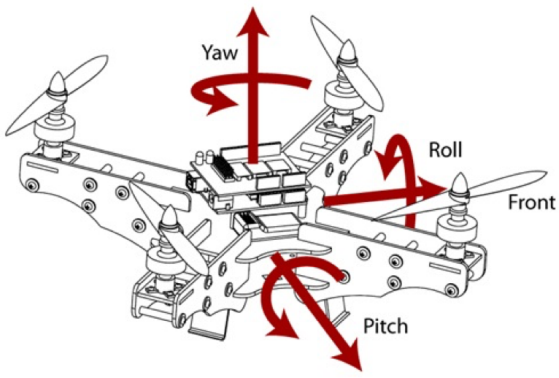


Figura 3.7: Ángulos de giro sobre los ejes del dron

Los controladores que usa *ArduCopter* son filtros PID, caracterizados por tres valores: Proporcional, Integral y Derivativo.

- Término P: se centra en el error actual. Realiza una corrección proporcional al error en ese momento.
- Término I: es la acumulación de errores pasados. Se fija en los hechos que suceden a lo largo del tiempo. Por ejemplo, si el *UAV* constantemente se aleja de una consigna debido al viento, actuará sobre los motores para contrarrestarlo. Es proporcional a la integral del error.
- Término D: es una predicción de errores futuros. Se centra en lo rápido que la señal medida se acerca a la señal de consigna. Es proporcional a la derivada del error.

Alterar estos valores PID del controlador puede afectar al comportamiento del *quadcopter* de diferentes maneras. A continuación, vamos a conocer cuáles son los efectos que puede provocar la modificación de cada uno de ellos.

- La ganancia P determina el esfuerzo del controlador de vuelo para corregir el error y lograr el vuelo deseado. Si aumentamos demasiado esta ganancia, el vehículo se volverá muy sensible y tenderá a sobrecorregir continuamente su posición, lo que puede causar sobreimpulsos y frecuencias de oscilación altas. En cambio, si la ganancia es muy baja, el dron puede volverse poco sólido en su vuelo, con una reacción más lenta ante las consignas.
- La ganancia I determina el esfuerzo del autopiloto para mantener la posición y orientación del dron ante fuerzas externas. Si se hace demasiado grande, podemos causar que el vehículo no responda correctamente ante las consginas, dando la sensación de ser demasiado rígido, con reacciones lentas.
- La ganancia D actúa de amortiguador, reduce las sobrecorrecciones y los sobreimpulsos causadas por la ganancia P. Si esta ganancia es muy baja, pueden aparecer pequeños rebotes no deseados al finalizar los giros sobre los ejes. Aumentando la D, puede introducir vibraciones en el *quadcopter*, dado que amplifica el ruido del sistema. Esto puede causar sobrecalentamiento en los motores y oscilaciones en el vuelo del dron.

En la figura 3.8 podemos ver un esquema de cómo realiza *ArduCopter* el control angular del dron (*Attitude Control*) [14] para cada eje.

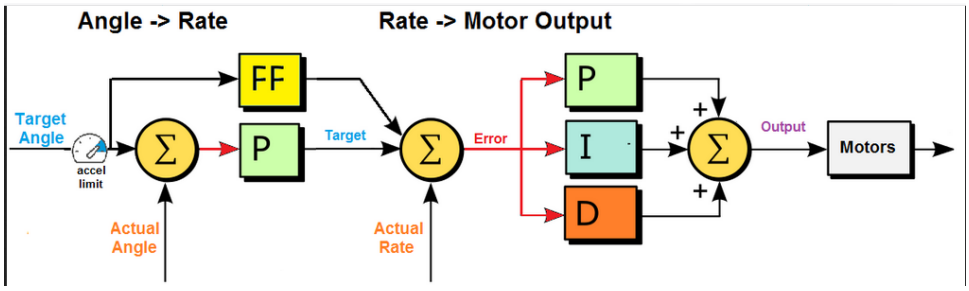


Figura 3.8: Esquema de *Attitude Control* de *ArduCopter*

El controlador P es usado para convertir el error de ángulo en una velocidad de rotación deseada. Esta se compara con la velocidad de rotación actual, y se obtiene la señal de error de velocidad de rotación. El controlador PID convierte el error de rotación en actuación para los motores. Como podemos observar, sigue el esquema básico de funcionamiento que explicamos en el inicio de esta sección.

ArduCopter posee diversos controladores de este tipo, los cuales podemos visualizar haciendo uso de una estación de control, por ejemplo *APMPlanner 2*. Podemos encontrarlos dentro de la sección *Config/Tuning*, en el apartado *Extended Tuning*, tal y como vimos en el capítulo anterior. En la figura 3.9 podemos ver de nuevo esta sección, con los controladores de *ArduCopter* resaltados con colores.

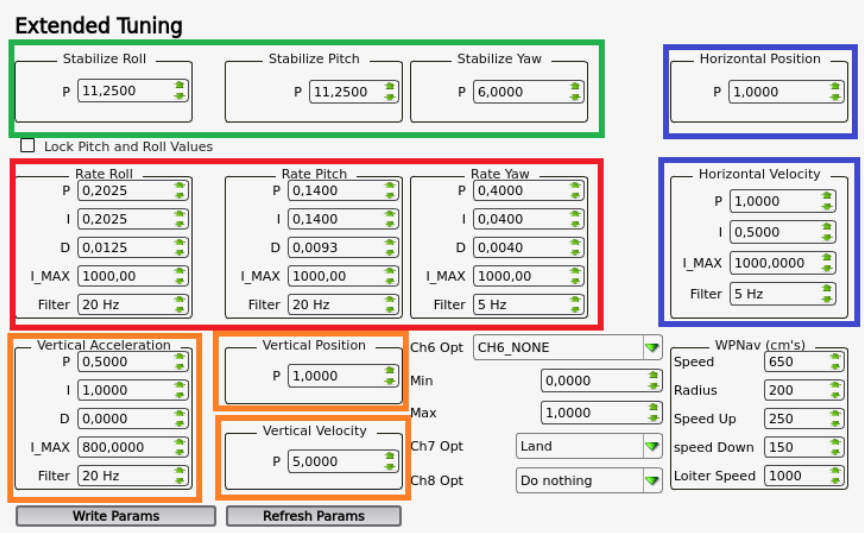


Figura 3.9: Controladores de *ArduCopter* en *APM Planner 2*

El controlador que está activo en cada momento depende del modo de vuelo. En el código de *ArduCopter*, cada modo de vuelo está programado para llamar a las funciones correspondientes, que inician los controladores necesarios en el modo actual.

Los controladores de velocidad de rotación, *Rate Roll, Pitch y Yaw*, resaltados en rojo en la figura 3.9, son los más importantes y están activos en todos los modos. Si estos están bien ajustados, el vehículo se mantendrá estable en el vuelo.

El resto de controladores serán utilizados por el autopiloto según el modo de vuelo, lo cual veremos más en detalle en la sección 3.3. Los controladores *Stabilize Roll, Pitch y Yaw* son los controladores de ángulo, podemos verlos recuadrados en color verde en la figura. Resaltados en color naranja podemos encontrar dos controladores P, para la posición y la velocidad en el eje vertical, y un PID para la aceleración en ese mismo eje. Por último, en color azul, podemos encontrar un controlador P para la posición horizontal, mientras que la velocidad en los ejes horizontales es controlada mediante un PI.

Los valores de estas ganancias, mostrados en la figura 3.9 anterior, son las configuradas por defecto en los parámetros de *ArduCopter*, y en nuestro caso aseguran un vuelo estable del dron en la simulación. Posteriormente, en el capítulo 5.2.2, comprobaremos que cambiar algunas de estas ganancias puede afectar a los resultados del vuelo simulado.

3.2.1 Filtro de Kalman extendido (EKF)

ArduCopter utiliza este algoritmo para estimar la posición, velocidad y orientación angular del vehículo. Para ello, se basa en las medidas de los giróscopos, acelerómetros, brújulas, GPS y barómetro. A lo largo del tiempo, ha habido una evolución en los filtros de estimación usados por *ArduCopter*. En la versión usada para este proyecto, la 3.4, el filtro que usa es el *EKF2*, como vemos en la imagen 3.10.

La gran ventaja que ofrece el *EKF* sobre otros filtros más simples, como *Inertial Nav*, es que fusionando todas las medidas disponibles es capaz de rechazar medidas con errores muy significativos. Esto hace al dron menos susceptible ante fallos que afecten a un solo sensor. Además, el *EKF2* permite ejecutar dos núcleos/instancias en paralelo, si se tienen dos (o más) *IMUs* presentes en el sistema. Cada núcleo usará las medidas de una *IMU* diferente. Se tendrá en cuenta solamente la salida de uno de los núcleos, siendo ese núcleo el que reporta mejor consistencia en los datos de la *IMU* asociada.

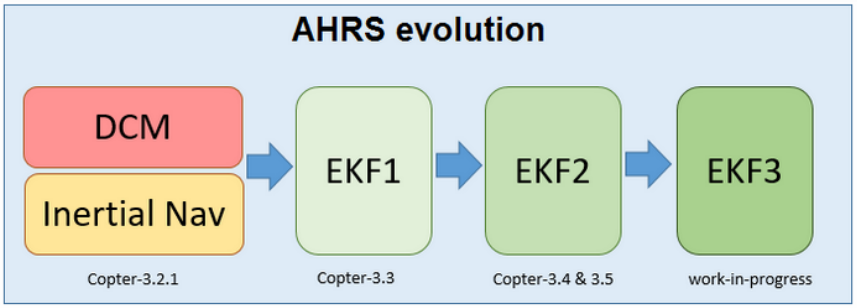


Figura 3.10: Evolución filtros de estimación en *ArduCopter*

El *EKF2* estima un total de 24 estados que caracterizan la dinámica del dron, como puede ser *roll, pitch, yaw*, posiciones, velocidades, etc. El principio de funcionamiento del filtro es el siguiente:

1. En primer lugar, se integran las velocidades angulares proporcionadas por los giróscopos de la *IMU* [15] para calcular la posición angular.

2. Las aceleraciones medidas por los acelerómetros de la *IMU* se convierten de sistema de coordenadas de cuerpo (con su origen en el centro de masas del vehículo, ya que los sensores integrados miden con respecto a ese sistema de referencia) a coordenadas *North East Down* [18], usando la posición angular obtenida anteriormente. Estas aceleraciones se integran para calcular la velocidad, y se vuelve a integrar para obtener la posición.

3. La información del *IMU* (acelerómetros y giróscopos) permite caracterizar también el ruido que forma parte de la matriz de covarianza de ruido de estado del *EKF*.

Los tres pasos anteriores son repetidos cada vez que se obtengan nuevos valores de la *IMU*, hasta que una nueva medida en otro sensor esté disponible. El filtro proporciona información de posicionamiento local (relativo) a partir de una doble integración, por este motivo los errores en posición y velocidad crecen muy rápido en poco tiempo. Es por esto que el *EKF* combina las medidas de la *IMU* con las de otros sensores, como el GPS, barómetro, brújula, etc. que le permiten calcular una estimación más precisa de la posición, velocidad y orientación.

Cuando se recibe una medida del GPS, se corrige la estimación de los estados del filtro (entre ellos la pose 3D del dron) y la matriz de covarianza del error de estimación. Esto es un ejemplo de cómo el *EKF* usa la medida de posición horizontal del GPS para corregir, el proceso con la medida de otro sensor sería similar.

Como hemos visto, con la utilización de este filtro el autopiloto es capaz de obtener una estimación cada 10 Hz de su posición, velocidad y orientación, gracias a las medidas que proporcionan los diferentes sensores que lleva integrado el dron.

3.3 Modos de vuelo

Tras conocer los distintos controladores que posee *ArduCopter*, vamos a analizar los modos de vuelo que proporciona este autopiloto. Tiene una gran cantidad de modos disponibles [16], en este apartado mencionaremos los más importantes y comúnmente usados. Cada modo tiene una función específica diferente; para poder llevarla a cabo son fundamentales los distintos controladores disponibles. A continuación, se expone en detalle el funcionamiento de cada modo.

- **Stabilize**: este modo dota al piloto del control total del vehículo. Permite controlar el dron de forma manual, pero autoajusta los ángulos *roll* y *pitch* para conseguir estabilizarlo. Como vemos, este modo tiene una gran utilidad en cuanto a manejo manual del vehículo, sin embargo en modo automático no resulta demasiado útil.

  Para este modo, son fundamentales los controladores *Rate Roll, Pitch*, y *Stabilize Roll, Pitch*, representados en colores rojo y verde en la figura 3.9.

  – Los controladores *Stabilize Roll y Pitch* controlan la capacidad de respuesta del dron para dichos ángulos en función de la entrada proporcionada y el error entre los ángulos *pitch y roll* deseados y los reales. Un valor demasiado grande de P hará que el vehículo oscile bruscamente durante esos giros, mientras que un valor demasiado pequeño puede provocar dificultades a la hora de manejar el dron por la lentitud de los giros.

  – Los controladores *Rate Roll y Pitch* controlan la salida a los motores basándose en la velocidad de rotación deseada.

- **AltHold**: modo de mantenimiento de altura (*altitude hold mode*). En este modo, el dron tratará de mantener una altura fija, mientras permite controlar los ángulos *roll, pitch y yaw* al piloto.

  Los controladores verticales de posición, velocidad y aceleración, resaltados en naranja en la figura 3.9, son básicos para el correcto funcionamiento de este modo.

  – *Vertical Acceleration* se encarga de convertir el error de aceleración (la diferencia entre la aceleración deseada y la actual) en actuación para los motores. En este controlador, es importante que se mantenga la relación 1:2 entre la P y la I si se modifican estas ganancias (es decir, la ganancia de la parte integral tiene que ser el doble que la parte proporcional).

  – El lazo de control de velocidad, en el que está incluido el controlador *Vertical Velocity*, tiene la función de convertir velocidad deseada en el eje vertical en la aceleración deseada.

  – El controlador *Vertical Position* es usado para convertir el error de altura (la diferencia entre la altura deseada y la actual) en la velocidad deseada en el eje vertical. Cuanto mayor sea su valor, más agresiva será la respuesta para mantener la altura.

  En la figura 3.11 podemos ver el triple lazo de control utilizado en este modo.

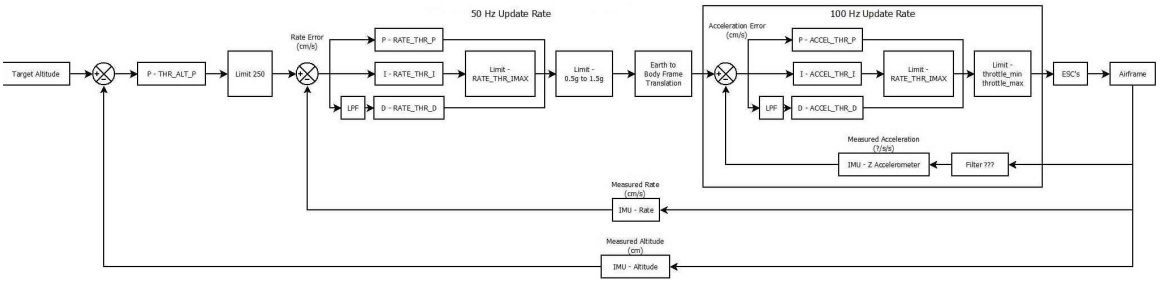


Figura 3.11: Lazo de control del modo *AltHold*

- **Loiter**: en este modo, el vehículo tratará de mantener automáticamente su posición, altura y orientación actuales cuando el piloto no introduzca ninguna entrada. Es decir, cuando se gobierna manualmente el dron, si no se introduce ninguna consigna, el vehículo trata de mantenerse en el aire en la posición en la que está. Es, por tanto, un modo que depende en gran medida del GPS, de la brújula y del barómetro.

Loiter trae incorporado el controlador de altura del modo *AltHold*, el cual se explicó anteriormente. Para la posición horizontal, utiliza los controladores horizontales de posición y velocidad, recuadrados en azul en la figura 3.9.

  - El controlador *Horizontal Position* convierte el error de posición horizontal (la diferencia entre la posición deseada y la posición actual) a una velocidad deseada.
  - *Horizontal Velocity* convierte la velocidad deseada en una aceleración deseada. Esta se convierte en un ángulo de inclinación deseado, el cual se introduce en el mismo controlador angular que en el modo *Stabilize* visto anteriormente, usando los controladores *Rate Roll, Pitch*, y *Stabilize Roll, Pitch*.

- **Land**: es un modo de aterrizaje. Al activar este modo, el dron tratará de aterrizar en la posición en la que se encuentre, simplemente modificando su posición en el eje vertical. Una vez que el autopiloto detecte que ha llegado a tierra, se desarmarán los motores. Si el vehículo dispone de señal GPS, durante el aterrizaje controlará su posición horizontal para descender de la forma más recta posible.

  El principal problema que tienen los modos de aterrizaje es el siguiente: cuando el dron se acerca al suelo, el movimiento de las hélices crea una burbuja de aire por debajo de él que modifica la presión, y puede afectar a las medidas proporcionadas por el barómetro. Si esto sucede en la simulación, podemos tener el problema de que los motores no se desarmen una vez que el dron ha llegado a tierra, ya que debido a este efecto el autopiloto podría no detectar la llegada al suelo del vehículo. Una solución posible sería blindar el barómetro mediante alguna cápsula, evitando así que se vea modificado por ese efecto.

- **RTL**: se trata de otro modo de vuelta a tierra (return to launch). La principal diferencia con el modo *Land* es que este modo trata de aterrizar el dron en la posición donde fueron armados los motores, es decir, en la posición "casa". Para ello, necesita señal de GPS, para poder conocer la posición a la que tiene que llegar.

  Los dos parámetros más importantes de este modo son *RTL_ALT*, que indica la altura a la que viajará el dron antes de volver a la posición de aterrizaje, y *RTL_ALT_FINAL*, que determina la altura que mantendrá el vehículo una vez llegue a la posición de "casa". En nuestro caso, configuramos *RTL_ALT* a 0 para que viaje a la altura actual hasta la posición de aterrizaje, y *RTL_ALT_FINAL* a 0, para que al llegar a la posición el dron aterrice.

- **Guided**: es un modo que permite guiar el vehículo a un punto objetivo de manera autónoma. Se puede utilizar de varias formas, enviándole los puntos objetivo de diferentes maneras, por ejemplo posiciones o velocidades. En nuestro caso, lo utilizaremos con posiciones locales, como se verá en el capítulo posterior.

  Este modo utiliza el control de altura del modo *AltHold* y el control de posición del modo *Loiter*, los cuales vimos anteriormente. De esta forma, cada vez que establezcamos un punto objetivo, el dron viajará hasta él, y mantendrá la posición y la altura hasta que llegue un punto objetivo nuevo al que desplazarse.

Capítulo 4

Simulación del Erle-Copter

Una vez instaladas todas las herramientas necesarias para la simulación y configurado el entorno de trabajo, descrito en el apéndice B, ya estamos en disposición de lanzar la simulación del *Erle-Copter* en nuestro ordenador.

En este capítulo se describe paso a paso cómo llegar a lanzar esta simulación, las primeras pruebas de funcionamiento con *MAVProxy*, como cambiar el modo de vuelo o hacer despegar el *UAV*. Posteriormente, se programará una trayectoria con varios puntos objetivo a los que el dron tendrá que viajar de forma autónoma, para lo cual haremos uso de la herramienta *ROS*.

4.1 Lanzamiento de la simulación

Para comenzar, tenemos que ejecutar *MAVProxy* y *ArduCopter* sobre *SITL*. Para ello, abriremos una terminal y ejecutamos los siguientes comandos:

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

sim_vehicle.sh es el *script* de ejecución de *SITL*. Una vez ejecutado, se iniciará la simulación del autopiloto conjuntamente con *MAVProxy*. Podemos ver la terminal resultante en la figura 4.1, la cual está esperando a una conexión con *ROS/Gazebo* para poder iniciar *MAVProxy*.

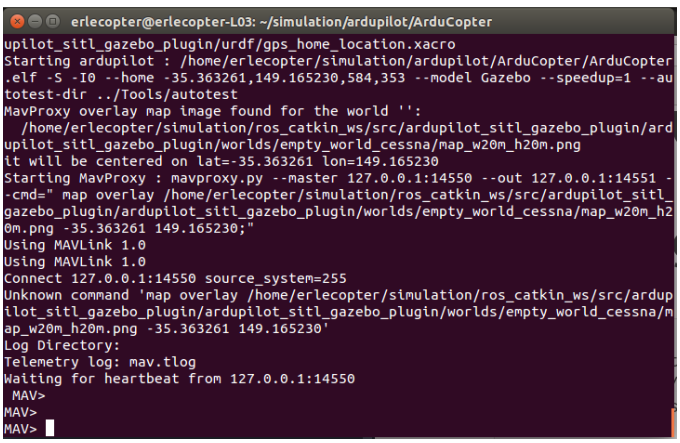


Figura 4.1: Terminal con *sim_vehicle.sh* ejecutado

A continuación, vamos a lanzar el plugin *ardupilot_sitl_gazebo_plugin*, que actúa de interfaz entre *ROS/Gazebo* y *ArduCopter*. Para ello, abrimos una nueva terminal y ejecutamos lo siguiente:

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

Una vez ejecutado, se abrirá *Gazebo*, tal y como vemos en la ilustración 4.2, y también se incializa *mavros*, que permite la comunicación entre *ROS* y el autopiloto.

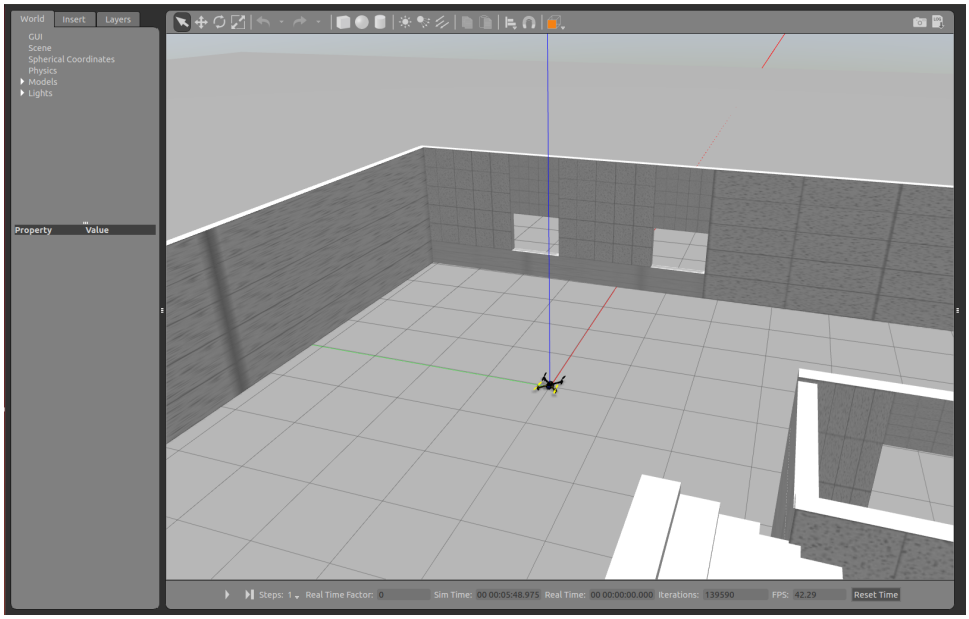
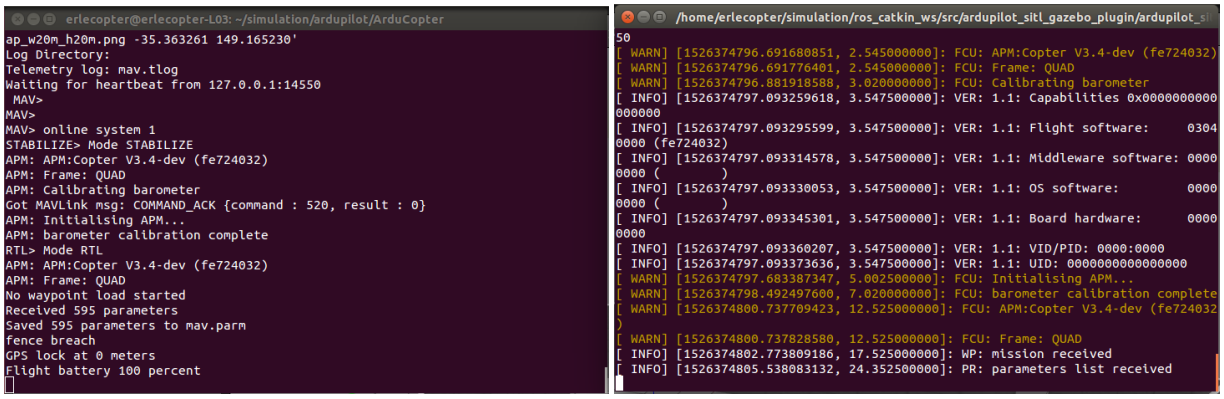


Figura 4.2: Inicialización de la simulación del *Erle-Copter* en *Gazebo*

Las dos terminales que hemos utilizado quedarán de la siguiente forma: en la primera, tendremos la línea de comandos de *MAVProxy* (*MAVProxy Command Prompt*), como vemos en la figura 4.3a, en la cual podemos enviar comandos directamente al autopiloto.
En la segunda terminal (figura 4.3b), podremos ver los mensajes que lanza *mavros* acerca del controlador de vuelo. Vemos que indica la versión de *ArduCopter* que está usando (3.4 en este caso), su inicialización, información acerca de la calibración de sensores, etc. Todo esto indica que la conexión con el autopiloto se ha realizado correctamente.



(a) Línea de comandos de *MAVProxy* (b) Terminal de *mavros*

Figura 4.3: Terminales de *MAVProxy* y *mavros* listas para la simulación

Una vez se haya cargado completamente la simulación y tengamos las dos terminales como en la figura 4.3, tendremos que cargar los parámetros necesarios para que el dron simulado se comporte de manera

estable y sea lo más fiel posible a la realidad. Estos parámetros son proporcionados por *Erle Robotics* y han sido previamente testados. Podemos cargarlos directamente en el autopiloto usando la terminal de MAVProxy, con el siguiente comando:

```
param load home/erlecopter/simulation/ardupilot/Tools/Frame_params/Erle-Copter.param
```

Nótese que en el caso de este proyecto, el directorio en el cual se encuentra la carpeta *simulation* es en *home/erlecopter*, pero se debe sustituir este directorio en el comando anterior por el que corresponda en cada caso.

A modo de resumen, en la figura 4.4 se expone un diagrama en el que se pueden seguir los pasos realizados en esta sección para lograr el lanzamiento de la simulación.

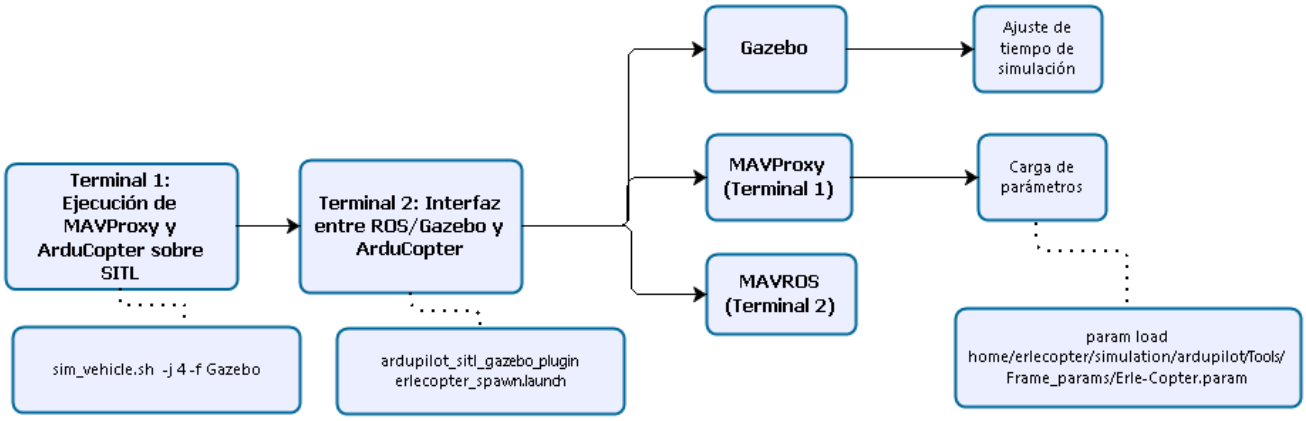


Figura 4.4: Diagrama de bloques del proceso de lanzamiento de la simulación

Una vez finalizado todo este proceso, tendremos un sistema de simulación con el cual podemos enviar comandos mediante *MAVProxy* al autopiloto, y estos comandos podrán verse reflejados en el dron simulado en *Gazebo*, lo cual vamos a ver en la sección a continuación.

4.1.1 Primeras pruebas de simulación con MAVProxy

En primer lugar, antes de simular, es necesario ajustar el tiempo de simulación en *Gazebo* para que sea igual al tiempo real. *Gazebo* por defecto está configurado para ejecutarse lo más rápido que le sea posible. Por este motivo, si no lo ajustamos para que sea similar al tiempo real, al simular el dron sería "a cámara rápida".

Para ajustarlo, tenemos que ir a la ventana de *Gazebo* que se abrió al lanzar la simulación, la cual vimos en la figura 4.2. Si abrimos la pestaña *World* y desplegamos la sección *Physics*, veremos que aparecen una serie de propiedades con su valor. De estas, nos fijaremos en *real_time_update_rate* y *max_step_size* [17], tal y como vemos en la figura 4.5. El producto de estas dos propiedades es el que determina la relación del tiempo de simulación con el real, de manera que si el producto es igual 1, ambos tiempos coincidirán. El *max_step_size* es la duración, en segundos, del paso de actualización de las físicas. Es fijo, de 2.5 ms. El parámetro *real_time_update_rate* especifica, en Hz, la cantidad de actualizaciones de las físicas que se intentarán por segundo. Por tanto, tendremos que ajustarlo a 400 Hz para que al multiplicarlo por 2.5 ms el producto nos dé 1, y de esta manera el tiempo de la simulación irá a la misma velocidad que el tiempo real.

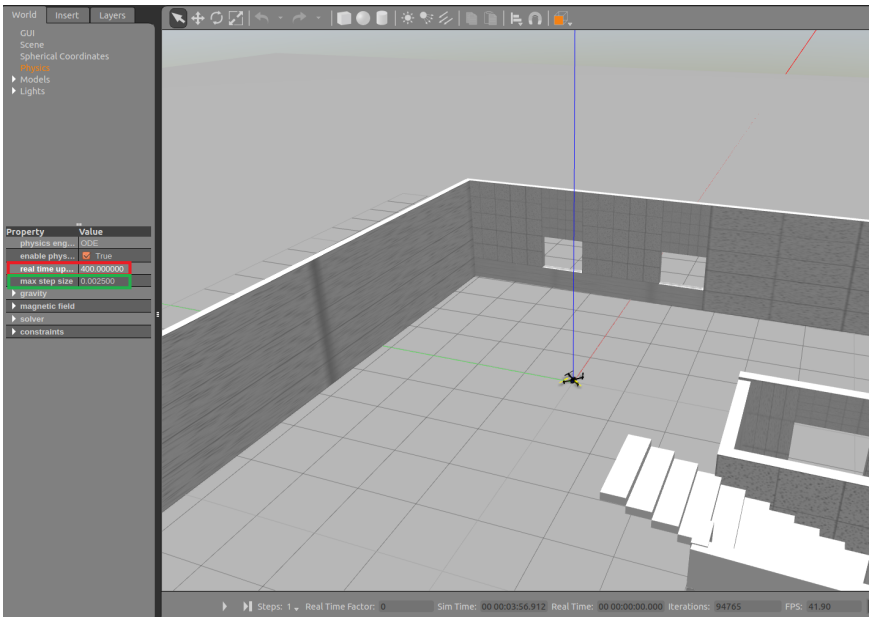


Figura 4.5: Ajuste del tiempo de simulación en *Gazebo*

Una vez ajustado, podremos iniciar las primeras pruebas de movimiento. En primer lugar, tendremos que empezar por cambiar el modo de vuelo a *GUIDED*, posteriormente armar los motores y realizar un despegue a la altura que se desee. Podemos hacer todo esto desde la terminal de *MAVProxy*, con los siguientes comandos:

```
mode GUIDED
arm throttle  --> arma los motores
takeoff 4     --> despega a la altura que indiquemos
```

Como podemos observar en la figura 4.6, en este caso de ejemplo despegamos a una altura de 4 metros.

Una vez despegado, podemos enviarle consignas de posición, mediante el comando *position x y z*, donde x, y, z representan la posición en cada uno de los tres ejes. Hay que tener en cuenta que el sistema de coordenadas que usa este comando es *NED [18] (North,East,Down)*. Es un sistema muy usado en aviación, en el cual la x representa la posición en el eje de latitud (positivo hacia el Norte), la y en el eje de longitud (positivo hacia el Este) y la z la posición vertical, siendo positiva hacia abajo para cumplir la regla de la mano derecha (con el pulgar indicando Norte). En el caso de nuestra simulación, está definido de tal manera que el dron está mirando hacia el Sur. Por tanto, usando este comando, con un incremento positivo de la coordenada x se movería hacia atrás (dirección Norte), mientras que con un incremento positivo de la coordenada y se movería hacia el Este. Sin embargo, hay que tener en cuenta que el comando está invertido, de forma que un incremento positivo de x hará mover al vehículo hacia el Sur, un incremento positivo en la y lo moverá hacia el Oeste, y un incremento positivo en la z lo moverá hacia arriba (cuando debería moverse hacia abajo por ser coordenadas *NED*). Podemos ver un ejemplo de uso en la figura 4.7. El comando lo enviamos de la siguiente forma, en la terminal de *MAVProxy*:

```
position 1 0 0
```

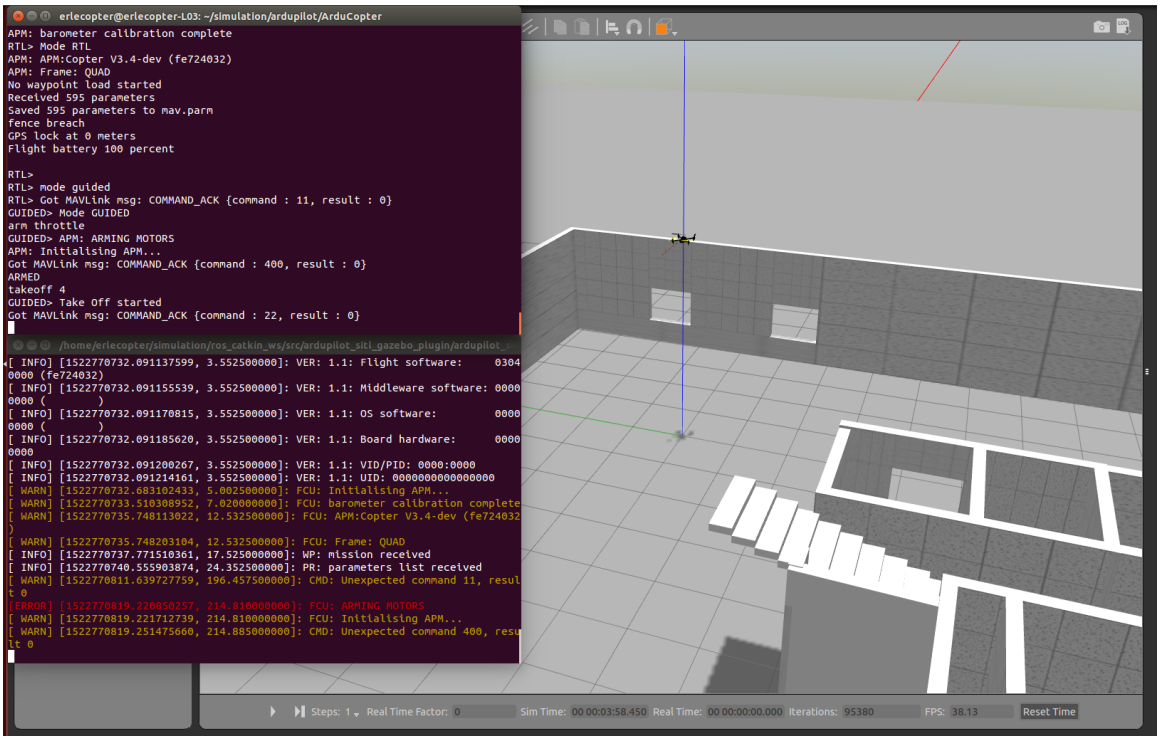


Figura 4.6: Ejemplo de despegue en *Gazebo* con *MAVProxy*

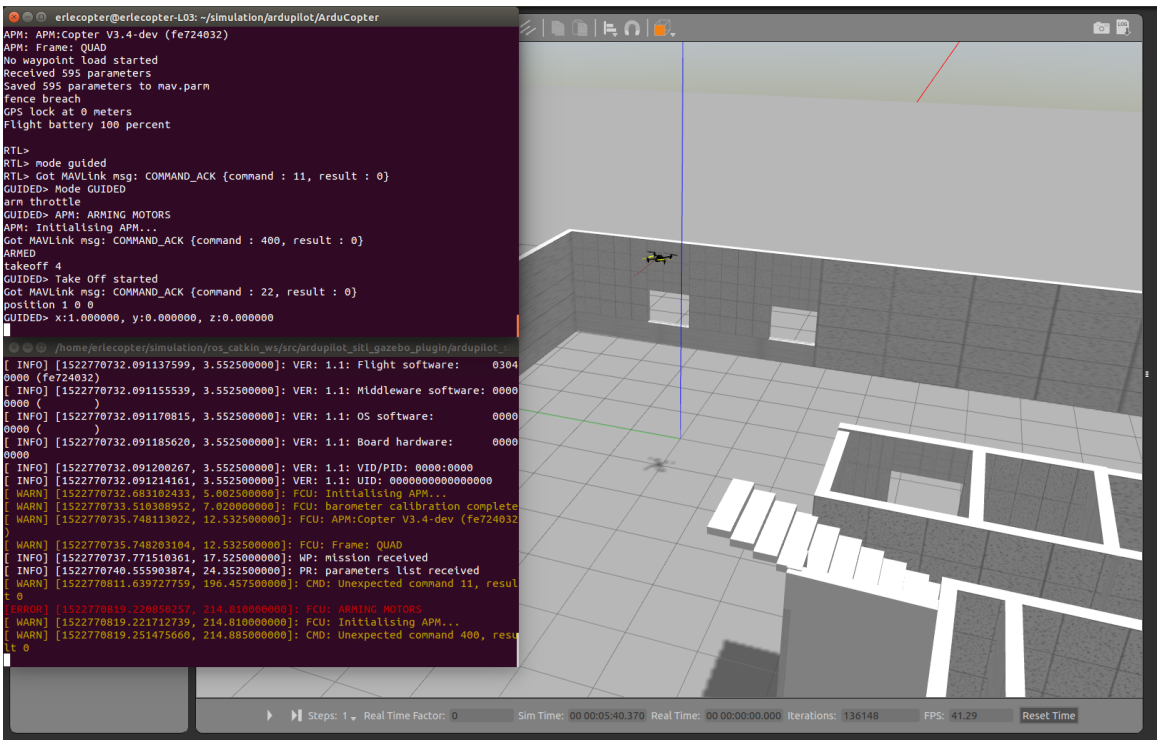


Figura 4.7: Ejemplo de uso del comando *position* de *MAVProxy*

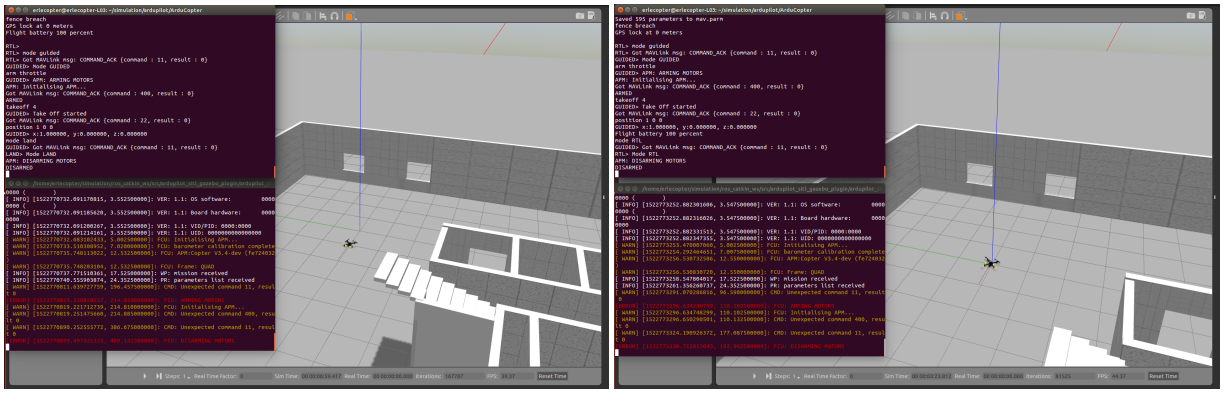
Para devolver el dron a tierra, tenemos dos opciones de aterrizaje: el modo *LAND* y el modo *RTL*, explicados anteriormente en el capítulo 3. Vamos a comprobar el funcionamiento de cada uno a continuación. En la figura 4.8a vemos un aterrizaje con modo *LAND*; el dron aterriza en la posición en la que está actualmente, solo modifica su posición en el eje vertical. Una vez detecta que está en tierra, desarma los motores. Por otro lado, en la figura 4.8b, vemos un ejemplo con el modo *RTL*. En este caso, el dron

vuelve a la posición donde fueron armados los motores, y una vez detecta la llegada a tierra, se desarma.

Para realizar cualquiera de los dos aterrizajes, basta con cambiar el modo de vuelo a *LAND* o *RTL*, según el caso, en la terminal de *MAVProxy*.

```
mode LAND
```

```
mode RTL
```



(a) Ejemplo con modo *LAND* (b) Ejemplo con modo *RTL*

Figura 4.8: Uso de los modos de aterrizaje en la simulación

La realización de estas primeras pruebas nos han servido para comprobar que todos los elementos de la simulación funcionan correctamente.

4.2 Simulación de trayectoria usando ROS

Tal y como vimos en la sección 4.1 , al lanzar la simulación, se inicializa *mavros*. Este crea una serie de *topics* a partir de la información de *ArduCopter*. Si abrimos una terminal nueva, con la simulación en ejecución, podemos visualizar la lista de *topics* que se crean:

```
rostopic list
```

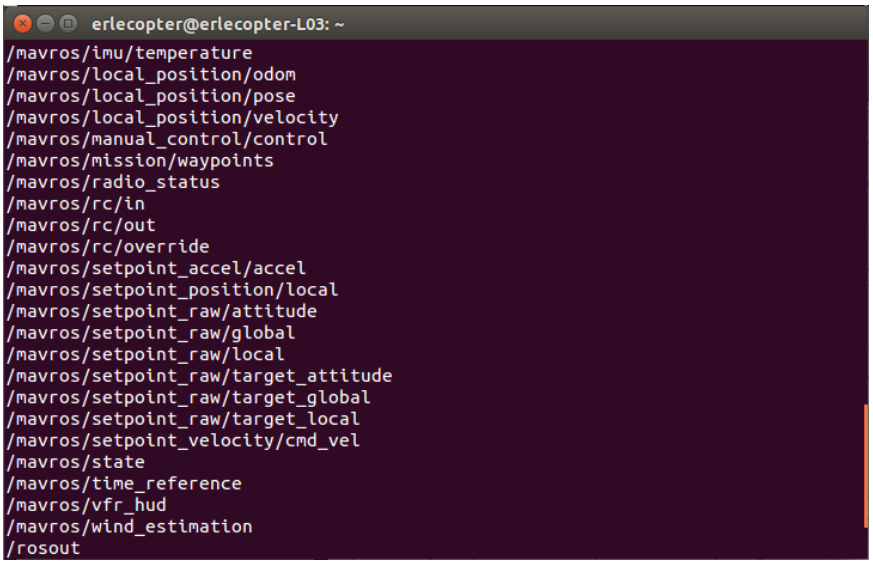


Figura 4.9: Lista de topics de *mavros*

Gracias a esto, podemos crear un nodo de *ROS* que se suscriba al *topic* de *mavros* que nos interese, consiguiendo de esa forma que el mensaje le llegue al autopiloto y podamos visualizarlo gráficamente en *Gazebo*. A continuación, vamos a ver un ejemplo en el cual creamos un nodo que publica una posición local en el topic */mavros/setpoint_position/local*, y veremos cómo esta información es recibida por el autopiloto, provocando el movimiento del dron en la simulación de *Gazebo*.

El sistema de coordenadas en la simulación está definido tal como vemos en la imagen 4.10, de tal forma que al enviarle una posición local se moverá con respecto a esa definición de ejes.

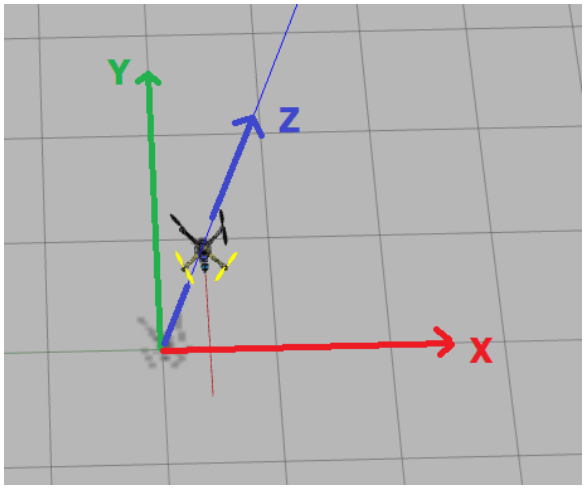


Figura 4.10: Coordenadas locales en la simulación

Para comenzar, vamos a crear un paquete *catkin* de *ROS* donde incluyamos nuestro código. Todo paquete posee tres características principales:

- archivo *package.xml*, que contiene meta información del paquete, como el nombre, número de versión, autores,etc.
- archivo *CMakeLists.txt*, es la entrada para el sistema de compilación de archivos, incluye instrucciones sobre la compilación del código y su instalación. Debe tener un formato ya definido para que los paquetes se ejecuten correctamente.
- cada paquete debe tener su directorio propio, aunque podemos tener varios paquetes dentro del mismo espacio de trabajo.

Para crear el paquete, el cual incluiremos en el espacio de trabajo realizado en B.4.1, vamos a usar *catkin_create_pkg*. El uso de este comando podemos verlo a continuación:

```
catkin_create_pkg <nombre_del_paquete> [dependencia1] [dependencia2] [dependencia3] . . .
```

En nuestro caso, el nombre del paquete de ejemplo será rombo (ya que es la forma de la trayectoria que describirá el dron en la simulación) y la dependencia necesaria para la creación de nuestro paquete será *mavros*.

Por tanto, debemos introducir el directorio del espacio de trabajo y después crear el paquete:

```
cd ~/ros_catkin_ws/src
catkin_create_pkg rombo mavros
```

Una vez creado, es necesario modificar el archivo *CMakeLists.txt*, para añadir los ejecutables necesarios para nuestro paquete. Debe quedar de la siguiente forma:

```
cmake_minimum_required(VERSION 2.8.3)
project(rombo)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)

## Generate added messages and services
generate_messages(DEPENDENCIES std_msgs)

## Declare a catkin package
catkin_package()

include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(rombo src/main.cpp)
target_link_libraries(rombo ${catkin_LIBRARIES})
add_dependencies(rombo rombo_generate_messages_cpp)

install(
        TARGETS rombo
        ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
        LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
        RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

El siguiente paso será crear un archivo de texto para escribir nuestro código. Lo haremos en el mismo directorio del espacio de trabajo; el nombre del archivo debe ser *main.cpp*, ya que es el que especificamos en el *CMakeLists.txt*. Si se quisiera poner otro nombre al archivo de texto, habría que modificarlo también en el *CMakeLists*.

A continuación, vamos a analizar paso a paso el código que se ha realizado. Este mismo se puede encontrar completo en el apéndice C.1.

En primer lugar, hay que añadir librerías y encabezados necesarios, además del tipo de mensajes que vamos a usar:

```
#include <cstdlib>
#include <ros/ros.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/CommandTOL.h>
#include <mavros_msgs/CommandInt.h>
#include <mavros_msgs/SetMode.h>
#include <geometry_msgs/PoseStamped.h>
```

Con la función *ros::init()* inicializamos *ROS*, además de definir el nombre de nuestro nodo, que en este caso es *mavros_takeoff*.

```
ros::init(argc, argv, "mavros_takeoff");
```

NodeHandle es el punto de acceso principal para las comunicaciones con el sistema de *ROS*. El *Rate* define la frecuencia de los bucles.

```
ros::NodeHandle n;
ros::Rate r(rate);
```

En el siguiente fragmento se crea un cliente para el servicio *mavros::SetMode*, que se encarga de cambiar el modo de vuelo. En este caso lo cambiamos a *GUIDED*. El objeto *ros::ServiceClient* es usado

para llamar al servicio. Si la llamada al servicio se ha realizado con éxito, la función *call()* será verdadera y el valor de *srv.response* será 1, mientras que si la llamada se realiza sin éxito el resultado será falso.

```
ros::ServiceClient cl = n.serviceClient<mavros_msgs::SetMode>("/mavros/set_mode");

mavros_msgs::SetMode srv_setMode;

srv_setMode.request.base_mode = 0;
srv_setMode.request.custom_mode = "GUIDED";

if(cl.call(srv_setMode))
{
ROS_INFO("modo de vuelo enviado ok; %d", srv_setMode.response.success);
}
else
  {
        ROS_ERROR("Failed SetMode");
        return -1;
  }
```

Continuamos con la creación de un cliente para el servicio de armado de motores *(arming service)*. Si configuramos *srv.request.value* para que sea verdadero (*true*), el vehículo tratará de armarse. Nuevamente, si la llamada al servicio se ha realizado con éxito, la función *call()* será verdadera y *srv.response.success* arrojará un 1 como resultado.

```
ros::ServiceClient arming_cl = n.serviceClient<mavros_msgs::CommandBool>("/mavros/cmd/
    arming");

mavros_msgs::CommandBool srv;

srv.request.value = true;

if(arming_cl.call(srv))
{
ROS_INFO("motores armados ok; %d", srv.response.success);
}
else
  {
        ROS_ERROR("Failed arming or disarming");
  }
```

Para despegar, se crea un cliente para el servicio de despegue *(takeoff service)*. Con el mensaje del servicio *srv_takeoff.request.altitude* podemos configurar la altura a la que deseamos subir. En este caso, se configura a 3 metros.

```
ros::ServiceClient takeoff_cl = n.serviceClient<mavros_msgs::CommandTOL>("/mavros/cmd/
    takeoff");

mavros_msgs::CommandTOL srv_takeoff;

srv_takeoff.request.altitude = 3;
srv_takeoff.request.latitude = 0;
srv_takeoff.request.longitude = 0;
srv_takeoff.request.min_pitch = 0;
srv_takeoff.request.yaw = 0;

if(takeoff_cl.call(srv_takeoff))
{
ROS_INFO("despegue enviado ok; %d", srv_takeoff.response.success);
```

```
}
else
{
        ROS_ERROR("Failed Takeoff");
}
```

Una vez despegado, usaremos la función *ros::Publisher* para publicar en el topic *mavros/setpoint_position/local* las posiciones en x, y, z que deseemos. En este caso, vemos en el código que mantenemos los 3 metros de altura del despegue y el dron se mueve 3 metros a lo largo del eje x. Los mensajes que se publican en este *topic* son *geometry_msgs* de tipo *PoseStamped*, que están definidos para determinar la posición de un punto en el espacio mediante x, y y z.

```
ros::Publisher local_pos_pub = n.advertise<geometry_msgs::PoseStamped>("mavros/
    setpoint_position/local",10);

geometry_msgs::PoseStamped pose;

pose.pose.position.x = 3;
pose.pose.position.y = 0;
pose.pose.position.z = 3;

ROS_INFO("Primer vertice");

for(int i = 100; ros::ok() && i>0; --i)
{
        local_pos_pub.publish(pose);
        ros::spinOnce();
        r.sleep();
}
```

Es posible publicar tantos puntos objetivo como se desee, siguiendo el mismo patrón mostrado en este fragmento anterior. En nuestro caso de ejemplo, escribimos los puntos necesarios para que el dron describa la figura de un rombo. Podemos ver todas esas posiciones en el código completo en el apéndice C.1.

Para concluir, se crea un cliente para el servicio de aterrizaje *(land service)*. El dron aterrizará y desarmará los motores cuando detecte la llegada a tierra.

```
ros::ServiceClient land_cl = n.serviceClient<mavros_msgs::CommandTOL>("/mavros/cmd/land");

mavros_msgs::CommandTOL srv_land;

srv_land.request.altitude = 3;
srv_land.request.latitude = 0;
srv_land.request.longitude = 0;
srv_land.request.min_pitch = 0;
srv_land.request.yaw = 0;

if(land_cl.call(srv_land))
{
ROS_INFO("land enviado ok %d", srv_land.response.success);
}else
 {
        ROS_ERROR("Failed Land");
 }
```

Finalmente, una vez tenemos nuestro código completo, compilamos el paquete con *catkin_make* en la terminal. Si posteriormente se introducen modificaciones en el código, es necesario volver a compilarlo

para que los cambios tengan efecto.

```
cd ~/ros_catkin_ws
catkin_make --pkg rombo
```

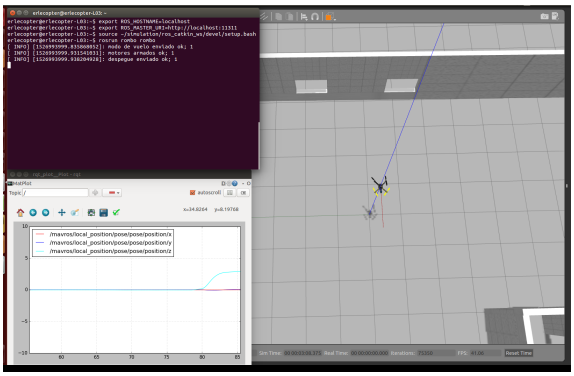
Para ejecutarlo, tendremos que tener la simulación iniciada, explicado anteriormente en la sección 4.1. En estas condiciones, abrimos una terminal, y ejecutamos el paquete haciendo uso de la herramienta *rosrun*:

```
rosrun rombo rombo
```

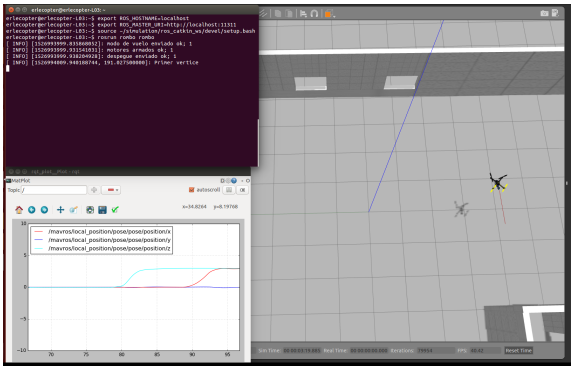
En la terminal en la que lo hemos ejecutado, podremos ir viendo la información de *ROS* según se van realizando los movimientos, informando del éxito de los mismos, o bien de algún fallo que pudiera ocurrir al realizar alguno de ellos, tal y como hemos programado.
A continuación, en la figura 4.11, se muestran imágenes de la simulación del dron. Se puede observar cómo va describiendo la trayectoria deseada en *Gazebo*. Haciendo uso de la herramienta de *ROS rqt_plot*, podemos visualizar los datos publicados en cualquier topic del sistema en forma de gráfica. En este caso, representamos el *topic /mavros/local_position/pose*, en el cual se publica el valor de x, y y z según avanza la simulación. Esto nos sirve para comprobar de forma sencilla que el dron se desplaza los metros que le corresponde en cada eje. Para usar esta herramienta, basta con abrir una nueva terminal y ejecutar lo siguiente:

```
rqt_plot
```

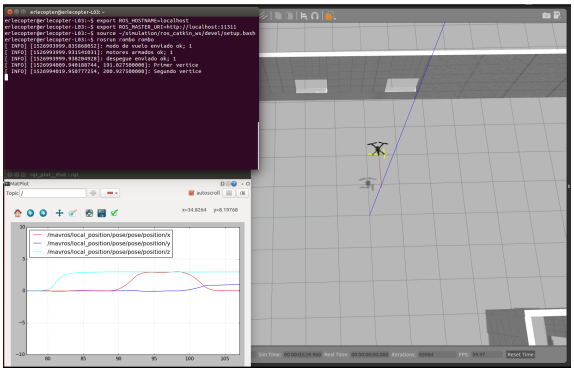
Una vez abierto, se especifica el *topic* del cual se quiere ver la información, en este caso elegimos */mavros/local_position/pose*, que muestra la posición local en los tres ejes.



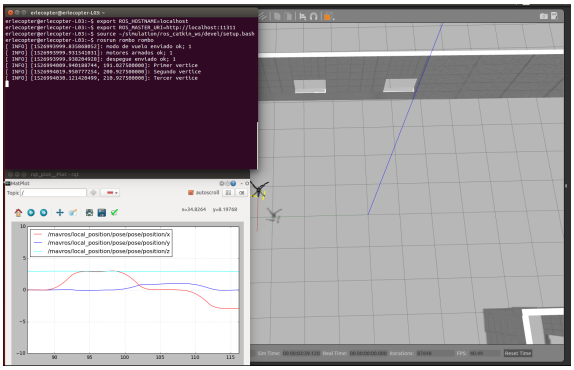
(a) Despegue a 3 metros de altura



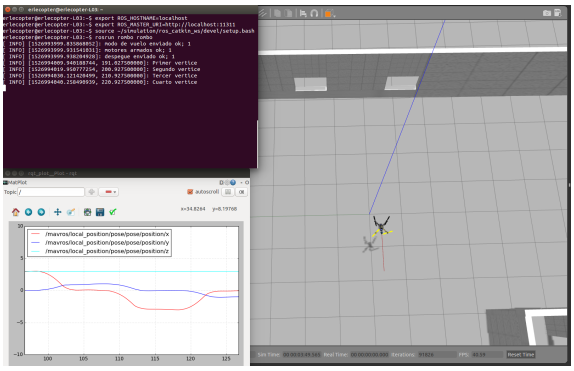
(b) Primer vértice del rombo (3 metros eje x)



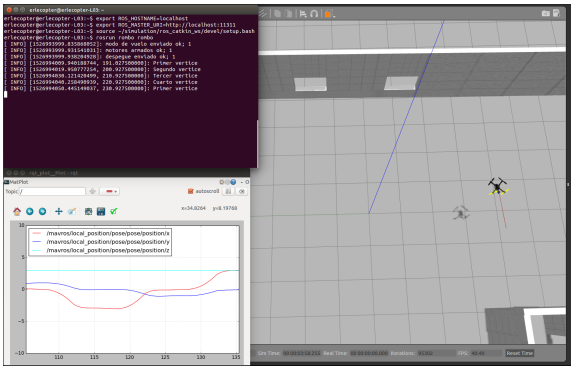
(c) Segundo vértice del rombo (1m eje y)



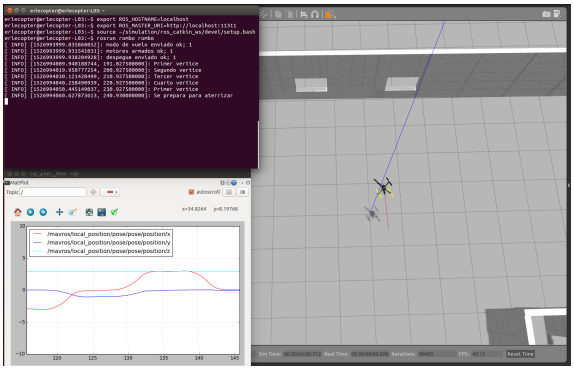
(d) Tercer vértice del rombo (-3m eje x)



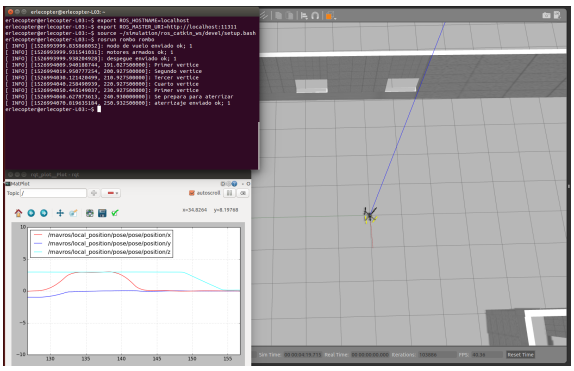
(e) Cuarto vértice del rombo (-1m eje y)



(f) Cierre de la figura (primer vértice)



(g) Vuelta al punto inicial



(h) Aterrizaje

Figura 4.11: Simulación de trayectoria autónoma

Capítulo 5

Resultados de la simulación

Para poder analizar cómo se ha comportado el vehículo en la simulación, son necesarias gráficas de resultados del vuelo, que nos muestren posición, altura, *roll*, *pitch* y *yaw*, aceleraciones en los tres ejes, velocidades, datos de sensores, etc. Es por esto que el autopiloto genera un archivo de datos de vuelo cada vez que el dron arma los motores, comienza a registrar datos de la sesión, y deja de hacerlo cuando el dron se desarma.

En este capítulo se analizarán los datos de vuelo de la simulación llevada a cabo en el capitulo anterior, y se examinará el comportamiento del mismo, gracias a las distintas gráficas que podremos crear con los datos registrados por el autopiloto. Se mostrará el directorio donde se almacenan los archivos del vuelo, además de diversas herramientas necesarias para poder manejar esos datos. Una vez analizado esto, modificaremos algún parámetro del *Erle-Copter* y lanzaremos una nueva simulación para evaluar su influencia en las gráficas de comportamiento.

5.1 Evaluación del comportamiento

5.1.1 Registro de datos de la simulación

Tras desarmar los motores del dron en la simulación, el autopiloto genera un archivo de datos de la sesión de vuelo. Estos archivos se denominan *logs*. Son de tipo *.bin*, y se almacenan en la carpeta de *ArduCopter*. El directorio completo es: *simulation/ardupilot/ArduCopter/logs*.

Estos archivos tienen diferentes campos, en cada uno de los cuales se almacena una información determinada [19]. A continuación vamos a ver los más significativos:

- *CTUN*: información sobre la altura del vehículo. Muestra la altura filtrada por el *EKF* (Alt), la altura sensada por el barómetro (BarAlt) y la consigna de altura, o altura deseada (DAlt).

- *NTUN*: información de navegación filtrada por el *EKF*, basándose en coordenadas *NED*, nombrando x al eje de latitud e y al eje de longitud. Muestra la posición en el eje de latitud (PosX), la posición en el eje de longitud (PosY), la velocidad en la dirección de latitud (VelX) y la velocidad deseada (DVelX), la velocidad en la dirección de longitud (VelY) y la velocidad deseada (DVelY). En el eje de latitud, la posición y velocidad serán positivas cuando se mueva con dirección Norte, mientras que en el de longitud, la posición y velocidad serán positivas cuando se mueva hacia el Este.

- *ATT*: información de orientación (*attitude*). Se refiere a la información de ángulos en los tres ejes, *roll, pitch y yaw*. Muestra el *roll* actual del vehículo (Roll), el ángulo *roll* deseado (DesRoll), el *pitch* actual del vehículo (Pitch), el ángulo *pitch* deseado (DesPitch), el *yaw* actual del vehículo (Yaw), y el ángulo *yaw* deseado (DesYaw).
- *IMU*: información sobre los acelerómetros y giróscopos. Muestra los valores del acelerómetro en los tres ejes (AccX, AccY, AccZ) y la velocidad de la rotación de los giróscopos en los tres ejes (GyrX, GyrY, GyrZ).
- *RCOU*: información de las señales *pwm* enviadas a los motores. Muestra la salida *pwm* enviada por el controlador de vuelo a cada motor (Ch1, Ch2, Ch3 Ch4).

Una vez hemos visto cómo se registran los datos del autopiloto, vamos a tratar de representarlos gráficamente para poder analizar el rendimiento del *UAV* durante el vuelo.

5.1.2 Gráficas de datos del vuelo

Los archivos *log*, según el formato que tienen inicialmente, no son fácilmente manejables para el usuario. Es por ello que el paquete *MAVLink* proporciona una serie de herramientas [20] que pueden ser utilizadas para analizar estos archivos después del vuelo. Las que han resultado más útiles en el desarrollo de este proyecto han sido *MAVGraph* y *MAVTomfile*, de las cuales vamos a ver ejemplos de uso a continuación. Para poder usarlas, debemos estar en el directorio en el que se encuentran los archivos *log*:

```
cd ~/simulaction/ardupilot/ArduCopter/logs
```

MAVGraph nos permite visualizar gráficamente cualquier dato del vuelo. Su uso es el siguiente:

```
mavgraph.py archivo.bin "datos a visualizar"
```

Por ejemplo, si se desea ver la altura filtrada, deseada y la del barómetro del archivo.bin, se haría de la siguiente forma:

```
mavgraph.py archivo.bin "CTUN.Alt" "CTUN.DAlt" "CTUN.BarAlt"
```

Obtenemos como resultado una gráfica como la mostrada en la figura 5.1.

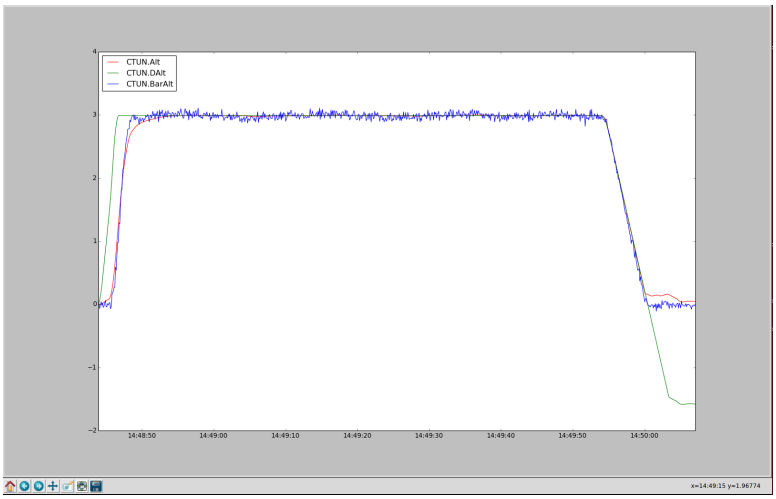


Figura 5.1: Gráfica obtenida con la herramienta *MAVGraph*

En algunos casos, es preferible usar un analizador de gráficas más completo, como es el caso de *MATLAB*. Para ello, es muy útil la otra herramienta mencionada anteriormente, *MAVTomfile*. Esta

herramienta nos permite convertir un archivo tipo *log* a un archivo .m de *MATLAB*. Para convertirlo, debemos ejecutar lo siguiente en la terminal:

```
mavtomfile.py archivo.bin
```

El archivo .m se crea en el mismo directorio donde se almacenan los archivos *log*. En nuestro caso, le ponemos de nombre *datos_rombo.m*. Está formado por un conjunto de estructuras que contienen todos los datos del vuelo, y se cargan en el *workspace* de *MATLAB* al ejecutarlo. Para representar gráficamente los datos que deseemos, es necesario crear un pequeño *script* de *MATLAB*, el cual podemos ver a continuación.

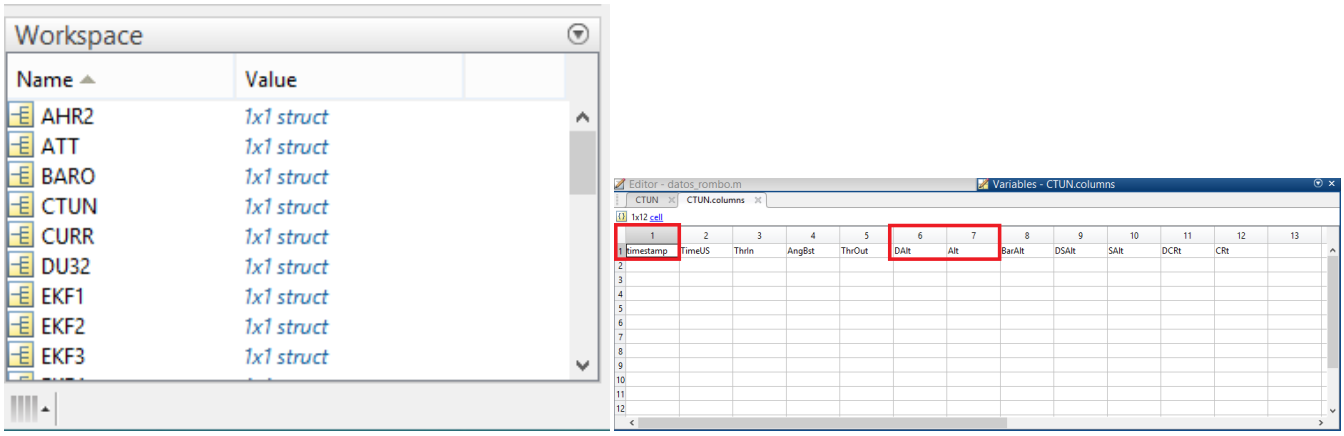
```
clear all
run datos_rombo.m

%----------------CTUN----------------
figure;
set(gcf,'Name','ALTURA DESEADA Y FILTRADA')

plot(CTUN.data(:,1), CTUN.data(:,6),'g')
hold on;
plot (CTUN.data(:,1), CTUN.data(:,7),'r')

title ('Altura deseada y filtrada, en metros')
legend ('Altura deseada','Altura filtrada')
```

En este caso, por ejemplo, representamos la altura deseada y la altura filtrada. Como vemos en el código anterior, seleccionamos los datos de *CTUN* de la columna 1, que es la base de tiempos, y los representamos en el eje x de la gráfica, y también seleccionamos los datos de la columna 6, que es la altura deseada, y lo representamos en el eje y. Para representar la altura filtrada se sigue la misma estrategia; hacemos otra gráfica, en la que seleccionamos los datos de la columna 7 en lugar de la 6. Para saber qué datos están presentes en cada columna de la estructura, basta con hacer doble click sobre ella en el *workspace* de *MATLAB*, y podremos ver los datos contenidos en cada columna, como en la figura 5.2.



(a) Estructuras de datos en el *workspace* de *MATLAB* (b) Significado de cada columna

Figura 5.2: Estructuras de datos y significado de columnas en *MATLAB*

Para representar el resto de gráficas, se seguirá este mismo patrón en el código, pero sustituyendo la estructura de datos de *CTUN* por la que corresponda en cada caso, de los mencionados en la sección 5.1.1. El *script* de *MATLAB* completo para obtener todas las gráficas está presente al final del documento, en C.2.

Una vez que tenemos representadas las gráficas de datos, es el momento de examinarlas. La sesión de vuelo que analizaremos es la descrita anteriormente en la sección 4.2, en la cual el *UAV* describía una trayectoria con forma de rombo. En las gráficas de cada uno de los ensayos que se mostrarán posteriormente, el eje de tiempos corresponde al tiempo de simulación en *Gazebo*, por tanto el instante en el cual comienzan las gráficas corresponde a los segundos transcurridos desde que se lanzó la simulación hasta que se ejecuta el código de la trayectoria.

Comenzaremos con la gráfica de la altura, la cual podemos ver en la figura 5.3.

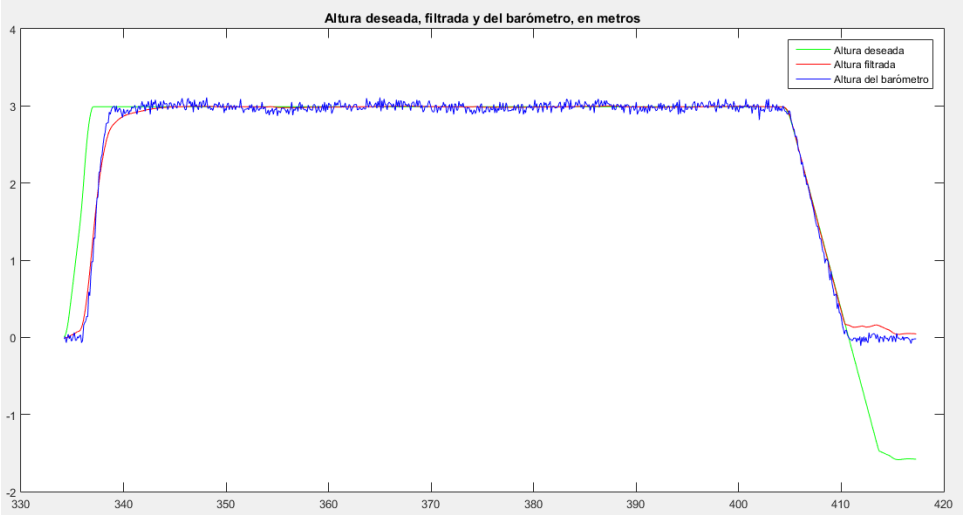


Figura 5.3: Altura del dron

Se puede observar como la consigna de altura son 3 metros, desde el despegue hasta el aterrizaje del dron. Vemos que tanto la altura filtrada como la registrada por el barómetro son bastante parecidas entre sí, y a su vez, siguen con bastante exactitud la señal de altura deseada, lo que nos hace concluir que el dron se mantiene bastante estable en cuanto a altura se refiere. En el tramo inicial de la trayectoria, se observa un pequeño retardo desde que se envía la consigna de altura hasta que el vehículo comienza a subir. En el tramo final, vemos que la consigna de altura desciende incluso por debajo de 0 hasta que el vehículo detecta que ha aterrizado, momento en el que se desarman los motores y se termina el registro de datos.

A continuación, observaremos las gráficas de posición y velocidades del dron. Como ya se comentó en la sección 5.1.1, esta información es registrada basándose en los ejes de latitud y longitud. Podemos ver cómo están definidos los puntos cardinales en el entorno de simulación en la figura 5.4. El dron está orientado mirando hacia el Sur, tal y como se comentó anteriormente.

Las gráficas de posición y velocidades están presentes en las figuras 5.5 y 5.6.

Vemos que cuando el dron se mueve en el eje de latitud con dirección al Norte, registra incrementos positivos en la posición, mientras que hacia el Sur son negativos. En el eje de longitud, son positivos al moverse hacia el Este, tal y como esperábamos. Podemos observar que la posición real (filtrada por el *EKF*) es muy parecida a la consigna de posición en ambos ejes; el dron sigue bastante fielmente los movimientos que se esperaban horizontalmente.

En cuanto a las gráficas de velocidad, se adaptan bastante bien a las consignas, tal y como era de esperar. Las posiciones que hemos analizado anteriormente eran bastante precisas, y eso es consecuencia de un buen seguimiento de la velocidad deseada por parte del dron.

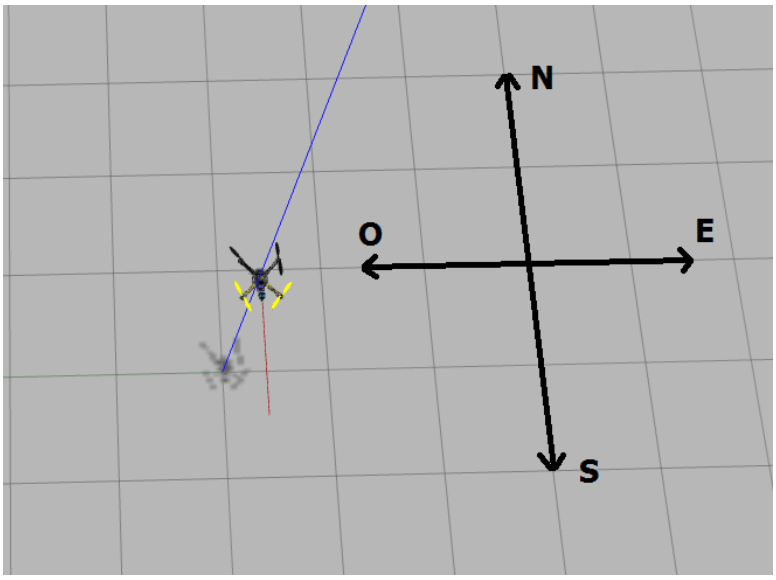
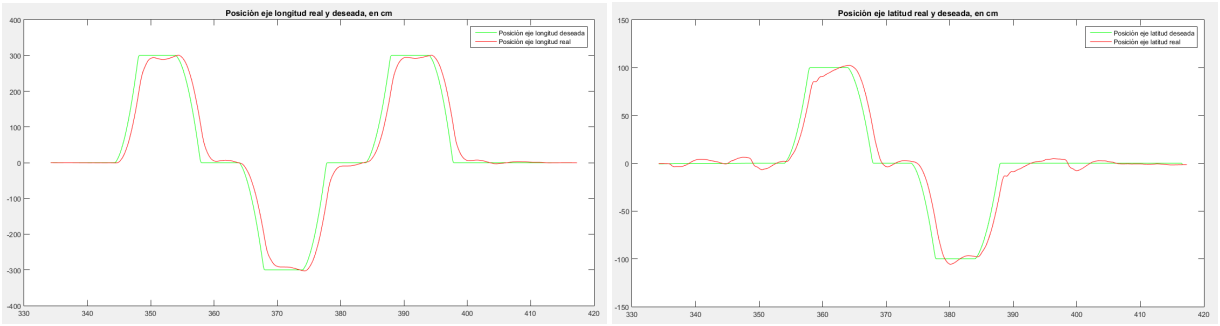
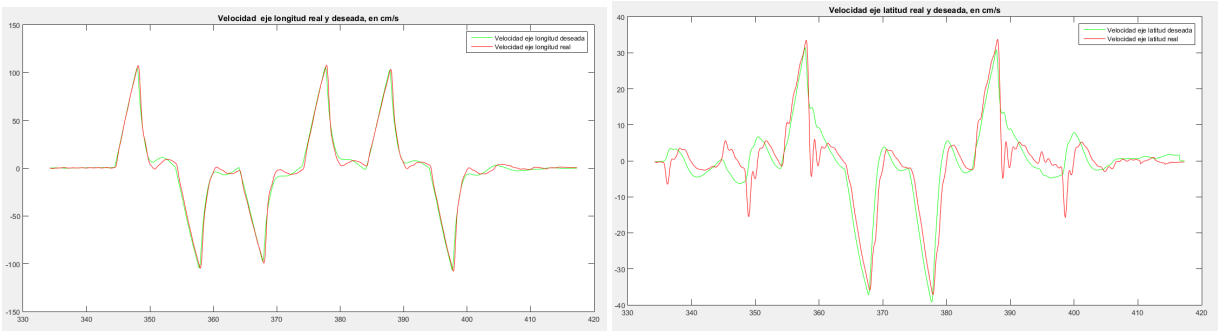


Figura 5.4: Definición de puntos cardinales en la simulación



(a) Posiciones real y deseada en el eje de longitud (b) Posiciones real y deseada en el eje de latitud

Figura 5.5: Posición del dron en los ejes de latitud y longitud

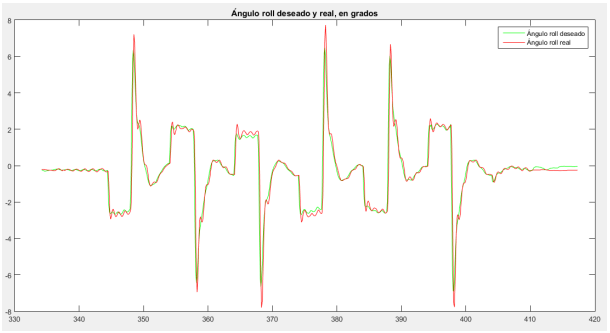


(a) Velocidades real y deseada en el eje de longitud (b) Velocidades real y deseada en el eje de latitud

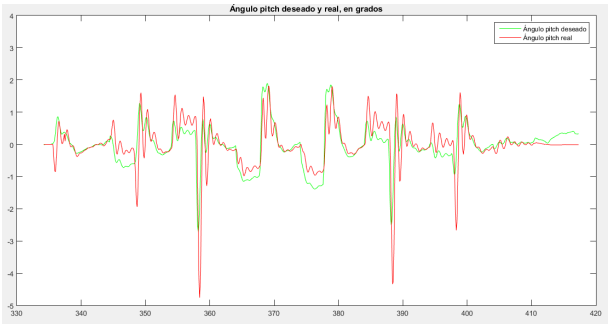
Figura 5.6: Velocidad del dron en los ejes de latitud y longitud

Continuaremos con las gráficas de los ángulos en los tres ejes, que podemos ver en la figura 5.7.

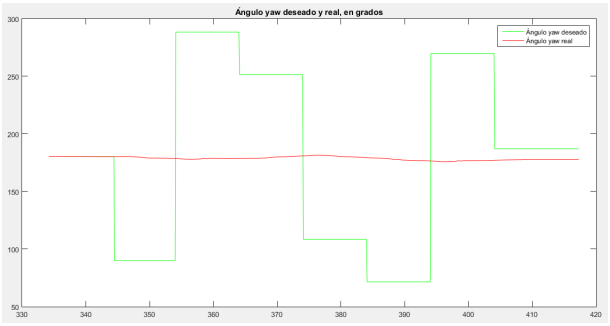
En las gráficas del *roll* y *pitch* se puede apreciar como el dron sigue de forma precisa las consignas en todo momento, lo que hace que sea muy estable en el vuelo. La gráfica del *yaw* es más peculiar. Vemos que no cambia su orientación en ningún momento, se queda fijo en 180º (0º sería mirando hacia el Norte). Las consignas enviadas son debidas a que el dron intenta apuntar con su parte delantera hacia la posición a la que viaja en cada momento. Sin embargo, en la simulación no es posible modificar el *yaw*, por eso el



(a) Ángulo *roll* deseado y real



(b) Ángulo *pitch* deseado y real

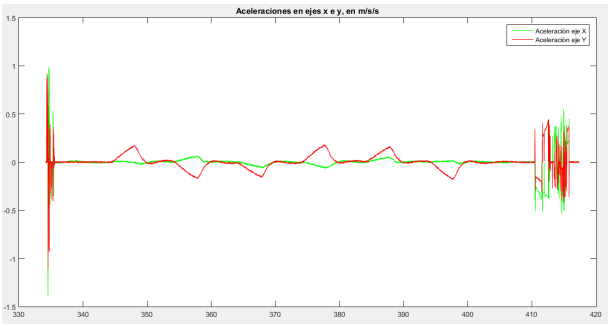


(c) Ángulo *yaw* deseado y real

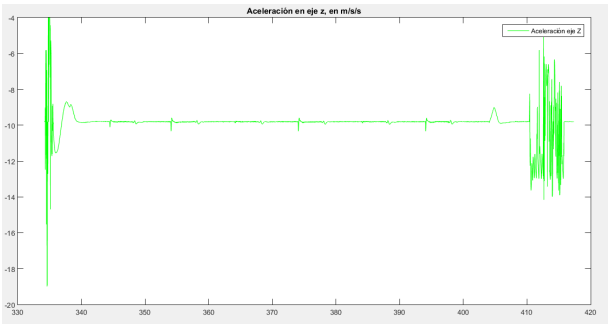
Figura 5.7: Ángulos en los tres ejes del dron

yaw real se mantiene fijo y no sigue a la consigna de *yaw* deseado.

En la figura 5.8 podemos ver las aceleraciones en los tres ejes que registra el acelerómetro.



(a) Aceleración ejes x e y



(b) Aceleración eje z

Figura 5.8: Aceleraciones en los tres ejes registradas por el acelerómetro

Estos valores registrados están contaminados por las vibraciones en cada uno de los ejes. Si aparecen valores muy altos, el dron puede tener problemas de estabilidad debido a la mala estimación de su altura y posición, provocando muchas dificultades para mantenerse en un punto en el aire. En los ejes x e y, los valores aceptables [21] deben estar en el rango de -3 a +3 m/s^2, mientras que en el eje z el valor aceptable sería entre -15 y -5 m/s^2. Por encima o debajo de estos valores las vibraciones pueden llevar al dron a comportamientos erróneos. Como vemos en las gráficas de la figura 5.8, las mayores vibraciones suceden en el armado y desarmado de los motores, lo cual entra dentro de la normalidad. Durante el vuelo, vemos que en ningún momento se superan los rangos de valores que hemos comentado, por lo tanto podemos concluir que los acelerómetros de la *IMU* simulada en el *Erle-Copter* están funcionando de forma correcta.

Para finalizar, en la figura 5.9, se muestran las señales *pwm* enviadas a cada uno de los motores por

parte del controlador de vuelo.

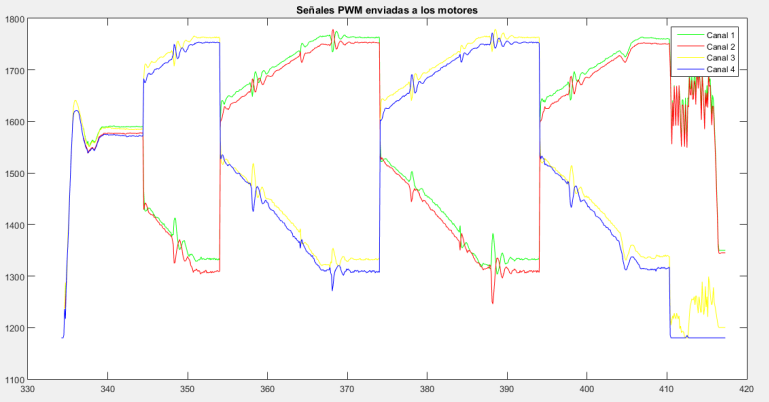


Figura 5.9: Señales *pwm* enviadas a los motores

Cada canal representa la señal enviada a cada motor. Vemos que la señal *pwm* tiene un rango de valores entre 1100 y 1900 aproximadamente. Al inicio, las señales de los canales están al mínimo. Cuando se requiere un despegue, vemos como se incrementan conjuntamente las señales de los cuatro motores. Una vez alcanza la altura de consigna, envía señales en torno a 1500-1600 para mantener la posición en esa altura. Para realizar los movimientos horizontalmente, alterna la velocidad de los motores para conseguir desplazarse hacia una dirección u otra. El controlador de vuelo puede manejar la rotación de *roll* y *pitch* acelerando dos motores de un lado y decelerando los otros dos. Por ejemplo, para desplazarse hacia la izquierda incrementará la velocidad de los dos motores del lado derecho del vehículo y los dos motores de la izquierda girarán a menor velocidad. De la misma manera, si quiere moverse hacia delante, acelera los dos motores traseros y los dos delanteros girarán a menor velocidad. Al finalizar, se aprecia el desarme de los motores cuando envía una señal *pwm* de valor en torno a 1200-1300 a cada uno.

5.2 Variación de parámetros del Erle-Copter

En la sección anterior, hemos analizado las gráficas de datos de una sesión de vuelo, bajo los parámetros por defecto del modelo de *Erle-Copter*, sus parámetros nominales. A continuación, aplicaremos algún cambio a estos parámetros, tanto físicos como del controlador de vuelo, para comprobar cuál es el efecto de estas modificaciones en el vuelo de nuestro *UAV*. Para ello, volveremos a simular la misma trayectoria del apartado 4.2 y trataremos de comparar las gráficas de resultados que obtengamos con las anteriores en busca de diferencias.

5.2.1 Parámetros físicos

Como vimos en el capítulo 3, las propiedades físicas del *Erle-Copter* en la simulación están presentes en el archivo *erlecopter.xacro*, el cual es modificable por el usuario.

En la imagen 5.10 podemos ver de nuevo estos parámetros, de los cuales se han elegido dos que se modificarán para comprobar su influencia: la masa del vehículo (*mass*) y la masa de los rotores (*mass_rotor*). Ambas modificaciones podrían suceder en un caso real; por ejemplo, si el dron tuviese que transportar una carga, aumentaría la masa del vehículo. De la misma forma, si fueran necesarios rotores más potentes en algún caso determinado, estos podrían tener una masa mayor que la actual.

En primer lugar, comenzaremos con un ensayo en el que variamos el parámetro *mass* a 1.6 kg, es decir, añadimos medio kilogramo a su valor por defecto. Es un cambio significativo, ya que supone una

```
<!-- Properties -->
<xacro:property name="namespace" value="erlecopter" />
<xacro:property name="rotor_velocity_slowdown_sim" value="10" />
<xacro:property name="mesh_file" value="erlecopter.dae" />
<xacro:property name="mesh_scale" value="1 1 1"/> <!-- 1 1 1 -->
<xacro:property name="mesh_scale_prop" value="1 1 1"/>
<xacro:property name="mass" value="1.1" /> <!-- [kg] -->
<xacro:property name="body_length" value="0.18" /> <!-- [m] 0.10 -->
<xacro:property name="body_width" value="0.12" /> <!-- [m] 0.10 -->
<xacro:property name="body_height" value="0.10" /> <!-- [m] -->
<xacro:property name="mass_rotor" value="0.005" /> <!-- [kg] -->
<xacro:property name="arm_length_front_x" value="0.141" /> <!-- [m] 0.1425  0.22 -->
<xacro:property name="arm_length_back_x" value="0.141" /> <!-- [m] 0.154  0.22 -->
<xacro:property name="arm_length_front_y" value="0.141" /> <!-- [m] 0.251  0.22 -->
<xacro:property name="arm_length_back_y" value="0.141" /> <!-- [m] 0.234  0.22 -->
<xacro:property name="rotor_offset_top" value="0.030" /> <!-- [m] 0.023-->
<xacro:property name="radius_rotor" value="0.12" /> <!-- [m] -->
<xacro:property name="motor_constant" value="8.54858e-06" /> <!-- [kg.m/s^2] -->
<xacro:property name="moment_constant" value="0.016" /> <!-- [m] -->
<xacro:property name="time_constant_up" value="0.0125" /> <!-- [s] -->
<xacro:property name="time_constant_down" value="0.025" /> <!-- [s] -->
<xacro:property name="max_rot_velocity" value="838" /> <!-- [rad/s] -->
<xacro:property name="sin30" value="0.5" />
<xacro:property name="cos30" value="0.866025403784" />
<xacro:property name="sqrt2" value="1.4142135623730951" />
<xacro:property name="rotor_drag_coefficient" value="8.06428e-05" />
<xacro:property name="rolling_moment_coefficient" value="0.000001" />
<xacro:property name="color" value="DarkGrey" />
```

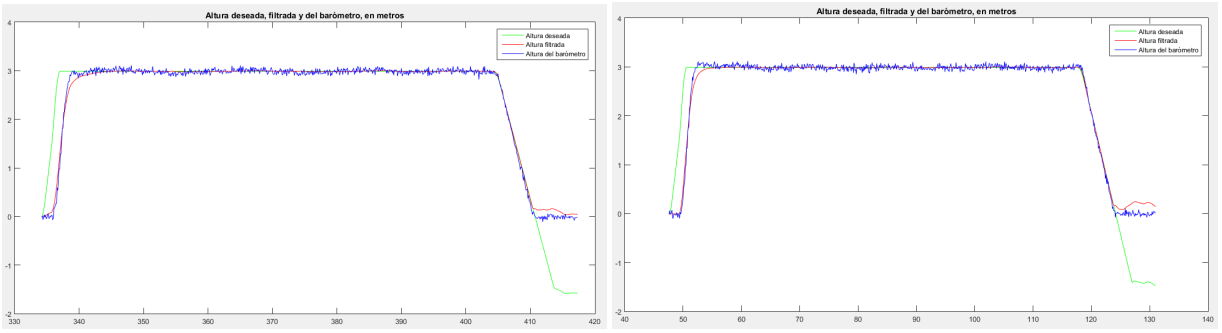
Figura 5.10: Parámetros físicos modificables del *Erle-Copter*

masa próxima al 50 % mayor que la nominal. A continuación comprobaremos su influencia con la ayuda de las gráficas post-vuelo.

En cuanto al parámetro *mass_rotor*, modificaremos su valor a 0.01 kg. Es decir, cada rotor tendrá una masa de 10 gramos, el doble de su valor por defecto. Posteriormente comprobaremos si esta modificación puede llegar a ser significativa en el vuelo del *UAV*.

- **Ensayo con la masa modificada a 1.6 kg**

  Comenzaremos comparando las gráficas de altura de ambos ensayos, en la figura 5.11. Vemos que en el ensayo anterior, con la masa nominal, el dron mantenía la altura de forma correcta hasta que se requería el aterrizaje. En este nuevo ensayo sucede algo muy similar, por tanto no hay modificación de ningún tipo en cuanto a altura se refiere.

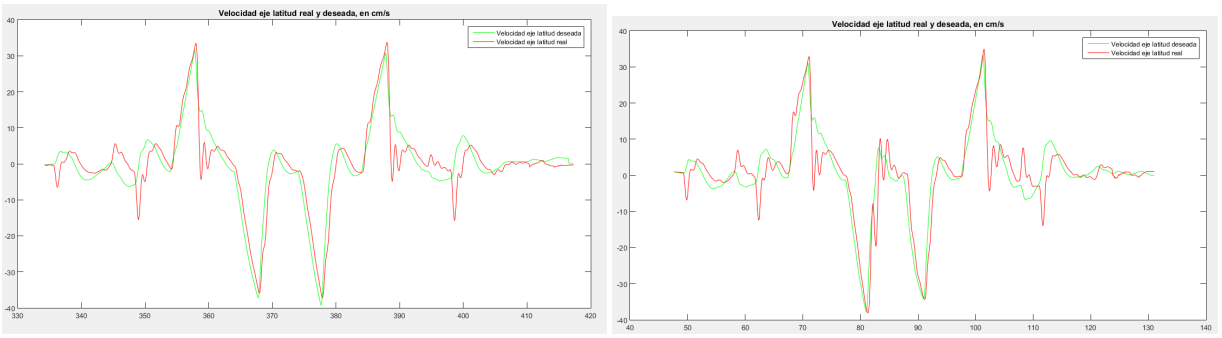


(a) Gráfica de altura con la masa nominal (b) Gráfica de altura con la masa modificada a 1.6 kg

Figura 5.11: Comparación de gráficas de altura en los ensayos nominal y con masa modificada

Analizaremos ahora las gráficas de velocidad de ambos ensayos. Dado que las velocidades en ambos ejes tienen un perfil muy parecido, elegimos las velocidades en el eje de latitud, por ejemplo. Podemos verlas en la figura 5.12. Si nos fijamos en la gráfica del ensayo con parámetros nominales, vemos como

la velocidad seguía a la consigna de forma bastante precisa, sin cambios excesivamente bruscos. En este nuevo ensayo con la masa modificada también sucede así. Sin embargo, en algunos instantes podemos ver cambios repentinos de la velocidad en instantes pequeños de tiempo, lo que se traduce en "picos" en la gráfica, que podemos ver por ejemplo en torno al segundo 85 en la gráfica de este nuevo ensayo. Estas variaciones, a pesar de que el dron pudo terminar la trayectoria sin problemas, causan una apariencia más inestable del vehículo durante esos instantes del vuelo.

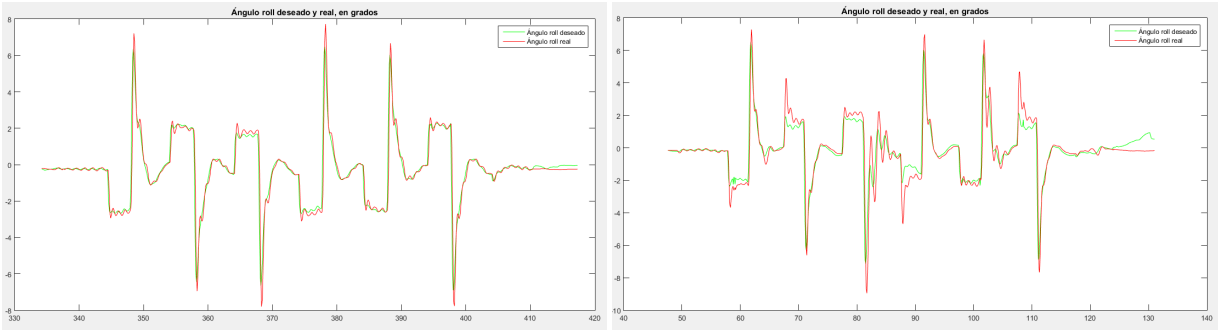


(a) Gráfica de velocidad en el eje de latitud con masa nominal

(b) Gráfica de velocidad en el eje de latitud con la masa modificada a 1.6 kg

Figura 5.12: Comparación de gráficas de velocidades en los ensayos nominal y con masa modificada

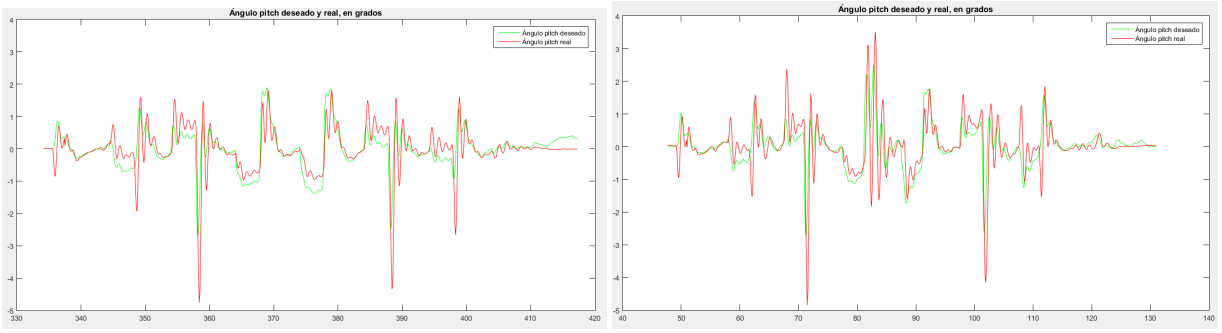
A continuación, vamos a analizar los ángulos *roll y pitch* de ambos ensayos.



(a) Gráfica de ángulo *roll* con masa nominal

(b) Gráfica de ángulo *roll* con masa modificada a 1.6 kg

Figura 5.13: Comparación de gráficas de ángulo *roll* en los ensayos nominal y con masa modificada



(a) Gráfica de ángulo *pitch* con masa nominal

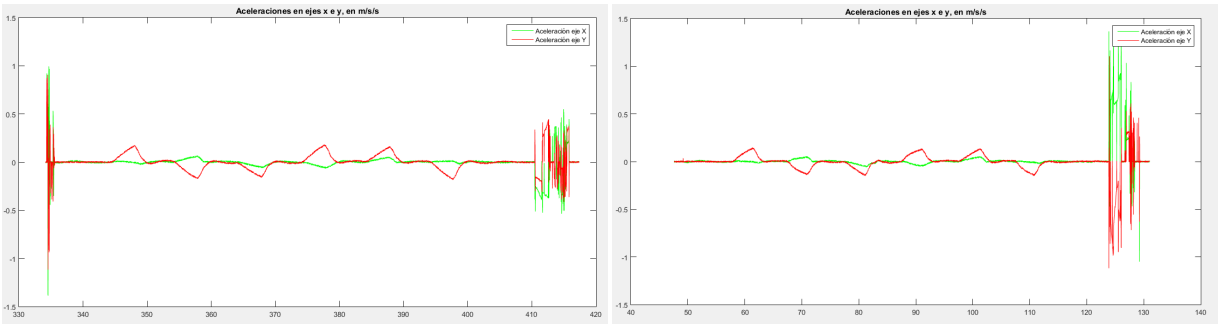
(b) Gráfica de *pitch* con masa modificada a 1.6 kg

Figura 5.14: Comparación de gráficas de ángulo *pitch* en los ensayos nominal y con masa modificada

En las figuras 5.13 y 5.14 podemos ver las cuatro gráficas. Como podemos observar, sucede algo parecido a lo que veíamos anteriormente en la gráfica de velocidad: los ángulos *roll y pitch* presentan

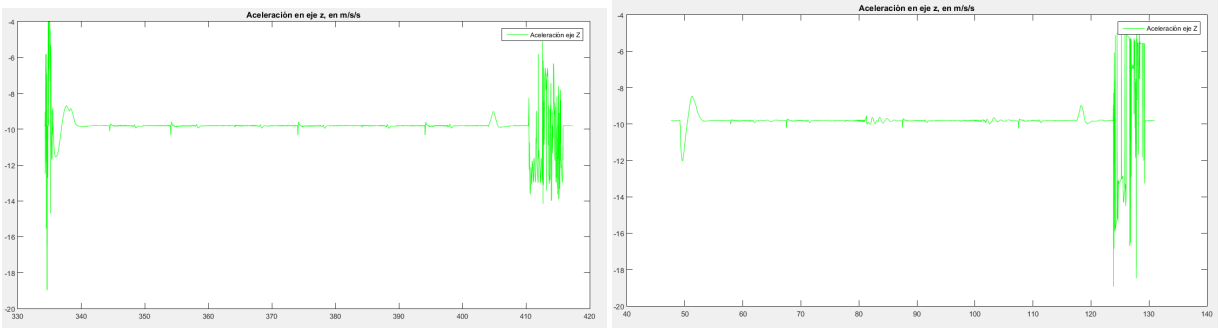
más oscilaciones en ciertos instantes en las gráficas del ensayo con la masa modificada, dado que al haber cambios bruscos de velocidad, como vimos anteriormente, los ángulos de giro sobre los ejes también tienden a ser más inestables, y les cuesta más adaptarse a la consigna. Por ejemplo, en la gráfica 5.14, se observa que mientras en el ensayo anterior el ángulo *pitch* en ningún momento llegaba a superar los 2º, en este nuevo ensayo hay "picos" que casi alcanzan los 4º de inclinación en algún instante, el doble de inclinación que en el ensayo anterior, lo cual puede ser significativo durante el vuelo.

Por último, visualizaremos las gráficas de aceleraciones medidas por el acelerómetro (figuras 5.15 y 5.16). Comprobamos que los valores de aceleraciones en los tres ejes durante ambos ensayos, en general, son bastante parecidos. En este nuevo ensayo tampoco se superan los límites para estas aceleraciones comentados anteriormente.



(a) Gráfica de aceleraciones con masa nominal (b) Gráfica de acel. con masa modificada a 1.6 kg

Figura 5.15: Comparación de gráficas de aceleraciones en ejes x e y en los ensayos nominal y con masa modificada



(a) Gráfica de aceleración con masa nominal (b) Gráfica de aceleración con masa modificada a 1.6 kg

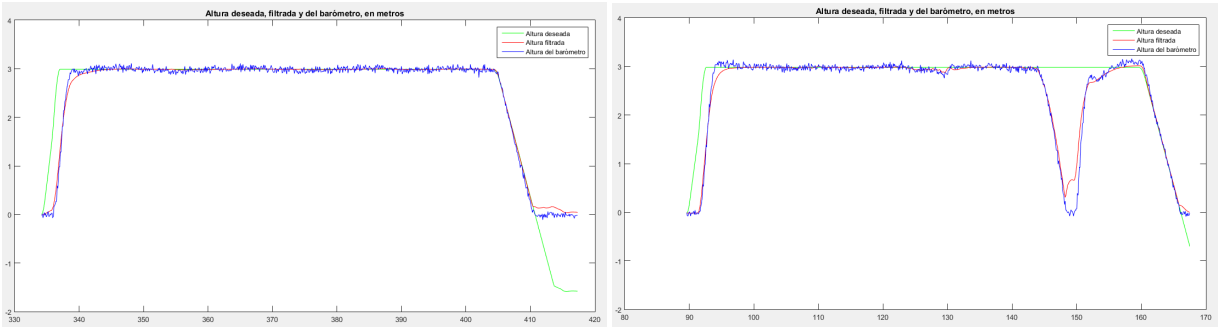
Figura 5.16: Comparación de gráficas de aceleraciones en el eje z en los ensayos nominal y con masa modificada

Cabe destacar que durante una nueva realización de este ensayo con la masa modificada surgió una circunstancia peculiar que puede llegar a suceder durante la simulación, por este motivo vamos a analizar las gráficas de este nuevo ensayo.

Podemos verlas en las figuras 5.17, 5.18 y 5.19 . Vemos que hay varias fases en las que al dron le cuesta mantener la altura. Por ejemplo, en el segundo 130 de la gráfica se puede observar como la altura filtrada y la del barómetro están unos metros por debajo de la altura de consigna, en concreto pierde unos 20 cm de altura en ese punto. El momento más crítico sucede en torno al segundo 150, en el que el dron comienza a perder altura, hasta llegar a los 36 cm por encima del suelo cuando la consigna en ese momento seguía siendo de 3 metros. Sin embargo, vemos que consigue recuperarse

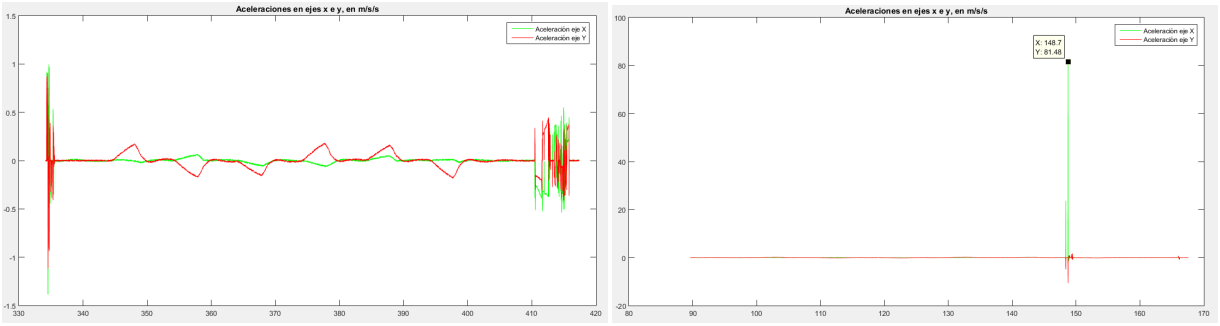
para luego efectuar un aterrizaje sin problemas.

En las gráficas de aceleración, hay un instante de tiempo en el que se disparan las medidas en este nuevo ensayo, en torno al segundo 150. Como vemos en las figuras, en el caso de la aceleración en el eje x llega hasta los 81.48 m/s^2, mientras que en el eje z se dispara hasta unos -98 m/s^2. Este instante coincide precisamente con el momento en que el dron perdía altura de forma inesperada, lo cual hemos visto en la gráfica de altura. Posiblemente una desconexión repentina del autopiloto en la simulación provocó esta falla de medida por parte del acelerómetro y puede ser el causante de una pérdida de metros tan mayúscula en ese instante.



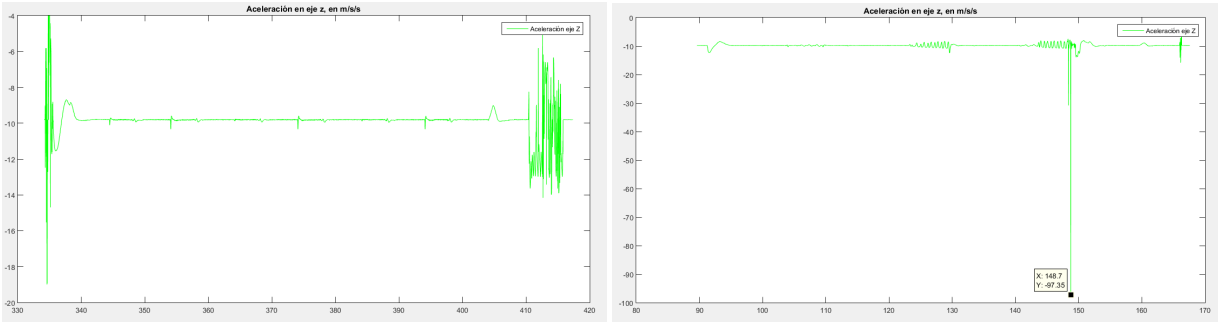
(a) Gráfica de altura con la masa nominal (b) Gráfica de altura con la masa modificada a 1.6 kg

Figura 5.17: Comparación de gráficas de altura en los ensayos nominal y con masa modificada con pérdida de altura



(a) Gráfica de aceleraciones con masa nominal (b) Gráfica de acel. con masa modificada a 1.6 kg

Figura 5.18: Comparación de gráficas de aceleraciones en ejes x e y en los ensayos nominal y con masa modificada con pérdida de altura

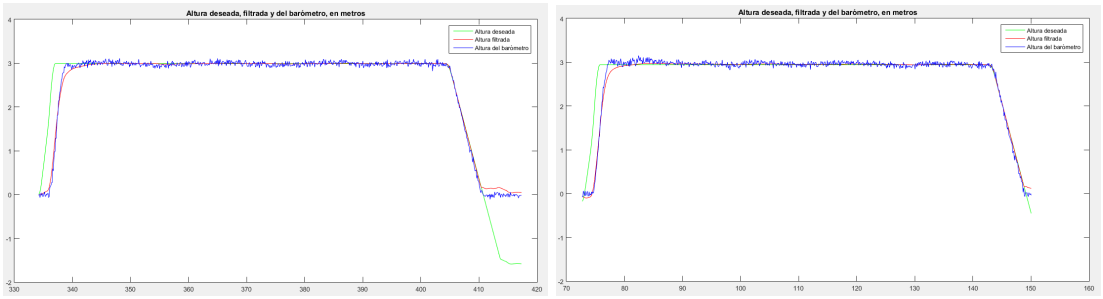


(a) Gráfica de aceleración con masa nominal (b) Gráfica de aceleración con masa modificada a 1.6 kg

Figura 5.19: Comparación de gráficas de aceleraciones en el eje z en los ensayos nominal y con masa modificada con pérdida de altura

- **Ensayo con la masa de los rotores modificada a 0.01 kg**

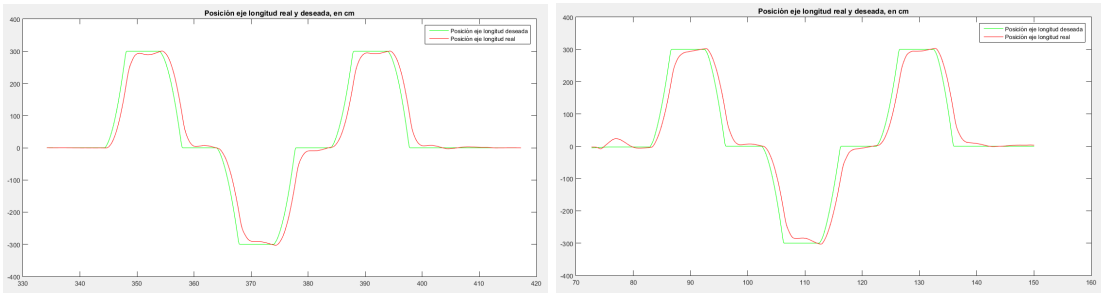
  En este ensayo, las gráficas obtenidas tras el vuelo son muy similares a las gráficas del ensayo con parámetros nominales, no hay cambios apreciables en ninguna de ellas. Por tanto, concluimos que la modificación en la masa de los rotores no significa ningún cambio apreciable en el vuelo del vehículo, al menos con el valor de 10 gramos por rotor que hemos introducido (el doble del valor por defecto). Es algo razonable, ya que solamente introducimos un total de 0.04 kg en el modelo físico, y además de forma equilibrada ya que todos los rotores aumentan su masa por igual (0.01 kg por rotor). En las figuras posteriores se muestran algunas gráficas de comparación para mostrar la similitud.



(a) Gráfica de altura con la masa nominal de rotores

(b) Gráfica de altura con la masa de rotores modificada

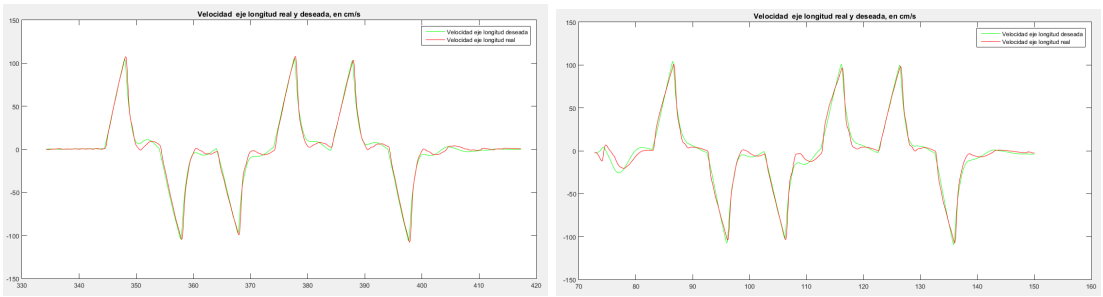
Figura 5.20: Comparación de gráficas de altura en los ensayos nominal y con rotores modificados



(a) Gráfica de posición en el eje de longitud con la masa nominal de rotores

(b) Gráfica de posición en eje de longitud con la masa de rotores modificada

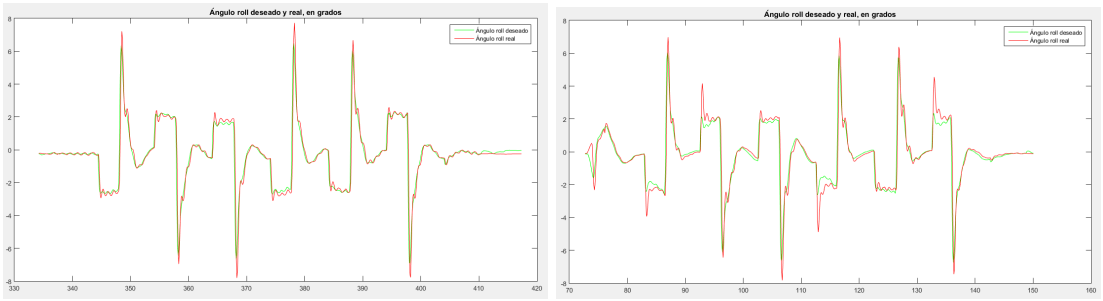
Figura 5.21: Comparación de gráficas de posiciones en los ensayos nominal y con rotores modificados



(a) Gráfica de velocidad en el eje de longitud con masa nominal de rotores

(b) Gráfica de velocidad en el eje de longitud con la masa de rotores modificada

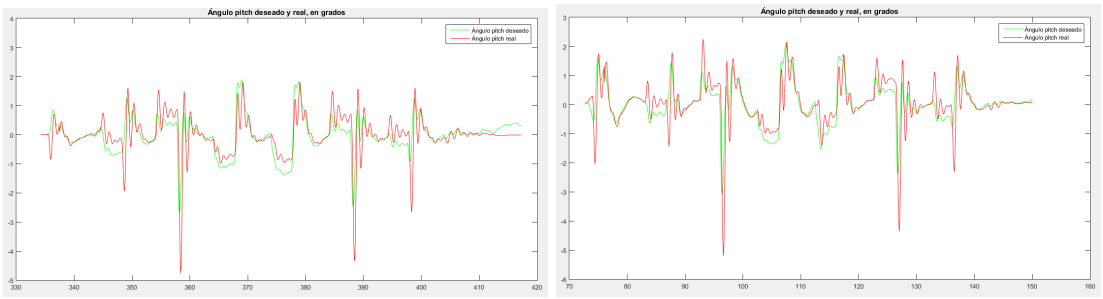
Figura 5.22: Comparación de gráficas de velocidades en los ensayos nominal y con rotores modificados



(a) Gráfica de ángulo *roll* con masa nominal de rotores

(b) Gráfica de ángulo *roll* con masa de rotores modificada

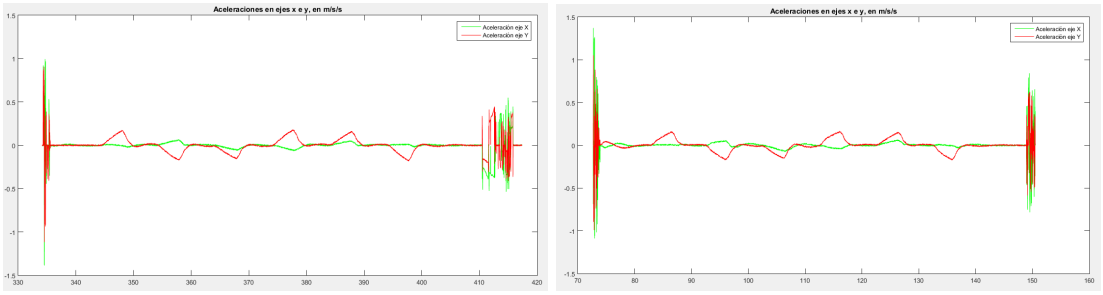
Figura 5.23: Comparación de gráficas de ángulo *roll* en los ensayos nominal y con rotores modificados



(a) Gráfica de ángulo *pitch* con masa nominal de rotores

(b) Gráfica de *pitch* con masa de rotores modificada

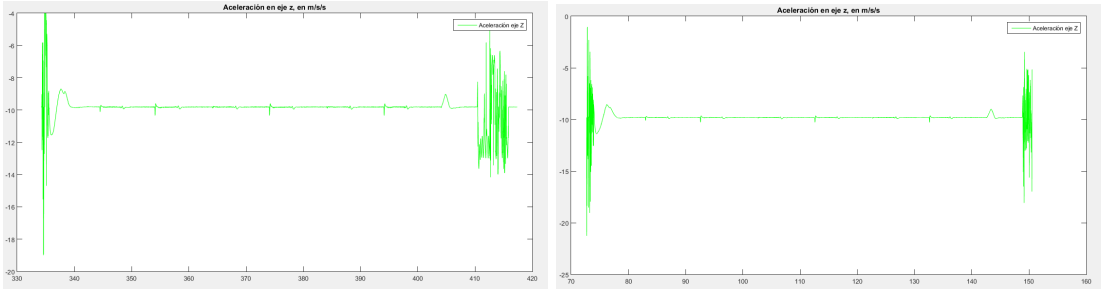
Figura 5.24: Comparación de gráficas de ángulo *pitch* en los ensayos nominal y con rotores modificados



(a) Gráfica de aceleraciones con masa nominal de rotores

(b) Gráfica de acel. con masa de rotores modificada

Figura 5.25: Comparación de gráficas de aceleraciones en los ensayos nominal y con rotores modificados



(a) Gráfica de aceleración con masa nominal de rotores

(b) Gráfica de aceleración con masa de rotores modificada

Figura 5.26: Comparación de gráficas de aceleraciones en los ensayos nominal y con rotores modificados

5.2.2 Parámetros del controlador (PID)

En el capítulo 3 conocimos los diferentes controladores presentes en *ArduCopter*, los cuales pueden ser modificados usando una estación de control, por ejemplo *APMPlanner 2*. En la imagen 5.27 podemos verlos de nuevo, con sus valores por defecto. Esta vez, se han recuadrado solamente los controladores de aceleración vertical y velocidad horizontal, ya que son los que modificaremos.

En esta sección, volveremos a simular la misma trayectoria que en los apartados anteriores. En este caso, introduciremos alguna modificación en los controladores mencionados anteriormente, para comprobar cuál es la influencia de un cambio en alguno de ellos sobre la estabilidad del vuelo del *UAV*.

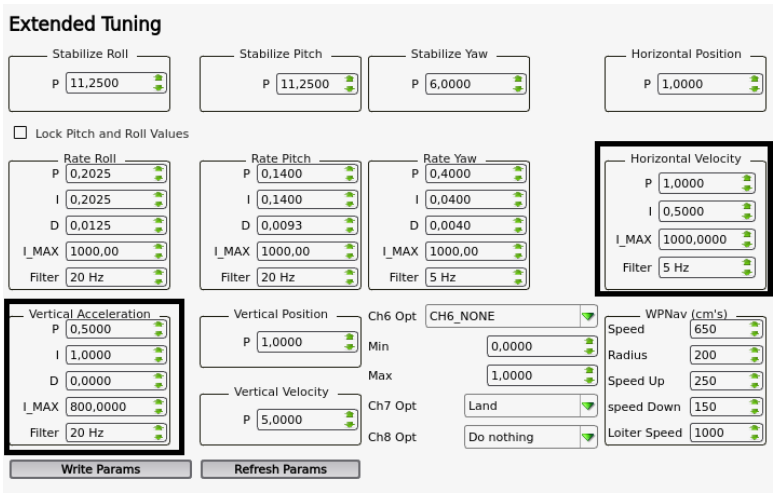


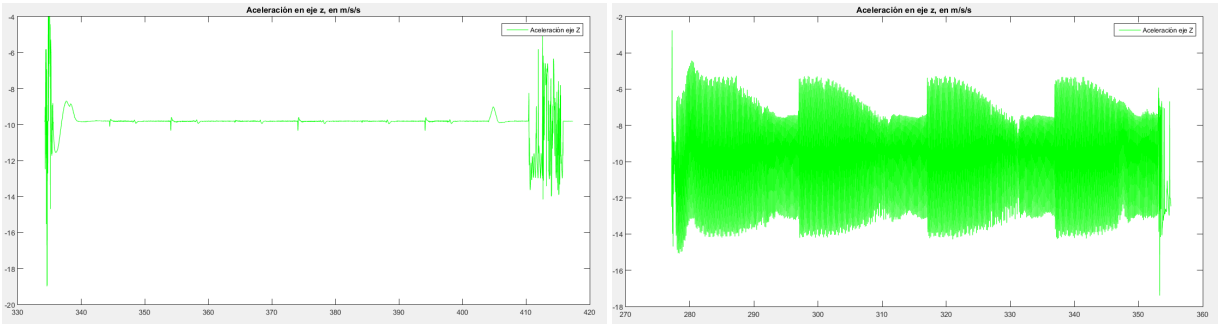
Figura 5.27: Controladores de *ArduCopter*

En primer lugar, comenzaremos modificando el controlador de aceleración vertical, que se encargaba de convertir el error de aceleración en la actuación de los motores. Introduciremos cambios en el PID de *Vertical Acceleration*, teniendo en cuenta que hay que mantener la relación 1:2 entre la P y la I, como ya se comentó en el capítulo 3. Sus valores por defecto, según vemos en la figura anterior, son P=0.5 e I=1. Para este nuevo ensayo, introduciremos un valor de P=3 e I=6, y comprobaremos mediante las gráficas la influencia de este cambio en la sesión de vuelo.

Posteriormente, modificaremos el controlador de velocidad horizontal. Vemos que los valores por defecto de *Horizontal Velocity* son P=1 e I=0.5. En este caso, realizaremos dos ensayos distintos: en primer lugar introduciremos un valor de P=2 e I=1, y posteriormente otro ensayo con P=5 e I=2.5. En ambos casos, analizaremos las gráficas obtenidas para encontrar las posibles diferencias con el ensayo original.

- **Ensayo modificando el controlador *Vertical Acceleration*: P=3, I=6**

  Analizando las gráficas de este ensayo en *MATLAB*, comprobamos que en general son muy similares a las gráficas del ensayo original, no hay grandes modificaciones; la altura, velocidades, posiciones y ángulos siguen a las consignas de forma correcta, y el dron realiza la trayectoria de forma satisfactoria. Sin embargo, analizando la gráfica de aceleración en el eje z, la cual vemos en la figura 5.28, observamos una gran diferencia. A pesar de que el vehículo pudo realizar la trayectoria sin problema, vemos que al aumentar la P y la I se genera una gran vibración en el eje vertical, lo cual es un efecto indeseable, ya que puede causar problemas en la señal de actuación y por tanto, en un caso real, también en la alimentación de los motores.

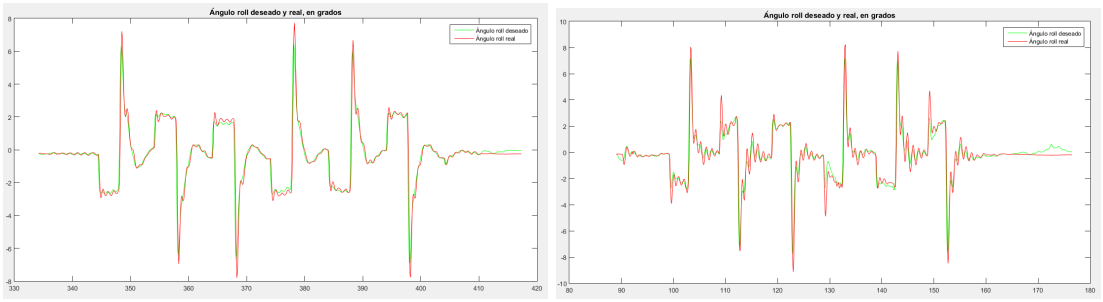


(a) Gráfica de aceleración con parámetros por defecto (b) Gráfica de aceleración con *Vertical Acc* modificado

Figura 5.28: Comparación de gráficas de aceleraciones en el eje z en los ensayos nominal y con controlador *Vertical Acc* modificado

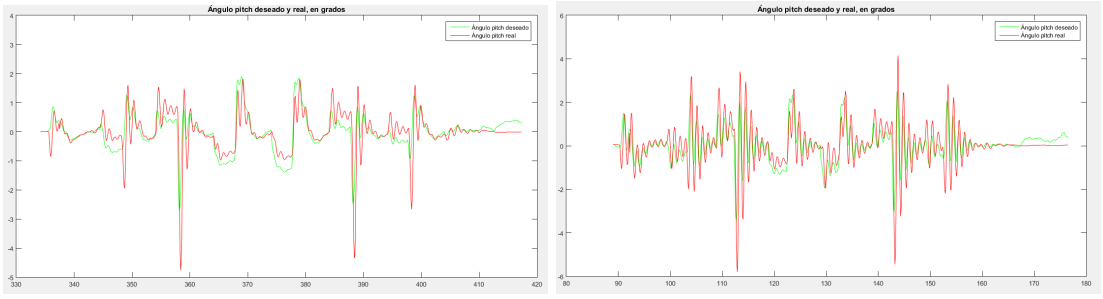
- **Ensayo modificando el controlador *Horizontal Velocity*: P=2, I=1**

  En la realización de esta simulación con los parámetros modificados, el dron es capaz de realizar la trayectoria correctamente, sin embargo se observa una respuesta menos amortiguada al finalizar los giros *roll y pitch*. Esto no afecta a la velocidad ni al posicionamiento del vehículo, que son bastante similares al ensayo nominal. En las gráficas de *roll y pitch*, comparadas con las del ensayo original, podemos visualizar esos rebotes en los giros, que hacen que el vehículo tome unos grados de inclinación mayor de los esperados, aunque sin llegar a la inestabilidad (figuras 5.29 y 5.30).



(a) Gráfica de ángulo *roll* con parámetros por defecto (b) Gráfica de ángulo *roll* con *Horizontal Vel* modificado

Figura 5.29: Comparación de gráficas de ángulo *roll* en los ensayos nominal y con controlador *Horizontal Velocity* modificado

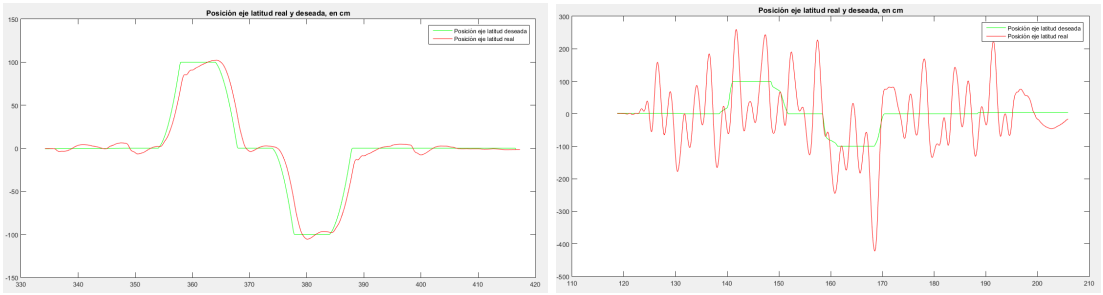


(a) Gráfica de ángulo *pitch* con parámetros por defecto (b) Gráfica de ángulo *pitch* con *Horizontal Vel* modificado

Figura 5.30: Comparación de gráficas de ángulo *pitch* en los ensayos nominal y con controlador *Horizontal Velocity* modificado

- **Ensayo modificando el controlador *Horizontal Velocity*: P=5, I=2.5**

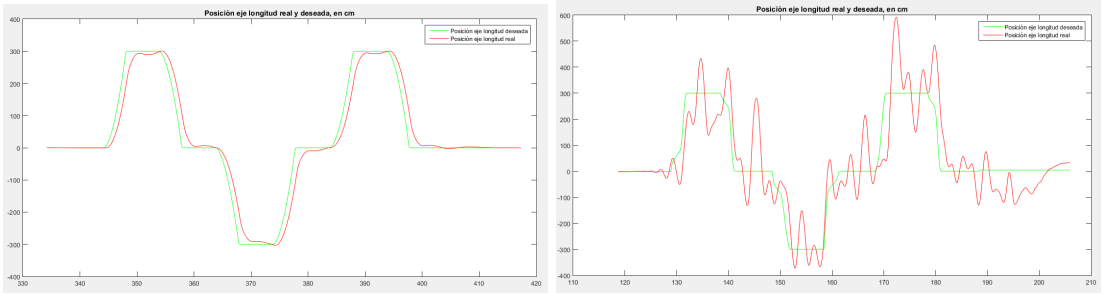
  Si aumentamos a un valor mayor la P y la I, como es el caso de este ensayo, el dron se vuelve completamente inestable, incapaz de mantener la posición en ningún punto de la trayectoria. La posición horizontal y velocidades no son capaces de seguir a la consigna deseada. Como consecuencia, el controlador angular deja de funcionar correctamente, enviando consignas de ángulos erróneas, llegando casi a los 40º de inclinación en algunos instantes, como podemos ver en las gráficas de comparación expuestas.



(a) Gráfica de posición en el eje de latitud con parámetros por defecto

(b) Gráfica de posición en eje de latitud con sistema inestable

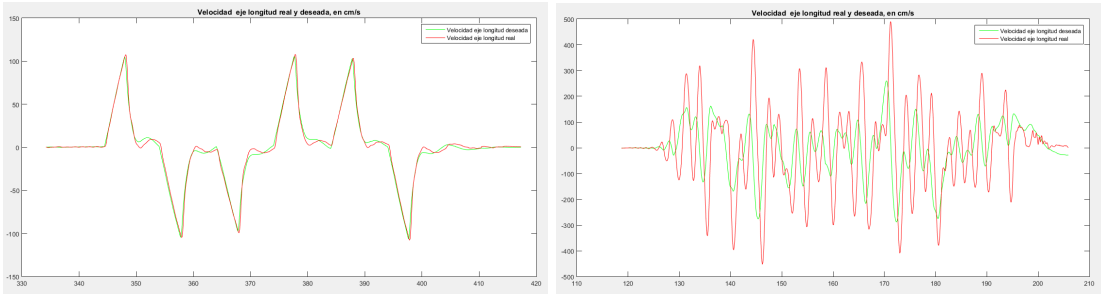
Figura 5.31: Comparación de gráficas de posiciones en los ensayos con P=1, I=0.5 (valores nominales con comportamiento estable), y con P=5, I=2.5 (valores modificados con comportamiento inestable)



(a) Gráfica de posición en el eje de longitud con parámetros por defecto

(b) Gráfica de posición en eje de longitud con sistema inestable

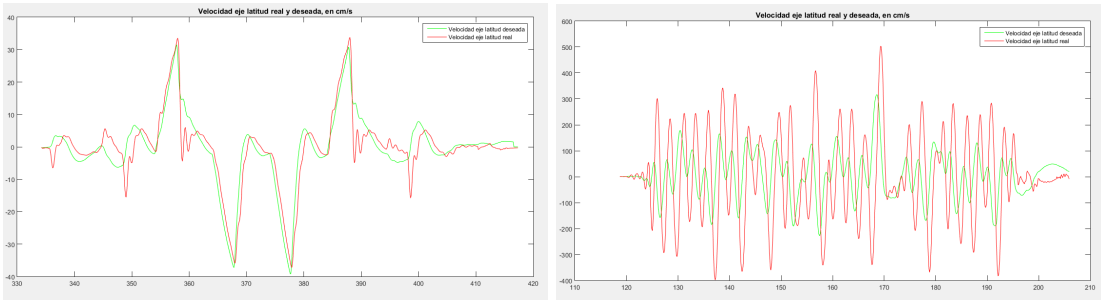
Figura 5.32: Comparación de gráficas de posiciones en los ensayos con P=1, I=0.5 (valores nominales con comportamiento estable), y con P=5, I=2.5 (valores modificados con comportamiento inestable)



(a) Gráfica de velocidad en el eje de longitud con con parámetros por defecto

(b) Gráfica de velocidad en el eje de longitud con con sistema inestable

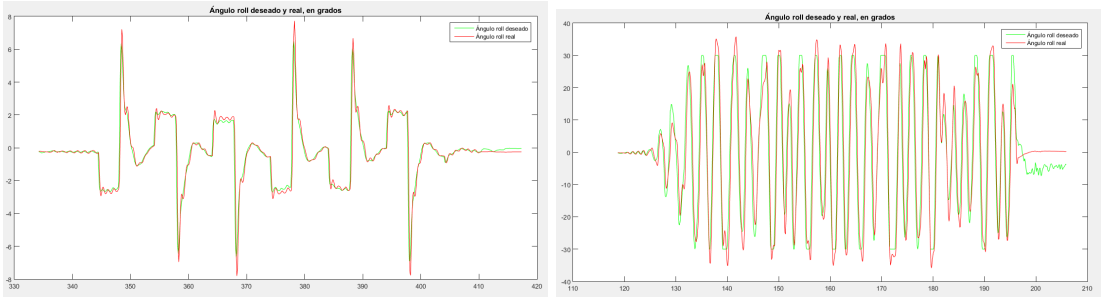
Figura 5.33: Comparación de gráficas de velocidades en los ensayos con P=1, I=0.5 (valores nominales con comportamiento estable), y con P=5, I=2.5 (valores modificados con comportamiento inestable)



(a) Gráfica de velocidad en el eje de latitud con con parámetros por defecto

(b) Gráfica de velocidad en el eje de latitud con con sistema inestable

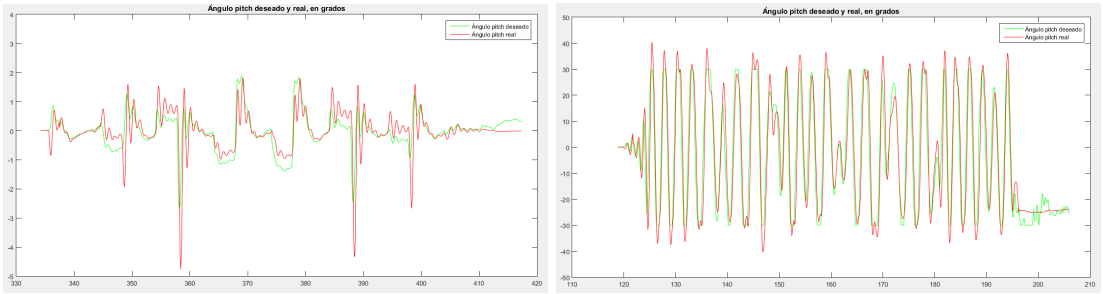
Figura 5.34: Comparación de gráficas de velocidades en los ensayos con P=1, I=0.5 (valores nominales con comportamiento estable), y con P=5, I=2.5 (valores modificados con comportamiento inestable)



(a) Gráfica de *roll* con parámetros por defecto

(b) Gráfica de ángulo *roll* con sistema inestable

Figura 5.35: Comparación de gráficas de ángulo *roll* en los ensayos con P=1, I=0.5 (valores nominales con comportamiento estable), y con P=5, I=2.5 (valores modificados con comportamiento inestable)



(a) Gráfica de *pitch* con parámetros por defecto

(b) Gráfica de ángulo *pitch* con sistema inestable

Figura 5.36: Comparación de gráficas de ángulo *pitch* en los ensayos con P=1, I=0.5 (valores nominales con comportamiento estable), y con P=5, I=2.5 (valores modificados con comportamiento inestable)

Como podemos ver, es incapaz de seguir las consignas de posición y velocidad en ningún momento durante la trayectoria. En el caso de los ángulos, vemos cómo oscila entre valores bastante elevados continuamente, siendo incapaz de estabilizarse.

Capítulo 6

Conclusiones

En este trabajo hemos logrado disponer de un dron *Erle-Copter* virtual dentro de un entorno de simulación en *ROS/Gazebo*, con el cual hemos podido evaluar su comportamiento y programar trayectorias en un entorno definido por el usuario.

En cuanto al *Erle-Copter* simulado, conocimos su modelado, tanto parámetros físicos como del autopiloto que utiliza. Vimos los controladores principales que implementa, y el uso de ellos en los distintos modos de vuelo que pone a nuestra disposición.

Para lograr la simulación dentro del entorno *Gazebo*, se presentaron las herramientas necesarias a instalar y los comandos a ejecutar para conseguir arrancar una simulación, además de ciertos ajustes previos antes de poder comenzar con las simulaciones. Adicionalmente, hemos conocido la herramienta *ROS*, que nos ha permitido simular una trayectoria con varios puntos objetivos a los que el dron tenía que desplazarse.

En el análisis del rendimiento del dron durante el vuelo, nos hemos ayudado de los datos post-vuelo que genera el autopiloto tras cada sesión, así como su representación en gráficas de *MATLAB*. En la primera fase de simulación de la trayectoria con todos los parámetros por defecto, vimos que el dron se comportaba de manera fiable y según lo esperado, siguiendo a las consignas correctamente, todo ello gracias al buen funcionamiento de los sensores simulados y de la fusión sensorial implementada con su versión de *EKF*.

Tras realizar la simulación de la fase anterior, modificamos algunos parámetros para comprobar su influencia en el vuelo. En el ensayo en el cual incrementamos la masa nominal, vimos que el dron tenía una apariencia menos estable durante el vuelo, debido a los cambios bruscos en cortos instantes de tiempo que sufría tanto la velocidad como los ángulos *roll y pitch*.

Finalmente, evaluamos el efecto de modificar parámetros de los controladores PID, de aceleración vertical y de velocidad horizontal. En general, el comportamiento del dron es bastante sensible a la variación de estos parámetros. Además, el triple lazo de control implementado dificulta el ajuste individualizado de los diferentes parámetros de control.

Bibliografía

[1] "Documentación de Erle Copter," http://docs.erlerobotics.com/erle_robots/erle_copter [Último acceso junio/2018].

[2] "Web oficial de Erle Robotics," http://erlerobotics.com/blog/ [Último acceso junio/2018].

[3] "Documentación de Erle Brain 3," http://docs.erlerobotics.com/brains/erle-brain-3 [Último acceso junio/2018].

[4] "Web oficial de ArduPilot," http://ardupilot.org/ [Último acceso junio/2018].

[5] "Información sobre ArduCopter," http://ardupilot.org/copter/index.html [Último acceso junio/2018].

[6] "Web oficial de ROS," http://www.ros.org/ [Último acceso junio/2018].

[7] "Documentación y tutoriales de ROS," http://wiki.ros.org/es [Último acceso junio/2018].

[8] "Web oficial de Gazebo," http://gazebosim.org/ [Último acceso junio/2018].

[9] "Documentación sobre MAVLink," https://erlerobotics.gitbooks.io/erlerobot/en/mavlink/mavlink.html [Último acceso junio/2018].

[10] "Información del paquete MAVROS," http://wiki.ros.org/mavros [Último acceso junio/2018].

[11] "Web oficial de APMPlanner2," http://ardupilot.org/planner2/ [Último acceso junio/2018].

[12] "Documentación de MAVProxy," https://ardupilot.github.io/MAVProxy/html/index.html [Último acceso junio/2018].

[13] "Información sobre el EKF," http://ardupilot.org/dev/docs/ekf.html [Último acceso junio/2018].

[14] "Información del Attitude Control de ArduCopter," http://ardupilot.org/dev/docs/apmcopter-programming-attitude-control-2.html [Último acceso junio/2018].

[15] "Información sobre la IMU y sus sensores integrados," http://students.iitk.ac.in/roboclub/lectures/IMU.pdf [Último acceso junio/2018].

[16] "Documentación sobre los modos de vuelo de ArduCopter," http://ardupilot.org/copter/docs/flight-modes.html [Último acceso junio/2018].

[17] "Información sobre propiedades físicas de la escena en Gazebo," http://gazebosim.org/tutorials?tut=modifying_world [Último acceso junio/2018].

[18] "Información sobre ejes de coordenadas NED," https://en.wikipedia.org/wiki/North_east_down [Último acceso junio/2018].

[19] "Documentación sobre archivos log," https://erlerobotics.gitbooks.io/erle-robotics-mav-tools-free/content/en/understanding_a_log_file/index.html [Último acceso junio/2018].

[20] "Información sobre herramientas del paquete MAVLink," http://ardupilot.github.io/MAVProxy/html/analysis_and_simulation/mavtools.html [Último acceso junio/2018].

[21] "Documento sobre los valores aceptables de las vibraciones del acelerómetro," https://erlerobotics.gitbooks.io/erle-robotics-mav-tools-free/content/en/understanding_a_log_file/imu.html [Último acceso junio/2018].

[22] A. C. Frasquet, "Development of a quaternion-based quadrotor control system based on disturbance rejection," Ph.D. dissertation, Universidad Politécnica de Valencia, 2016.

[23] H. M. Redouane Dargham, Adil Sayouti, "Euler and quaternion parameterization in vtol uav dynamics with test model efficiency," *International Journal of Applied Information Systems (IJAIS)*, vol. 9, no. 8, Octubre 2015.

[24] "Documentación sobre la configuración del entorno de simulación," http://docs.erlerobotics.com/simulation/configuring_your_environment [Último acceso junio/2018].

Apéndice A

Modelo de quadrotor basado en cuaterniones

La aparición de los cuaterniones surge con la idea de una generalización de los números complejos como herramienta para manejar rotaciones en 3D. Su expresión general es la siguiente:

$$q = q_0 + q_1 i + q_2 j + q_3 k$$

Puede ser tratado como un número hipercomplejo, donde la parte "real" es un número complejo $q_0 + q_1 i$ y la parte "imaginaria" es otro número complejo $q_2 + q_3 i$:

$$q = q_0 + q_1 i + (q_2 + q_3 i) j$$

Tras esta breve presentación de un cuaternio en su forma general, vamos a tratar de explicar el modelo de un quadrotor basado en cuaterniones y sus ventajas frente al modelado basado en ángulos de Euler. Se puede encontrar más información sobre cuaterniones, su definición y el modelo en las referencias [22] y [23].

En primer lugar, es necesario conocer los tres sistemas de coordenadas definidos para cualquier vehículo aéreo, que aparecerán durante la explicación del modelo.

- Sistema de coordenadas de cuerpo (*body frame*, lo llamaremos B). Tiene su origen en el centro de masas del vehículo, se mueve y rota junto a él.

- Sistema de coordenadas inercial (*inertial frame*, lo llamaremos I). Es un sistema de coordenadas fijo, definido en cualquier punto del entorno. Normalmente, usa coordenadas *NED*, donde el eje x apunta hacia al Norte, el eje y apunta al Este y el eje z apunta hacia abajo.

- Sistema de coordenadas de referencia (*reference frame*, lo llamaremos R). Estos ejes irán variando su posición y orientación según cambie la consigna por parte del piloto. El control del vehículo tratará de rotar y trasladar B para conseguir que siga a este sistema de referencia R.
  Es decir, el objetivo del algoritmo de control del vehículo es que el sistema de coordenadas de cuerpo B siga al sistema de referencia R, que es cambiante según la consigna que se envíe. Podemos ver un ejemplo gráfico de esto en la figura A.1.

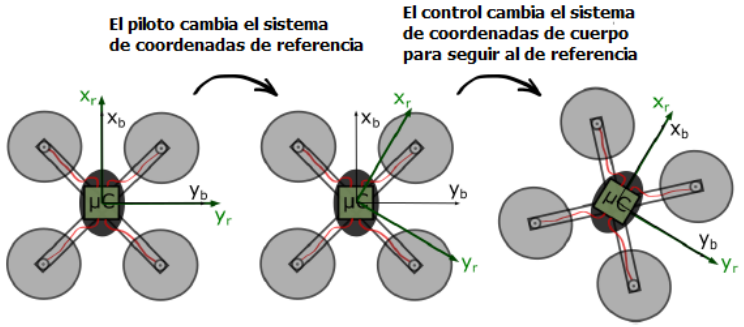


Figura A.1: Ejemplo de control que fuerza al sistema de coordenadas de cuerpo a seguir al sistema de referencia

El modelo del quadrotor puede ser tratado como una conexión en cascada de dos subsistemas: el modelo cinemático y el modelo dinámico. El dinámico depende de parámetros físicos y describe el cambio de la velocidad angular del vehículo ante las entradas del control y perturbaciones. El cinemático no depende de parámetros físicos y estudia la variación de la orientación del vehículo (*attitude variance*) con la velocidad angular.

A.1 Modelo cinemático

Comenzaremos con el modelo cinemático, donde las variables más relevantes se muestran en la figura A.2.

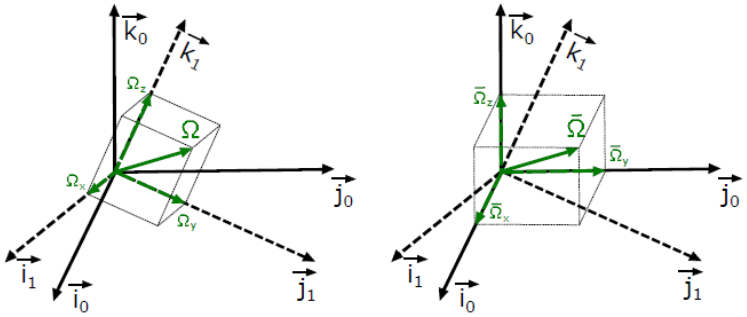


Figura A.2: El sistema B rota con respecto a I con una velocidad angular Ω

La orientación relativa del sistema de referencia de cuerpo B (i_1, j_1, k_1) con respecto al inercial I (i_0, j_0, k_0), se define como la rotación de un ángulo θ con respecto a un eje $u \in \mathbb{R}^3$, y puede ser codificada en un cuaternión de la siguiente forma:

$$q = \cos(\theta/2) + u\sin(\theta/2) \tag{A.1}$$

Si B está rotando con una velocidad angular $\bar{\Omega}$, expresada en I, la ecuación sería la siguiente:

$$\dot{q} = \frac{1}{2}\bar{\Omega}q \tag{A.2}$$

Normalmente, la velocidad angular la encontramos expresada en B (y no en I). En ese caso, podemos hacer $\bar{\Omega}$=qΩq*, y sustituyendo en la ecuación anterior quedaría:

$$\dot{q} = \frac{1}{2} q\Omega \tag{A.3}$$

Expresando la ecuación anterior con respecto a los sistemas de referencia B e I, obtenemos la siguiente ecuación, en la que $\omega_b = (0, \Omega_b)^T \in \mathbb{R}^4$ es el cuaternión asociado a la velocidad angular del cuerpo:

$$\dot{q}_I^B = \frac{1}{2} q_I^B \otimes \omega_b \tag{A.4}$$

La ecuación A.4 representa el modelo cinemático del quadrotor con respecto al sistema inercial, I. Para obtener un modelo cinemático más general, se asume que la orientación deseada del vehículo vendrá expresada por una trayectoria $q_I^R(t)$ en el sistema de coordenadas de referencia, R, y también se debe cumplir:

$$\dot{q}_I^R = \frac{1}{2} q_I^R \otimes \omega_d \tag{A.5}$$

donde $\omega_d = (0, \Omega_d)^T \in \mathbb{R}^4$ es el cuaternión asociado a la velocidad angular del sistema de referencia deseado.

El error de orientación relativo entre los sistemas B y R se puede obtener de una composición de rotaciones:

$$q_R^B = (q_I^R)^* \otimes q_I^B \tag{A.6}$$

El cuaternio q_R^B obtenido es una medida de la orientación del quadrotor, B, con respecto al marco de referencia deseado, R. Teniendo esto, y siguiendo el modelo de la ecuación anterior A.4, la expresión definitiva quedaría:

$$\dot{q}_R^B = \frac{1}{2} q_R^B \otimes \tilde{\omega} \tag{A.7}$$

con $\tilde{\omega} = (0, \tilde{\Omega}) = \omega_b - (q_R^B)^* \otimes \omega_d \otimes q_R^B$.

La ecuación A.7 representa el modelo cinemático completo del quadrotor.

A.2 Modelo dinámico

La dinámica rotacional del quadrotor viene dada por la segunda ley de Newton, de la siguiente forma:

$$\dot{\Omega}_b = I_{cm}^{-1} \cdot \tau(t) \tag{A.8}$$

donde Ω_b es la velocidad angular, τ son las fuerzas totales que actúan sobre el vehículo e I_{cm} es la matriz de inercia, que se considera una diagonal. Para un quadrotor, $\tau(t)$ puede expresarse como la suma de las fuerzas producidas por las hélices, el término de Coriolis y los efectos aerodinámicos y giroscópicos:

$$\tau(t) = M \cdot u - \Omega_b \times I_{cm}\Omega_b - G_a(t) - \tau_{aero}(t) + p(t) \tag{A.9}$$

La matriz $M = diag(m_{11}, m_{22}, m_{33})$ relaciona la fuerza con la velocidad de los motores, u $\in \mathbb{R}^3$ es el eje de rotación visto anteriormente, G_a y τ_{aero} son las fuerzas giroscópicas y aerodinámicas, respectivamente. El término p(t) representa otras perturbaciones externas desconocidas y dinámicas no modeladas.

Aunque con estos términos se podría proveer una descripción completa de la dinámica, es necesario remarcar que ninguno de ellos es conocido con exactitud, ya que dependen de parámetros o coeficientes muy difíciles de modelar o incluso desconocidos en algunos casos.

Por este motivo, vamos a reescribir el modelo dinámico del quadrotor de la siguiente forma:

$$\dot{\Omega}_b = d(t) + f(\Omega, t) + Bu(t) \tag{A.10}$$

donde $d(t) = I_{cm}^{-1} p(t)$ y $B = I_{cm}^{-1} M = diag(b_1, b_2, b_3)$. El término $f(\Omega, t)$ contiene el resto de dinámicas desconocidas: $f(\Omega, t) = -I_{cm}^{-1}(\Omega_b \times I_{cm}\Omega_b + G_a(t) + \tau_{aero}(t))$.

El control dinámico debería llevar al quadrotor a seguir una trayectoria de referencia dada, $\Omega_r(t)$. Por tanto, para el diseño del control es útil definir el error como $x_1 = \Omega_r - \Omega_b$. Diferenciando x_1, se obtiene:

$$\dot{x}_1 = \tilde{d}(t) + f(\Omega, t) + Bu(t) \tag{A.11}$$

donde $\dot{\Omega}_r$ es considerada una perturbación desconocida y de este modo $\tilde{d}(t) = d(t) - \dot{\Omega}_r$.

El modelo presente en la ecuación A.11, que contiene información sobre las dinámicas conocidas, será utilizado para el diseño del control.

A.3 Diferencias principales con el modelo basado en ángulos de Euler

El modelo de un quadrotor puede ser desarrollado usando cuaterniones, como hemos visto en este apéndice, o bien utilizando ángulos de Euler.

Este último tiene dos desventajas fundamentales: es fuertemente no lineal y tiene una singularidad en $\pi/2$. Por este motivo, el modelo basado en cuaterniones ha cobrado una gran importancia, ya que elimina la singularidad y reduce notablemente las no linealidades. Los cuaterniones suponen una gran herramienta para manejar rotaciones en 3D y por ello tienen una aplicación directa en el control de los quadrotor.

Desde el punto de vista matemático, el modelo basado en cuaterniones es numéricamente más eficiente y más estable en comparación con la formulación tradicional de rotación, lo que permite una implementación del algoritmo de control más rápida y eficiente.

Apéndice B

Configuración del entorno de simulación

Para poder lanzar la simulación del *Erle-Copter* en el ordenador, es necesario primero configurar el entorno de trabajo [24], instalando algunas herramientas necesarias y sus dependencias para conseguir un correcto funcionamiento de la simulación.

En este apéndice se verá paso a paso la instalación de los paquetes necesarios para tener un espacio de trabajo funcional en el que se pueda lanzar la simulación del *Erle-Copter* en *Gazebo*, además de algunas recomendaciones en la instalación de estas herramientas basadas en la propia experiencia al realizar este trabajo.

B.1 Versión de Ubuntu recomendada

La versión de *Ubuntu Linux* recomendada para poder realizar la simulación es **Ubuntu 14.04 64 bits**, ya que todas las herramientas y paquetes a instalar son compatibles con ella.

B.2 Instalación de paquetes base y MAVProxy

En primer lugar, vamos a instalar los paquetes base (*gawk, make, git, curl, cmake*) para la configuración de la simulación. A continuación se exponen los comandos necesarios para su instalación:

```
sudo apt-get update
sudo apt-get install gawk make git curl cmake -y
```

Una vez hecho esto, vamos a instalar dependencias necesarias para *MAVProxy*:

```
sudo apt-get install g++ python-pip python-matplotlib python-serial python-wxgtk2.8 python-scipy
python-opencv python-numpy python-pyparsing ccache realpath libopencv-dev -y
```

NOTA: si el paquete *python-wxgtk2.8* da problemas en la instalación y no encuentra el repositorio, se puede probar a instalar *python-wxgtk3.0* que es la versión siguiente.

```
sudo apt-get install python-wxgtk3.0
```

Ahora ya podemos instalar MAVProxy:

```
sudo pip install future
sudo apt-get install libxml2-dev libxslt1-dev -y
sudo pip2 install pymavlink catkin_pkg --upgrade
sudo pip install MAVProxy==1.5.2
```

Es importante instalar la versión mencionada en el último comando, la 1.5.2, ya que es la que tiene compatibilidad con el resto de herramientas para la simulación.

Para finalizar esta sección, vamos a descargar e instalar *ArUco*, que se trata de una librería de realidad aumentada desarrollada por el grupo A.V.A de la Universidad de Córdoba.

```
cd ~/Descargas
tar -xvzf aruco-1.3.0.tgz
cd aruco-1.3.0/
mkdir build && cd build
cmake ..
make
sudo make install
```

B.3 Instalación de ArduPilot

A continuación, vamos a proceder a la instalación de una rama especifica de *ArduPilot*. En esta rama que vamos a compilar está contenido *ArduCopter*, que es el autopiloto que nos interesa para este trabajo como ya se explicó en 2.2.1.
Crearemos una carpeta de nombre *simulation* en la cual lo instalaremos:

```
mkdir -p ~/simulation
cd ~/simulation
git clone https://github.com/erlerobot/ardupilot -b gazebo
```

B.4 Instalación de ROS Indigo

Para comenzar la instalación, antes debemos preparar el ordenador para poder aceptar software de *packages.ros.org* y configurar las claves, mediante los siguientes comandos:

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

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key 0xB01FA116
```

Actualizamos el índice de paquetes *Debian*:

```
sudo apt-get update
```

Una vez tenemos esto, procedemos a la instalación *Desktop-Full* de *ROS*. Es la recomendada ya que contiene todas las herramientas y paquetes de *ROS*.

```
sudo apt-get install ros-indigo-desktop-full
```

Por defecto, con *ROS Indigo* viene pre-instalado *Gazebo 2*. Como ya se comentó en 2.2.3, nosotros usaremos *Gazebo 7* para esta simulación, así que lo instalaremos posteriormente.

Antes de poder usar *ROS*, debemos inicializar *rosdep*, herramienta que permite instalar dependencias del sistema y es requerido para ejecutar algunos componentes del sistema de *ROS*. Lo hacemos de la siguiente forma:

```
sudo rosdep init
rosdep update
```

Para finalizar, instalaremos algunas dependencias adicionales de *ROS*:

```
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 \
                          ros-indigo-transmission-interface \
                          ros-indigo-joint-limits-interface \
                          unzip -y
```

B.4.1 Creación del espacio de trabajo de ROS (*ROS workspace*)

Un *ROS workspace* es un directorio donde se pueden modificar, compilar e instalar paquetes *catkin*, que es el sistema oficial de compilación de *ROS*.

En primer lugar, creamos un directorio llamado *ros_catkin_ws*, dentro de la carpeta *simulation* que hicimos anteriormente, donde crearemos e inicializaremos el espacio de trabajo:

```
mkdir -p ~/simulation/ros_catkin_ws/src
cd ~/simulation/ros_catkin_ws/src
catkin_init_workspace
```

```
cd ~/simulation/ros_catkin_ws
catkin_make
source devel/setup.bash
```

Una vez hecho esto, vamos a descargar en este espacio de trabajo paquetes necesarios para la simulación, tal y como podemos ver en los comandos a continuación:

```
cd src/

git clone https://github.com/erlerobot/ardupilot_sitl_gazebo_plugin
git clone https://github.com/tu-darmstadt-ros-pkg/hector_gazebo/
git clone https://github.com/erlerobot/rotors_simulator -b sonar_plugin
git clone https://github.com/PX4/mav_comm.git
git clone https://github.com/ethz-asl/glog_catkin.git
git clone https://github.com/catkin/catkin_simple.git
git clone https://github.com/erlerobot/mavros.git
git clone https://github.com/ros-simulation/gazebo_ros_pkgs.git -b indigo-devel
```

B.5 Instalación de Gazebo

Vamos a instalar *Gazebo 7* usando paquetes de *Ubuntu*. En primer lugar, tenemos que preparar el ordenador para poder aceptar software de *packages.osrfundation.org* y configurar las claves, mediante los siguientes comandos:

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

```
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
```

Para continuar, borraremos las versiones ya instaladas de *Gazebo* y sus dependencias (en este caso *Gazebo 2*, que se instaló por defecto al instalar *ROS*) e instalamos *Gazebo 7*:

```
sudo apt-get update
```

```
sudo apt-get remove .*gazebo.* '.*sdformat.*' '.*ignition-math.*'
&& sudo apt-get update && sudo apt-get install gazebo7 libgazebo7-dev drcsim7 -y
```

Una vez lo tenemos instalado, vamos a compilar todo junto en el *workspace* creado anteriormente en B.4.1, en la carpeta *ros_catkin_ws*:

```
cd ~/simulation/ros_catkin_ws
catkin_make --pkg mav_msgs mavros_msgs gazebo_msgs
source devel/setup.bash
catkin_make -j 4
```

Para concluir, descargamos algunos modelos de *Gazebo* necesarios para la simulación. Lo hacemos de la siguiente forma:

```
mkdir -p ~/.gazebo/models
git clone https://github.com/erlerobot/erle_gazebo_models
mv erle_gazebo_models/* ~/.gazebo/models
```

B.6 Instalación de APM Planner 2

En primer lugar, debemos descargar el paquete software de *APM Planner 2* para *Ubuntu* (archivo *.deb*), el cual podemos encontrar en la página oficial de *firmwares* de *ArduPilot*. Hay una gran cantidad de versiones disponibles para instalar; la escogida para este trabajo es la versión 2.0.18. Para instalarlo, ejecutamos el siguiente comando:

```
sudo dpkg -i apm_planner*.deb
```

Si la instalación da algún fallo antes de terminar, puede ser debido a la ausencia de alguna dependencia del programa. Estas dependencias se pueden instalar de la siguiente forma:

```
sudo apt-get -f install
```

Nuevamente, volvemos a lanzar la instalación con el comando anterior:

```
sudo dpkg -i apm_planner*.deb
```

Para ejecutar el programa, basta con poner en una terminal:

```
apmplanner2
```

Apéndice C

C.1 ANEXO: Código main.cpp de ejemplo de trayectoria usando ROS

```
#include <cstdlib>
#include <ros/ros.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/CommandTOL.h>
#include <mavros_msgs/CommandInt.h>
#include <mavros_msgs/SetMode.h>
#include <geometry_msgs/PoseStamped.h>

int main(int argc, char **argv)
{
int rate = 10;
int i=0;

ros::init(argc, argv, "mavros_takeoff");

ros::NodeHandle n;
ros::Rate r(rate);

////////////////////////////////////////////
////////////////GUIDED//////////////////////
////////////////////////////////////////////

ros::ServiceClient cl = n.serviceClient<mavros_msgs::SetMode>("/mavros/set_mode");

mavros_msgs::SetMode srv_setMode;

srv_setMode.request.base_mode = 0;
srv_setMode.request.custom_mode = "GUIDED";

if(cl.call(srv_setMode))
{
ROS_INFO("modo de vuelo enviado ok; %d", srv_setMode.response.success);
}
else
{
  ROS_ERROR("Failed SetMode");
  return -1;
}

////////////////////////////////////////////
```

```
//////////////////ARM//////////////////////
/////////////////////////////////////////////

ros::ServiceClient arming_cl = n.serviceClient<mavros_msgs::CommandBool>("/mavros/cmd/
    arming");

mavros_msgs::CommandBool srv;

srv.request.value = true;

if(arming_cl.call(srv))
{
ROS_INFO("motores armados ok; %d", srv.response.success);
}
else
{
  ROS_ERROR("Failed arming or disarming");
}

/////////////////////////////////////////////
/////////////////TAKEOFF/////////////////////
/////////////////////////////////////////////

ros::ServiceClient takeoff_cl = n.serviceClient<mavros_msgs::CommandTOL>("/mavros/cmd/
    takeoff");

mavros_msgs::CommandTOL srv_takeoff;

srv_takeoff.request.altitude = 3;
srv_takeoff.request.latitude = 0;
srv_takeoff.request.longitude = 0;
srv_takeoff.request.min_pitch = 0;
srv_takeoff.request.yaw = 0;

if(takeoff_cl.call(srv_takeoff))
{
ROS_INFO("despegue enviado ok; %d", srv_takeoff.response.success);
}
else
{
  ROS_ERROR("Failed Takeoff");
}

sleep(10);

/////////////////////////////////////////////
///////////primer vertice rombo//////////////
/////////////////////////////////////////////

ros::Publisher local_pos_pub = n.advertise<geometry_msgs::PoseStamped>("mavros/
    setpoint_position/local",10);

geometry_msgs::PoseStamped pose;

pose.pose.position.x = 3;
pose.pose.position.y = 0;
pose.pose.position.z = 3;

ROS_INFO("Primer vertice");
```

```
for(int i = 100; ros::ok() && i>0; --i)
{
  local_pos_pub.publish(pose);
  ros::spinOnce();
  r.sleep();
}


//////////////////////////////////////////
///////////segundo vertice rombo//////////
//////////////////////////////////////////

pose.pose.position.x = 0;
pose.pose.position.y = 1;
pose.pose.position.z = 3;

ROS_INFO("Segundo vertice");

for(int i = 100; ros::ok() && i>0; --i)
{
  local_pos_pub.publish(pose);
  ros::spinOnce();
  r.sleep();
}

//////////////////////////////////////////
///////////tercer vertice rombo///////////
//////////////////////////////////////////

pose.pose.position.x = -3;
pose.pose.position.y = 0;
pose.pose.position.z = 3;

ROS_INFO("Tercer vertice");

for(int i = 100; ros::ok() && i>0; --i)
{
  local_pos_pub.publish(pose);
  ros::spinOnce();
  r.sleep();
}

//////////////////////////////////////////
///////////cuarto vertice rombo///////////
//////////////////////////////////////////

pose.pose.position.x = 0;
pose.pose.position.y = -1;
pose.pose.position.z = 3;

ROS_INFO("Cuarto vertice");

for(int i = 100; ros::ok() && i>0; --i)
{
  local_pos_pub.publish(pose);
  ros::spinOnce();
  r.sleep();
}
```

```
//////////////////////////////////////////
/////////primer vertice rombo//////////////
//////////////////////////////////////////

pose.pose.position.x = 3;
pose.pose.position.y = 0;
pose.pose.position.z = 3;

ROS_INFO("Primer vertice");

for(int i = 100; ros::ok() && i>0; --i)
{
  local_pos_pub.publish(pose);
  ros::spinOnce();
  r.sleep();
}


//////////////////////////////////////////
////////vuelta al punto inicial///////////
//////////////////////////////////////////

pose.pose.position.x = 0;
pose.pose.position.y = 0;
pose.pose.position.z = 3;

ROS_INFO("Se prepara para aterrizar");

for(int i = 100; ros::ok() && i>0; --i)
{
  local_pos_pub.publish(pose);
  ros::spinOnce();
  r.sleep();
}


//////////////////////////////////////////
//////////////////LAND////////////////////
//////////////////////////////////////////

ros::ServiceClient land_cl = n.serviceClient<mavros_msgs::CommandTOL>("/mavros/cmd/land");

mavros_msgs::CommandTOL srv_land;

srv_land.request.altitude = 3;
srv_land.request.latitude = 0;
srv_land.request.longitude = 0;
srv_land.request.min_pitch = 0;
srv_land.request.yaw = 0;

if(land_cl.call(srv_land))
{
ROS_INFO("land enviado ok %d", srv_land.response.success);
}
else
{
  ROS_ERROR("Failed Land");
}
```

```
return 0;
}
```

C.2 ANEXO: Script de MATLAB para representación de gráficas de datos

```
clear all
run datos_rombo.m

%----------------CTUN----------------
figure;
set(gcf,'Name','ALTURA DESEADA Y FILTRADA')

plot(CTUN.data(:,1), CTUN.data(:,6),'g')
hold on;
plot (CTUN.data(:,1), CTUN.data(:,7),'r')
hold on;
plot (CTUN.data(:,1), CTUN.data(:,8),'b')

title ('Altura deseada, filtrada y del barómetro, en metros')
legend ('Altura deseada','Altura filtrada','Altura del barómetro')

%----------------NTUN----------------
figure;
set(gcf,'Name','POSICIÓN EJE LATITUD')

plot(NTUN.data(:,1), NTUN.data(:,3),'g')
hold on;
plot (NTUN.data(:,1), NTUN.data(:,5),'r')

title ('Posición eje latitud real y deseada, en cm')
legend ('Posición eje latitud deseada','Posición eje latitud real')

figure;
set(gcf,'Name','POSICIÓN EJE LONGITUD')

plot(NTUN.data(:,1), NTUN.data(:,4),'g')
hold on;
plot (NTUN.data(:,1), NTUN.data(:,6),'r')

title ('Posición eje longitud real y deseada, en cm')
legend ('Posición eje longitud deseada','Posición eje longitud real')
```

```
figure;
set(gcf,'Name','VELOCIDAD  EJE LATITUD')

plot(NTUN.data(:,1), NTUN.data(:,7),'g')
hold on;
plot (NTUN.data(:,1), NTUN.data(:,9),'r')

title ('Velocidad eje latitud real y deseada, en cm/s')
legend ('Velocidad eje latitud deseada','Velocidad eje latitud real')


figure;
set(gcf,'Name','VELOCIDAD  EJE LONGITUD')

plot(NTUN.data(:,1), NTUN.data(:,8),'g')
hold on;
plot (NTUN.data(:,1), NTUN.data(:,10),'r')

title ('Velocidad  eje longitud real y deseada, en cm/s')
legend ('Velocidad eje longitud deseada','Velocidad eje longitud real')

%----------ATT----------
figure;
set(gcf,'Name','ÁNGULO ROLL')

plot(ATT.data(:,1), ATT.data(:,3),'g')
hold on;
plot (ATT.data(:,1), ATT.data(:,4),'r')

title ('Ángulo roll deseado y real, en grados')
legend ('Ángulo roll deseado','Ángulo roll real')

figure;
set(gcf,'Name','ÁNGULO PITCH')

plot(ATT.data(:,1), ATT.data(:,5),'g')
hold on;
plot (ATT.data(:,1), ATT.data(:,6),'r')

title ('Ángulo pitch deseado y real, en grados')
legend ('Ángulo pitch deseado','Ángulo pitch real')

figure;
set(gcf,'Name','ÁNGULO YAW')

plot(ATT.data(:,1), ATT.data(:,7),'g')
hold on;
plot (ATT.data(:,1), ATT.data(:,8),'r')

title ('Ángulo yaw deseado y real, en grados')
legend ('Ángulo yaw deseado','Ángulo yaw real')
```

```
%----------------IMU----------------
figure;
set(gcf,'Name','ACC EJES X e Y')

plot(IMU.data(:,1), IMU.data(:,6),'g')
hold on;
plot (IMU.data(:,1), IMU.data(:,7),'r')

title ('Aceleraciones en ejes x e y, en m/s/s')
legend ('Aceleración eje X','Aceleración eje Y')

figure;
set(gcf,'Name','ACC EJE Z')

plot(IMU.data(:,1), IMU.data(:,8),'g')

title ('Aceleración en eje z, en m/s/s')
legend ('Aceleración eje Z')

%----------------RCOU----------------
figure;
set(gcf,'Name','SEÑALES PWM MOTORES')

plot(RCOU.data(:,1), RCOU.data(:,3),'g')
hold on;
plot (RCOU.data(:,1), RCOU.data(:,4),'r')
hold on;
plot(RCOU.data(:,1), RCOU.data(:,5),'y')
hold on;
plot (RCOU.data(:,1), RCOU.data(:,6),'b')

title ('Señales PWM enviadas a los motores')
legend ('Canal 1','Canal 2','Canal 3','Canal 4')
```

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá