Carrera
Ingeniería Civil Aeroespacial
Universidad de Concepción

Diseño e implementación de un ecosistema para la validación de algoritmos de decisión autónoma aplicado a RPAs

POR

José Antonio Santamaría Benitez

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil Aeroespacial

Profesor guía:
Bernardo Hernández V.

Profesor patrocinador
Frank Tinapp D.

Noviembre 2025
Concepción, Chile

Agradecimientos

Quiero dar este espacio para agradecer a mi familia, sobre todo mi madre y mi abuela quienes siempre estuvieron para mí en estos 6 años de carrera, a mi abuelo que ya no está, del que herede el ímpetu por querer resolver problemas siempre que algo parecía no tener solución y a mi perrito Duque que ahora desde el arcoíris me hace compañía en este proceso.

A mi pareja por estar apoyándome siempre (y por tenerme paciencia), pese a la distancia que nos separa, quien con su compañía y consejos me ha permitido lograr confiar más en mí mismo y en ser una mejor versión de mí.

A mis amigos y amigas que han sido una parte importante de mi formación, pese a que algunos ya no estén.

A mi comisión por todo su feedback y ayuda a lo largo del proyecto y sobre todo por permitirme entregar en condiciones extraordinarias esta memoria.

Por último, quiero dar una mención especial a mi amiga Pachi, quien fue un pilar super importante a lo largo de mi carrera, sobre todo en este último tramo. Gracias por ser una amiga tan fuerte y leal, he aprendido mucho de ti.

Resumen

El desarrollo de capacidades autónomas en RPAs requiere de entornos de validación seguros y reproducibles que permitan experimentar con algoritmos de decisión antes de su aplicación en vuelo real. En el ámbito académico, las limitaciones operativas y la ausencia de plataformas experimentales integradas dificultan este proceso. Frente a este problema, la presente memoria busca diseñar e implementar un ecosistema modular para la validación de algoritmos de decisión, basado en un autopiloto Pixhawk con firmware PX4 y una Raspberry Pi como computadora embebida.

La metodología se desarrolló en tres etapas i) implementación de un entorno de simulación SITL utilizando PX4 y Gazebo en una máquina virtual, ii) integración de la Raspberry Pi al ecosistema mediante MAVSDK-Python para ejecutar detección de estímulos y maniobras en modo de vuelo autónomo *OFFBOARD* y iii) transferencia de la arquitectura a un RPA físico mediante pruebas en un banco de ensayos. Este enfoque permitió contrastar el desempeño del sistema tanto en simulación como en hardware real.

Los resultados mostraron que SITL reproduce con buena precisión una misión de vuelo real, obteniendo un RMSE de 0.490 [m] respecto a los *waypoints* de referencia, frente a 0.623 [m] del vuelo físico. La Raspberry Pi logró modificar misión y ejecutar maniobras autónomas en ambos entornos. En el banco de ensayos, la estimación de la duración efectiva para las maniobras en modo *OFFBOARD* promedio 1.68 [s] para Roll, 1.73 [s] para Pitch y 2.96 [s] para Yaw, valores ligeramente menores a los definidos en el código (2 y 4 segundos). La ejecución de este mismo código en SITL generó discrepancias importantes, aunque el simulador generó maniobras más suaves, no respetó la lógica temporal programada, por lo que se evidencia que los algoritmos no son completamente transferibles entre entornos y deben tener ajustes específicos.

El ecosistema desarrollado permite validar de forma modular algoritmos complejos aplicados a RPAs, demostrando su viabilidad técnica y proporcionando una base para ensayos de mayor complejidad.

Abstract

The development of autonomous capabilities in RPAs requires safe and reproducible validation environments that enable experimentation with decision-making algorithms prior to their application in real flight. In academic environments, operational constraints and the lack of integrated experimental platforms hinder this process. To address this problem, the present work aims to design and implement a modular ecosystem for validating autonomous decision-making algorithms, based on a Pixhawk autopilot running PX4 firmware and a Raspberry Pi as an embedded companion computer.

The methodology was structured into three stages: i) implementation of a SITL simulation environment using PX4 and Gazebo in a virtual machine, ii) integration of the Raspberry Pi into the ecosystem through MAVSDK-Python to perform stimulus detection and autonomous maneuvers in *OFFBOARD* mode, and iii) transfer of the architecture to a physical RPA through bench-test experiments. This approach enabled a direct comparison of system performance in both simulation and real hardware.

The results showed that SITL reproduces a real flight mission with good accuracy, achieving an RMSE of 0.490 [m] with respect to the reference waypoints, compared to 0.623 [m] obtained in the actual flight. The Raspberry Pi successfully modified missions and executed autonomous maneuvers in both environments. In the bench tests, the effective duration of *OFFBOARD* maneuvers averaged 1.68 [s] for roll, 1.73 [s] for pitch, and 2.96 [s] for yaw, values slightly lower than those defined in the code (2 and 4 seconds). Running this same code in SITL revealed notable discrepancies: although the simulator produced smoother maneuvers, it did not respect the programmed timing logic, indicating that algorithms are not fully transferable between environments and require specific adjustments.

The ecosystem developed in this work enables modular validation of complex algorithms applied to RPAs, demonstrating its technical feasibility and providing a foundation for experiments of greater complexity.

Contenidos

Lista de Figuras

Lista de Tablas

Nomenclatura

RPA:	Remotely Piloted Aircraft
SITL:	Software-in-the-Loop
LTA:	Laboratorio de técnicas Aeroespaciales
MAVLink:	Micro Air Vehicle Link
GCS:	Ground Control Station
HITL:	Hardware-in-the-Loop
QGC:	QGroundControl
VM:	Virtual Machine
MAVSDK:	Micro Air Vehicle Software Development Kit
RMSE:	Root Mean Square Error
UART:	Universal Asynchronous Receiver Transmitter

1 Capitulo 1 Introducción

1.1 Contexto

En los últimos años, los sistemas de aeronaves pilotadas remotamente, comúnmente conocidos por sus siglas en inglés como RPAs, han experimentado una evolución significativa. En el contexto de la industria, han pasado de ser simples plataformas teledirigidas a convertirse en sistemas autónomos complejos con capacidades avanzadas de navegación, percepción del entorno y toma de decisiones [1].

Este avance ha impulsado su adopción en sectores como la vigilancia, monitoreo ambiental, inspección industrial y respuesta ante emergencias [2]. Se estima que el mercado global de drones superara los USD 73 mil millones en 2024 y podría alcanzar USD 163 mil millones en 2030 [3], con una tasa de crecimiento anual compuesto superior al 14%, lo que evidencia su creciente relevancia tecnológica y económica.

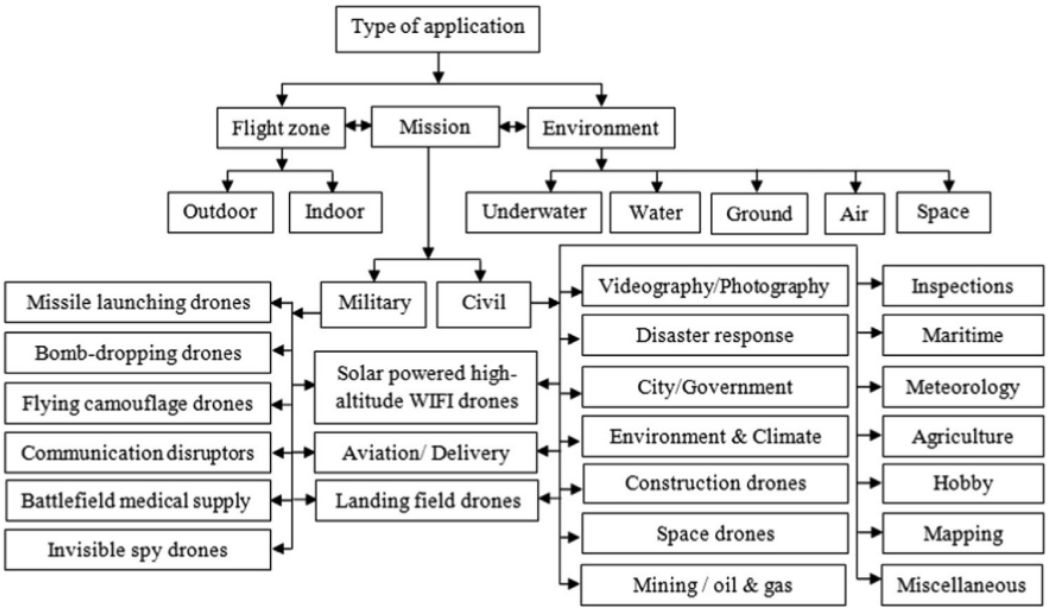


Figura 1: Distintos campos y aplicaciones de los RPA [1]

La Figura 1 muestra la diversidad de aplicaciones en las que actualmente se utilizan RPAs, abarcando tanto entornos civiles como industriales. Cada dominio impone requisitos distintos en autonomía, precisión y confiabilidad, lo que plantea el desafío de integrar hardware, sensores y algoritmos de control capaces de responder ante condiciones externas inesperadas.

No obstante, el desarrollo de estas capacidades requiere de ecosistemas de validación seguros y reproducibles. La normativa aeronáutica, junto con los costos y riesgos asociados

a las pruebas reales de vuelo, limita la experimentación libre en entornos académicos y de investigación [4].

Frente a este escenario, los entonos de simulación *Software-in-the-Loop* (SITL) se han consolidado como una herramienta clave para validar arquitecturas de control y decisión autónoma. En estos sistemas, el modelo dinámico de la aeronave se ejecuta en un entorno numérico, mientras el autopiloto opera como si estuviera en vuelo real [5]. Esta metodología permite evaluar algoritmos de navegación y respuesta ante estímulos externos sin poner hardware físico a riesgo operativo, acelerando el proceso de desarrollo.

1.2 Planteamiento del problema

A pesar del avance significativo en la investigación y desarrollo de las capacidades autónomas de los RPAs, gran parte de los desarrollos documentados en la literatura se enfocan en aplicaciones específicas [6], En contraste, son escasos los trabajos que aborden el diseño de plataformas experimentales versátiles y de código abierto, problema que se vuelve aún más relevante en el ámbito académico, donde las restricciones presupuestarias, regulaciones operacionales y la falta de entornos de prueba estandarizados dificultan la experimentación temprana de algoritmos de decisión autónoma funcionales.

Se propone el diseño e implementación de un ecosistema modular de validación aplicado a RPAs, compuesto por un entorno de simulación SITL, un autopiloto Pixhawk [7] y una Raspberry Pi [8] como sistema embebido de procesamiento. La plataforma contempla su validación mediante un algoritmo de decisión que permita evaluar el flujo de información y respuesta del sistema ante eventos definidos, con la proyección de ser implementada y evaluada en pruebas reales. De este modo se busca sentar una base técnica escalable que facilite el desarrollo futuro de pruebas con algoritmos de control más complejos.

1.3 Objetivos

1.3.1 Objetivo general

Diseñar e implementar un ecosistema modular para la validación de algoritmos de decisión autónoma en RPAs, basado en la integración de un autopiloto Pixhawk y una Raspberry Pi.

1.3.2 Objetivos específicos

OE1: Implementar un entorno de simulación SITL con PX4, evaluando su funcionamiento mediante la ejecución de misiones predefinidas y telemetría básica.

OE2: Integrar una Raspberry Pi como sistema embebido de procesamiento dentro del ecosistema SITL, ejecutando un algoritmo de decisión autónoma que modifique parámetros de misión en función de algún estimulo externo.

OE3: Validar el ecosistema en un RPA real, evaluando la capacidad del sistema para ejecutar decisiones autónomas bajo condiciones reales

1.4 Condiciones de diseño

Esta memoria de título se desarrolla bajo un conjunto de condiciones de diseño que delimitan su alcance y establecen el marco en el cual se desarrolla la propuesta. Estas condiciones están definidas principalmente por la disponibilidad de hardware en el Laboratorio de Técnicas Aeroespaciales (LTA), lo que orienta la elección de las plataformas de control y de los entornos de prueba. Se trabajará con un autopiloto Pixhawk 6C, con firmware PX4 y una Raspberry Pi 4B, utilizada como computadora embebida, ambos comunicados mediante el protocolo MAVLink [9]. Para la validación experimental, las pruebas en entorno simulado (SITL) se realizan sobre Ubuntu Linux utilizando el simulador de físicas Gazebo como motor principal de físicas del RPA, permitiendo reproducir misiones y comportamientos antes de su ejecución en hardware real.

La validación seguirá un esquema progresivo: i) Pruebas en SITL, ii) integración de la Raspberry Pi en entorno simulado y iii) transferencia de la arquitectura a un RPA físico en condiciones controladas. El sistema se considerará exitoso si es capaz de ejecutar misiones autónomas en SITL, modificar parámetros de vuelo en función de un algoritmo de decisión y replicar dicho comportamiento en al menos una prueba con hardware real.

El alcance se limita a la implementación y validación de dicho ecosistema mediante un algoritmo de decisión autónoma. De forma opcional, se evaluará la incorporación de un modelo de aprendizaje automático como prueba exploratoria, con el fin de analizar la robustez de la arquitectura para soportar algoritmos de mayor complejidad. La elaboración de escenarios operativos avanzados queda fuera del marco de esta memoria y se proyectan como posibles líneas futuras de investigación.

Asimismo, se espera que el presente informe sirva como una documentación técnica accesible para apoyar a estudiantes en el desarrollo de futuros algoritmos de prueba aplicados a RPAs.

1.5 Descripción del trabajo

El presente proyecto se organiza en etapas escalables, alineadas con los objetivos específicos y con la planificación temporal definida en la carta Gantt, la cual se incluye en el anexo 1.

En primer lugar, se desarrolla una revisión bibliográfica orientada a identificar arquitecturas y metodologías previas que trabajen en la integración de autopilotos, computadoras embebidas y protocolos de comunicación. Esta fase se llevará a cabo en los primeros meses del proyecto e incluye la recolección y análisis de artículos científicos. Su propósito es tener una base conceptual que guie el diseño del ecosistema propuesto. La lectura de artículos académicos es una actividad que se llevara a cabo a lo largo de todo el desarrollo del proyecto.

La segunda etapa corresponde a la implementación del entorno de simulación SITL, lo que permite ejecutar misiones autónomas y probar comunicación mediante Python/MAVLink. Este paso es el que corresponde al primer objetivo específico (**OE1**) y establece un marco de pruebas seguro y reproducible para las fases posteriores.

Posteriormente, se incorpora la Raspberry Pi como computadora embebida al ecosistema simulado, con el propósito de evaluar el flujo de datos y la ejecución de un algoritmo de decisión capaz de interactuar con el autopiloto mediante MAVLink. Esta fase se vincula al segundo objetivo específico (**OE2**). También se contempla un análisis de los parámetros en las simulaciones, con el fin de poder cuantificar la capacidad de detección de estímulos del sistema. La integración completa de la Raspberry Pi se proyecta como entregable para el informe final.

Finalmente, la arquitectura validada en la simulación se trasladará a un RPA físico equipado con un autopiloto Pixhawk y Raspberry Pi, con el propósito de comprobar el desempeño del sistema en condiciones reales de operación y contrastar los resultados con los obtenidos en la simulación. Previo a la ejecución de misiones de vuelo, se considera el uso del banco de ensayos disponible en el LTA [10], lo que permitirá verificar la respuesta del sistema en un entorno controlado, ayudando a reducir riesgos en la fase experimental y será parte de la validación planteada en el tercer objetivo específico (**OE3**), asegurando coherencia con el objetivo general del proyecto.

1.6 Metodología

Para el diseño metodológico del ecosistema de validación, con el fin de garantizar la replicabilidad, cada etapa de investigación se sustentará de plataformas de código abierto y herramientas estandarizadas, de modo que los procedimientos puedan ser reproducidos por terceros. La estrategia de trabajo considera que cada fase genere datos comparables y verificables, de modo de contrastar el desempeño del sistema en simulación y en plataforma real.

1.6.1 Entorno simulado

(I)

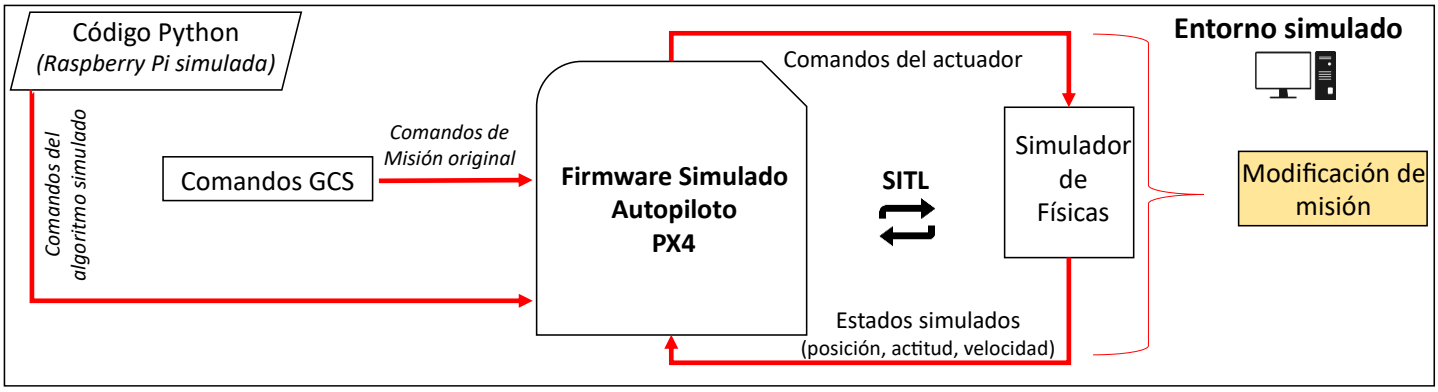


Figura 2: Esquema de pruebas en SITL

El diagrama de la Figura 2 representa el flujo de información en el entorno simulado. El autopiloto PX4 se ejecuta en firmware simulado y se conecta al simulador de físicas para reproducir la dinámica de vuelo. Los comandos originales de la misión provienen de la estación de control en tierra (GCS), mientras que un código en Python, simula la función de la Raspberry Pi modificando la misión en tiempo real.

1.6.2 Integración de la Raspberry Pi en SITL

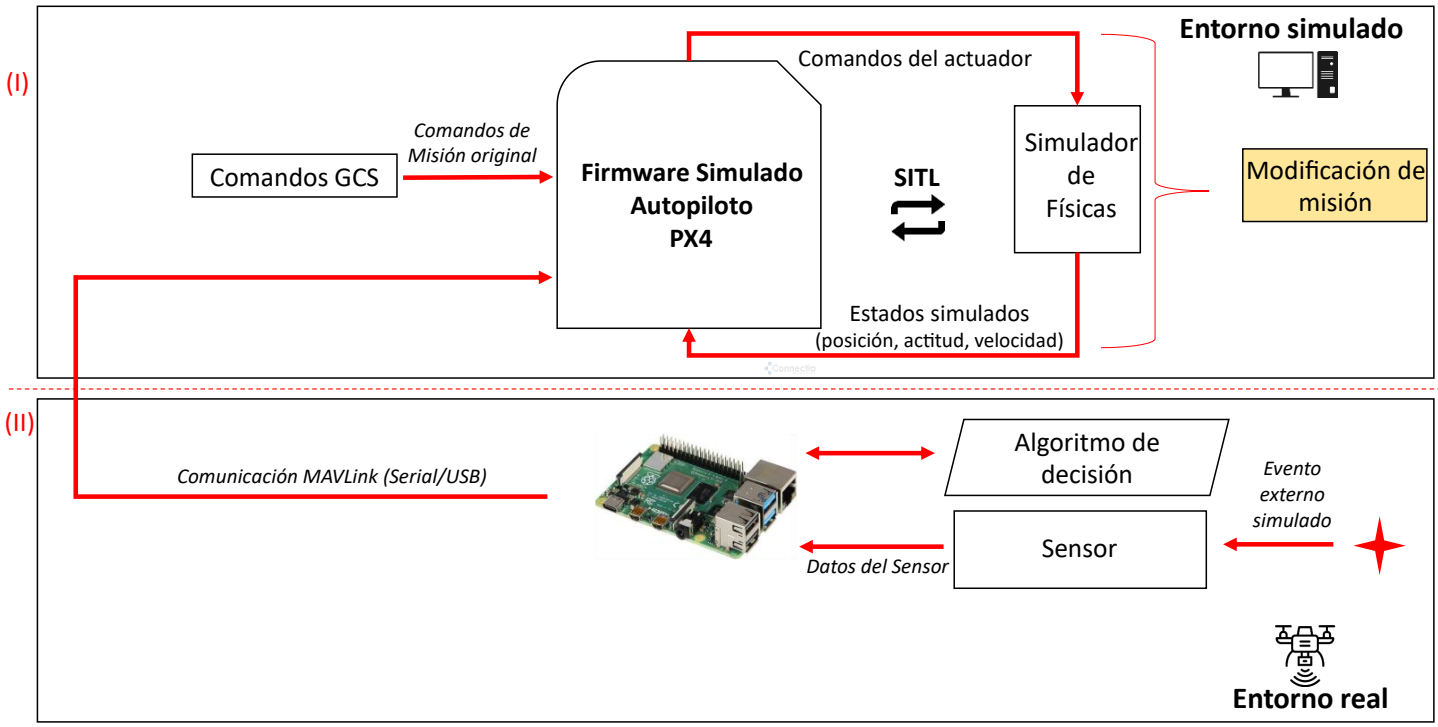


Figura 3:Integración de la Raspberry Pi en el entorno simulado

La Figura 3 representa el esquema extendido a la incorporación de la Raspberry Pi, la cual ejecuta un algoritmo de detección que recibe datos del sensor simulado. En esta configuración, los sensores simulados entregan datos sintéticos que emulan lecturas de dispositivos reales, como cámaras, mientras que los eventos externos simulados corresponden a condiciones programadas que desencadenan una respuesta del algoritmo. La comunicación de la Rasperry Pi con el autopiloto será mediante MAVLink, transmitido sobre un enlace serial.

1.6.3 Transferencia a hardware físico

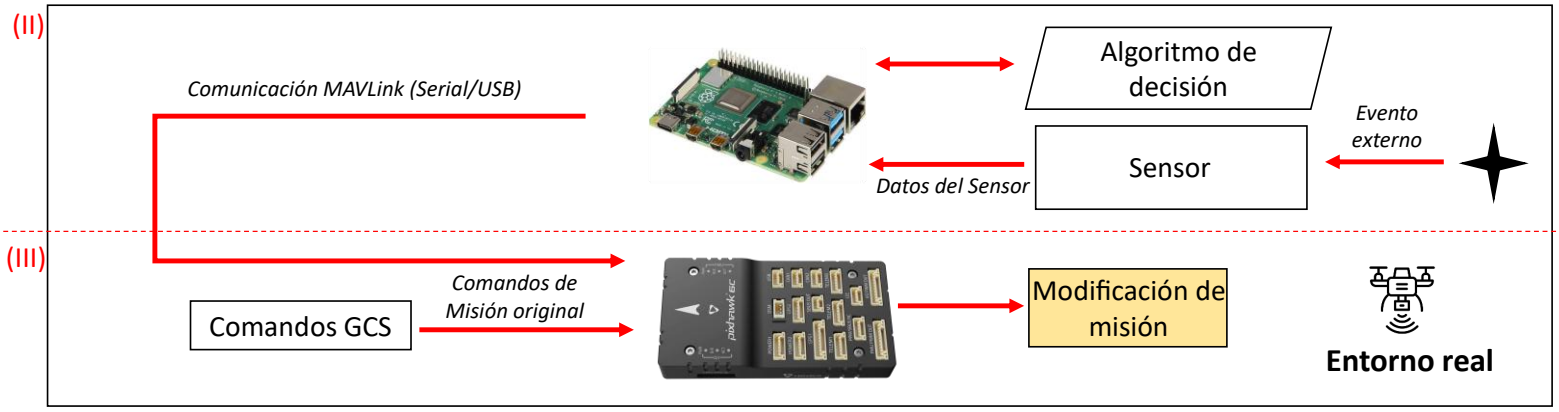


Figura 4: Transferencia de la arquitectura a Hardware físico.

El esquema de la Figura 4 muestra la transferencia del ecosistema a el entorno real. La Raspberry Pi se conecta físicamente al autopiloto Pixhawk, replicando el mismo flujo de datos validado previamente en SITL. Se espera que el sistema sea capaz de modificar la misión durante un vuelo en condiciones reales y controladas.

2 Capítulo 2: Marco teórico

Para comprender de mejor manera el contenido de este proyecto, se establecen las bases teóricas y definiciones críticas que son esenciales para su comprensión.

2.1 RPA y Autopiloto

Los RPA (Remotely Piloted Aircraft) corresponden a aeronaves no tripuladas controladas por pilotos a distancia, o bien dotadas de capacidades de autonomía parcial mediante un sistema de control de vuelo. En la normativa internacional, especialmente la establecida por la Organización de Aviación Civil Internacional [11], el termino RPA se usa para diferenciar este tipo de aeronaves de los drones de uso recreativo, asociándolo a aplicaciones profesionales, como las que se ilustran en la Figura 1. En el caso de este proyecto, el RPA utilizado es el X8-Quadcopter [12], mostrado en la Figura 5. Este modelo fue concebido originalmente como un vehículo de pruebas capaz de integrar distintos controladores de vuelo, sensores y configuración de carga útil. El modelo liberado permite reproducir un RPA económico, estable y fácilmente modificable para validar nuevas aplicaciones en un contexto académico.

Figura 5: RPA X8-Quadcopter

El cerebro de un RPA moderno es el controlador de vuelo, también conocido como autopiloto. Es el sistema electrónico encargado de estabilizar la aeronave y ejecutar maniobras básicas de vuelo.

Para evitar confusiones, se adopta la perspectiva del autopiloto como controlador, donde las lecturas de los sensores (actitud, velocidad y posición) se consideran sus entradas, mientras que las señales de control generadas por el propio autopiloto para regular la potencia de los motores corresponden a sus salidas. Estas señales, a su vez, actúan como entradas físicas del vehículo, generando el movimiento observable en el RPA [13].

Figura 6: Autopiloto Pixhawk 6C

En el contexto de este proyecto, se destacan los autopilotos de código abierto PX4 y Ardupilot, los cuales han favorecido la investigación y el desarrollo gracias a su flexibilidad y amplia comunidad desarrolladora. El hecho de ser código abierto, les permite modificar parámetros de control, incorporar algoritmos externos y realizar pruebas en entornos simulados, convirtiéndolos en plataformas idóneas para proyectos académicos y aplicaciones experimentales [14], [15].

De este modo, el concepto de RPA y autopiloto no se limitan únicamente al control básico de vuelo, sino que constituyen la base sobre la cual se integran y validan nuevas aplicaciones de desarrollo. Esta perspectiva resulta fundamental para el presente proyecto, cuyo eje es la validación de arquitecturas que extienden las capacidades del autopiloto en la toma de decisiones.

2.2 Protocolo de comunicación MAVLink

La comunicación es un componente esencial en los RPA, pues permite el intercambio continuo de información entre el autopiloto y la estación en tierra. En este contexto, el protocolo más extendido es MAVLink [9] (Micro Air Vehicle Link), elaborado originalmente en 2009 y adoptado como estándar en autopilotos de código abierto como Ardupilot y PX4.

MAVLink es un protocolo de mensajería binaria ligera, diseñado para transmitir telemetría, comandos y parámetros de configuración en tiempo real. Su eficiencia radica en que utiliza mensajes binarios compactos, lo que reduce el ancho de banda necesario para transmitir información de estado y comandos. Gracias a este diseño ligero, MAVLink puede operar sobre distintos medios físicos, como módulos de radio en 433/915 MHz y también sobre redes modernas basadas en TCP o UDP (protocolos de transporte Ethernet o WiFi). La comunicación es bidireccional, de modo que el controlador envía información de estado y recibe ordenes de control.

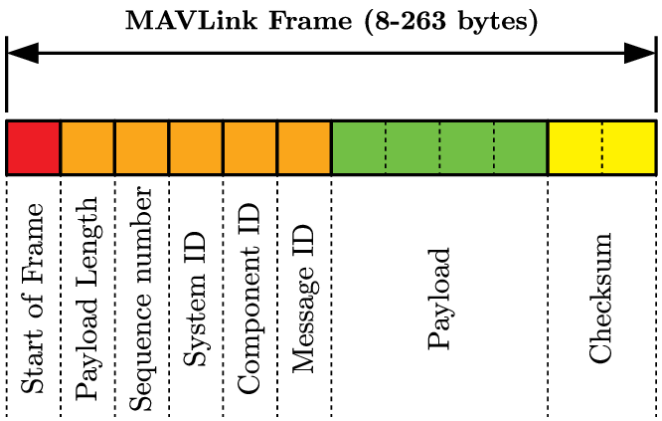


Figura 7: Estructura de mensaje MAVLink [16]

La Figura 7 muestra la estructura general de un mensaje MAVLink, cuya longitud puede variar entre 8 y 263 bytes. Cada mensaje incluye un encabezado con información de control, como el identificador del sistema y del componente junto con el tipo de mensaje, seguido de la carga útil o payload, que contiene los datos transmitidos y un campo de verificación que asegura la integridad del paquete.

Esta organización compacta permite que el autopiloto, la Raspberry Pi y la estación en tierra intercambien información en tiempo real, diferenciando tanto el origen de los datos como el tipo de instrucción enviada, lo que posibilita establecer flujos de comunicación estables tanto en el entorno simulado como en hardware físico.

Para aclarar, aunque MAVLink se utiliza frecuentemente sobre enlaces seriales, es importante distinguir entre el medio físico de transmisión y el protocolo en sí. La comunicación serial define únicamente la forma en que los bits se envían entre los dispositivos, mientras que MAVLink establece la estructura lógica de los mensajes y su interpretación (Figura 7).

2.3 Sistemas embebidos

Un sistema embebido es hardware y software orientado a una función específica dentro de un dispositivo mayor, con recursos limitados y requisitos de respuesta en tiempo real. Su arquitectura típica integra procesador, memoria, entradas/salidas y firmware que idealmente garantice baja latencia y alta confiabilidad [17]. Esta definición lo distingue de los computadores de propósito general y explica su alta eficiencia en áreas como la industria automotriz, aeroespacial y telecomunicaciones.

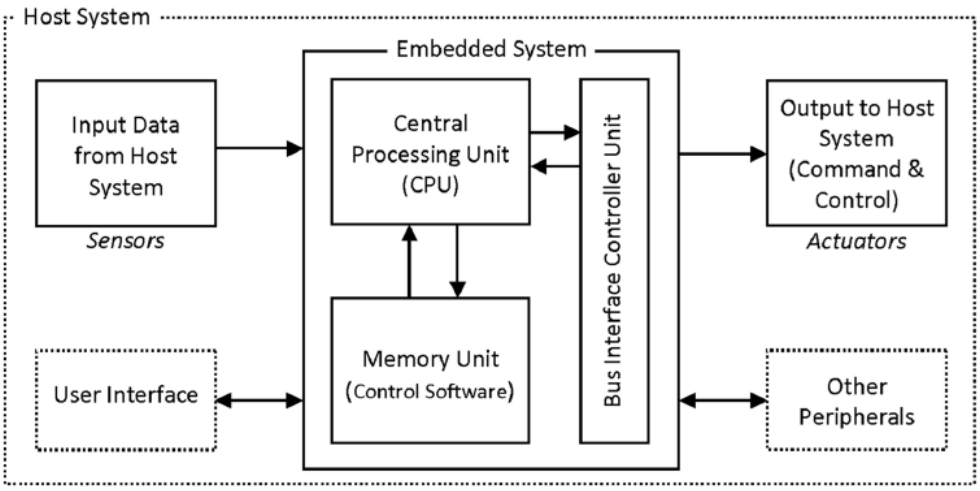


Figura 8: Componentes funcionales de un sistema embebido [18]

En el caso de los RPA, el sistema embebido actúa en complemento a él autopiloto, aportando una capa adicional de percepción y decisión. La forma estándar es la *companion computer* conectada al autopiloto vía serial o Ethernet, comunicándose mediante MAVLink, la premisa principal es la separación de funciones, lo que le permite ejecutar lógica avanzada sin interactuar directamente con la capa de control primario del autopiloto.

Bajo esta arquitectura, la Raspberry Pi, un microordenador de bajo costo y tamaño reducido diseñado por la Raspberry Pi Foundation [19], ampliamente utilizado en el ámbito académico es capaz de ejecutar Linux como sistema operativo, recibir telemetría, procesar información de sensores y enviar comandos al autopiloto. La integración se realiza con enlaces UART o USB-serial y en simulación sobre TCP/UDP, siendo el flujo de información el mismo a nivel de protocolo [20] lo que permite desarrollar algoritmos en SITL y luego transferirlos a hardware sin la necesidad de reescribir la mensajería.

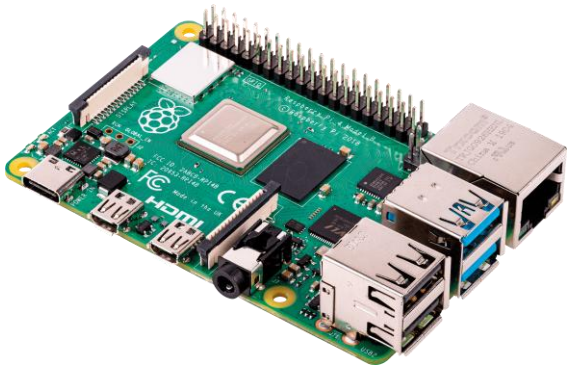


Figura 9: Raspberry Pi 4B

En comparación con otros computadores embebidos y en lo encontrado en la literatura, la Raspberry Pi ofrece un equilibrio favorable entre costo, consumo energético y disponibilidad de librerías de desarrollo, lo que la convierte en una opción practica para el desarrollo de este proyecto.

2.4 Técnicas de simulación

Dentro de las técnicas de simulación estudiadas, se destacan los enfoques *Software-in-the-Loop (SITL)* y *Hardware-in-the-Loop* (HITL) [5], ampliamente documentados en aplicaciones de robótica autónoma.

SITL consiste en ejecutar el mismo firmware del autopiloto en un entorno de simulación, donde la dinámica de la aeronave es representada mediante un modelo matemático o físico simulado, de manera que el autopiloto procesa los datos como si proviniesen de sensores reales y genera salidas que son aplicadas al modelo. En esta memoria, SITL se constituye como la técnica principal de validación.

HITL extiende el concepto anterior al incorporar dispositivos físicos reales dentro del ecosistema simulado, lo que permite probar de manera directa la comunicación entre el autopiloto y componentes externos bajo condiciones de simulación. Si bien HITL ofrece un mayor grado de realismo en las pruebas, implica también un aumento significativo en términos de complejidad y tiempo por lo que, dado el alcance de este proyecto, no se contempla su uso.

2.5 Herramientas de desarrollo

Las herramientas de desarrollo seleccionadas permiten integrar simulación, comunicación, planificación de misiones y control de vuelo de manera coherente. Cada una aporta funcionalidades específicas que en conjunto respaldan la arquitectura modular propuesta.

2.5.1 QGroundControl

QGroundControl (QGC) es una estación de control en tierra (GCS) de código abierto, compatible con vehículos que usan MAVLink [21]. Proporciona interfaz gráfica para configurar parámetros de vuelo, planificar misiones y visualizar telemetría en tiempo real. QGC tiene soporte multiplataforma (Windows, Linux, Android, iOS), la elección por sobre otras GCS es debido a que es el software de referencia oficial para PX4, con una interfaz más simple y mayor compatibilidad con Gazebo y flujos de trabajo en Ubuntu.

Además, desde QGC se pueden visualizar la lista de parámetros del autopiloto y descargar los logs de las misiones de vuelo, facilitando el análisis de los datos tanto de la simulación como del autopiloto físico.

Figura 10: Interfaz de QGC mostrando una misión en Carriel Sur.

2.5.2 Ubuntu Linux

Para trabajar en el entorno simulado, se escogió a Ubuntu [22], una distribución de Linux ampliamente usada diseñada para ofrecer un entorno estable y con buen soporte de software libre. Entre sus características se incluyen actualizaciones regulares, gran repositorio de librerías y paquetes, además de compatibilidad con herramientas de robótica como ROS (Robot Operating System) y un ecosistema robusto para el desarrollo centrado en ingeniería electrónica [23].

La elección de Ubuntu en este proyecto responde a la necesidad de un entorno operativo estable y reproducible, capaz de ejecutar de forma confiable los simuladores SITL, Gazebo y las interfaces de comunicación. Otra ventaja es que la misma base de software puede utilizarse tanto en la estación de simulación como en la Raspberry Pi, lo que reduce la probabilidad de inconsistencias al transferir algoritmos desde la simulación al hardware físico.

Para aislar el entorno y evitar conflictos con el sistema operativo anfitrión, Ubuntu se ejecuta en una máquina virtual (VM) dedicada, como se visualiza en la Figura 11. Esto permite disponer de un entorno en el que se puedan instalar las dependencias especificas del proyecto sin afectar otros programas del computador host. Además, el uso de una VM facilita la portabilidad, ya que la configuración puede ser replicada en diferentes equipos, asegurando coherencia en las pruebas y continuidad en el desarrollo.

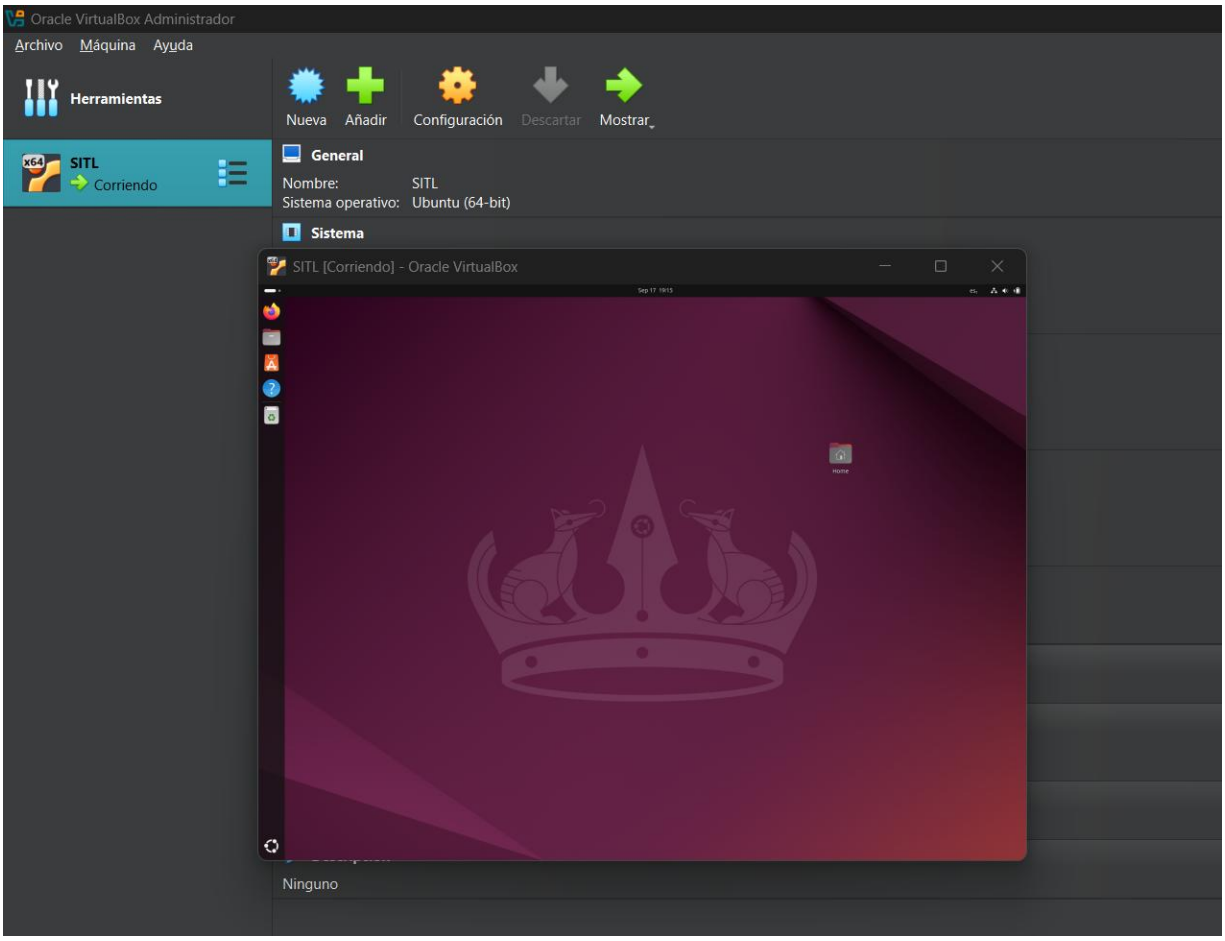


Figura 11: Ubuntu Linux corriendo en la máquina virtual (VM)

2.5.3 Gazebo

Gazebo es un simulador robótico de código abierto que permite recrear entornos tridimensionales con modelos dinámicos de vehículos, sensores y actuadores. Su motor físico incorpora características como gravedad, fricción, colisiones y aceleración, facilitando reproducir escenarios cercanos a la realidad en términos de comportamiento y respuesta a variables de entrada convirtiéndolo en una fuerte herramienta para la validación de sistemas autónomos [24]. El software es compatible de manera nativa con autopilotos de código abierto como lo es PX4 [25], a través del modo SITL, lo que permite simular las físicas del hardware real dentro de un entorno controlado.

En el presente proyecto, Gazebo se considera una herramienta indispensable para la simulación de las físicas del RPA, gracias a su documentado desarrollo con Ubuntu Linux, PX4 y QGC, lo que facilita el desarrollo del ambiente simulado. No obstante, su uso se plantea con alcance definido, sin profundizar en el desarrollo ni creación de modelos de dinámica avanzada en el desarrollo de escenarios complejos, siendo utilizado principalmente como plataforma de apoyo para el desarrollo de SITL, delimitando el foco central del trabajo hacia los objetivos plantados en la sección 1.3.

2.5.4 MAVSDK

MAVSDK (Micro Air Vehicle Software Development Kit) [26] es un conjunto de librerías que proporciona una interfaz de alto nivel para interactuar con vehículos aéreos que utilizan el protocolo MAVLink. Su propósito es simplificar la comunicación entre aplicaciones externas y el autopiloto permitiendo a los desarrolladores acceder a telemetría, enviar comandos y ejecutar misiones.

A diferencia de interactuar con los comandos MAVLink de manera directa, lo que requiere gestionar manualmente la serialización de mensajes, MAVSDK abstrae esta complejidad y ofrece acceso estructurado a funcionalidades como control de vuelo, planificación de misiones, manejo de cámaras y transmisión de parámetros. Está disponible en múltiples lenguajes de programación, lo que facilita la integración en diferentes plataformas.

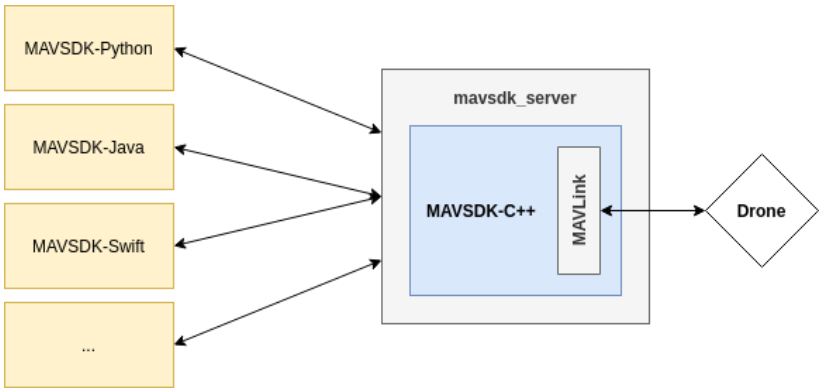


Figura 12: MAVSDK y su compatibilidad con distintos lenguajes [26]**.**

En el contexto de este proyecto, MAVSDK se emplea como la interfaz principal para establecer comunicación entre la Raspberry Pi y el autopiloto PX4. Esto permite que los algoritmos de decisión desarrollados en Python puedan interactuar con el vehículo en tiempo real. Además, al ser compatible tanto en simulación como en hardware real, MAVSDK asegura la continuidad del flujo de pruebas.

2.5.5 Lenguaje Python y repositorio GitHub

El desarrollo de todo el código, algoritmos y su integración con el autopiloto se realizará principalmente en Python [27], debido a su versatilidad, simplicidad sintáctica y amplia comunidad de soporte. Además, este lenguaje cuanta con diversas librerías que facilitan la interacción con el protocolo MAVLink, así como también herramientas para el análisis y visualización de los datos.

GitHub [28] será utilizado principalmente como herramienta de consulta, aprovechando su activa comunidad desarrolladora y la disponibilidad de repositorios públicos relacionados con MAVSDK y PX4. Su uso se centrará en la búsqueda de ejemplos, resolución de problemas de compatibilidad de versión y depuración de errores.

3 CAPITULO 3: Estado del arte

Las aplicaciones y desarrollo de sistemas autónomos en RPAs han evolucionado desde plataformas cerradas hacia arquitecturas más abiertas. En este capítulo se revisan trabajos recientes que exploran ámbitos como la navegación autónoma, la toma de decisiones a bordo y la integración de sistemas embebidos con autopilotos de código abierto, particularmente PX4. Las aplicaciones de los estudios varían entre sí, difiriendo en objetivos y contexto, pero mantienen el propósito de ampliar las capacidades del RPA, permitiendo identificar un marco de referencia sobre el cual se fundamenta la presente memoria.

3.1 Design and implementation of autonomous quadcopter using SITL Simulator

En este estudio se presenta el diseño e implementación de un cuadricóptero autónomo empleando el entorno SITL como etapa inicial de validación. La motivación del estudio es emplear un sistema de delivery de mensajes utilizando un companion PC, autopiloto Pixhawk con firmware Ardupilot y QGC como estación de control.

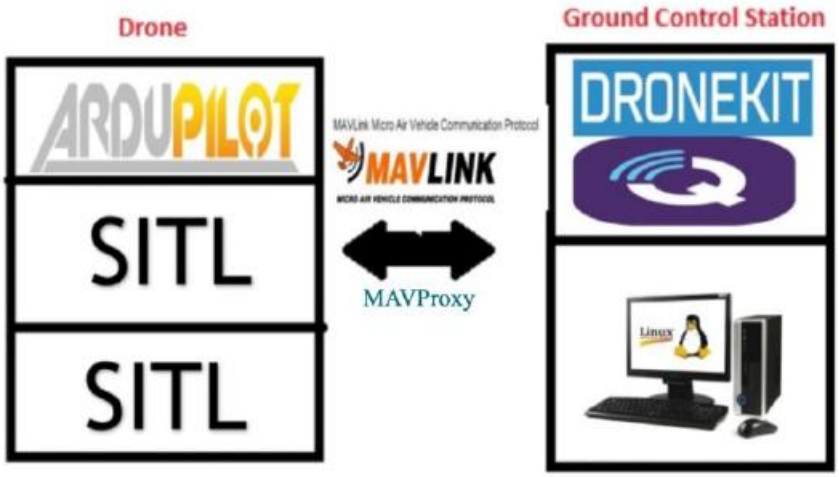


Figura 13: Diagrama de simulación SITL

El trabajo consistió en la puesta en marcha del del simulador SITL, para ejecutar misiones predefinidas y validar la comunicación con un computador embebido mediante MAVLink. Pese a que los resultados son poco concluyentes con respecto al propósito inicial propuesto por el proyecto, se demostró que SITL es una herramienta efectiva para determinar errores de configuración y latencia, ayudando a depurar algoritmos antes de llegar al vuelo real, lo que redujo considerablemente los tiempos de desarrollo.

El aporte principal de la investigación es la validación del flujo metodológico basado en SITL como plataforma de ensayo previo, confirmando que los algoritmos probados en simulación pueden ejecutarse en hardware sin la necesidad de modificar el código, lo que respalda la decisión de elegir SITL como base del ecosistema a implementar.

3.2 Software and Hardware-in-the-loop Verification of Flight Dynamics Model and Flight Control Simulation of a Fixed-wing UAVs [5]

En este trabajo se desarrolla una estrategia de validación de modelos de dinámica y control de UAVs de ala fija utilizando esquemas combinados de SITL y HITL. El propósito fue evaluar tanto el software de control como la interacción con hardware real antes de realizar vuelos experimentales.

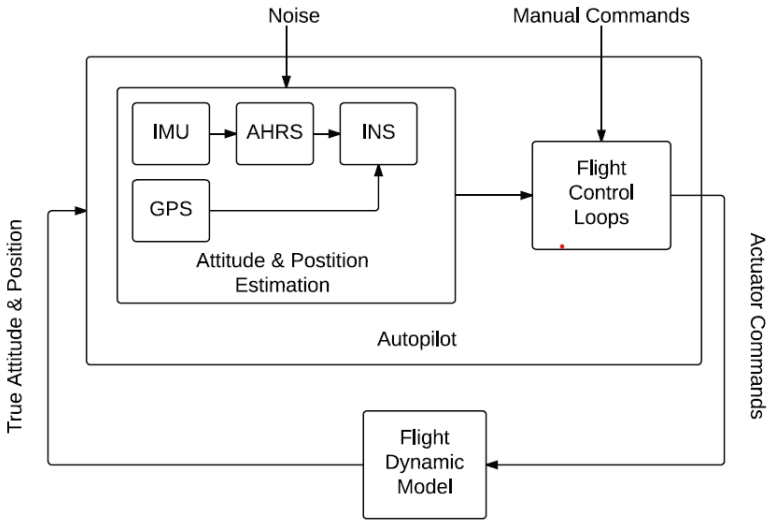


Figura 14: Esquema SITL del bloque de pruebas

El estudio implemento primero el autopiloto y la dinámica de la aeronave en un entorno puramente SITL (Figura 14), lo que permitió realizar ajustes preliminares de controladores y parámetros. Posteriormente, se incorporó el hardware del autopiloto en un esquema HITL, recibiendo y enviando señales reales al simulador, los resultados mostraron que este enfoque reduce discrepancias entre simulación y vuelo real, manteniendo el error en un margen de un 10%, pese a que los vuelos de prueba realizados fueron en un día ventoso.

Aunque el presente proyecto no contempla implementar HITL, se confirma con los resultados que SITL es suficiente para validar lógica de comunicación y control en etapas tempranas de desarrollo. En términos de escalabilidad, HITL representa un paso intermedio valioso, pues permite introducir el hardware real en el lazo de simulación antes de llegar al vuelo físico, sin embargo, en el caso de este proyecto al contar con el banco de ensayos, se puede validar el hardware real sin mayor riesgo a la hora de realizar las pruebas.

3.3 Low-Cost Computer-Vision-Based Embedded Systems for UAVs [29]

Este trabajo surge de la necesidad de dotar a los RPAs de sistemas auxiliares de percepción que permitan ejecutar maniobras críticas, como el aterrizaje autónomo la evasión de obstáculos, sin comprometer la autonomía de vuelo, ni incrementar de forma significativa el peso total del RPA. La motivación se enmarca en escenarios de riesgo ambiental como el monitoreo remoto de zonas volcánicas.

La propuesta metodología del estudio consiste en la implementación de un sistema de detección y evasión de obstáculos (LOAS) y de aterrizaje asistido por visión, utilizando

únicamente hardware accesible, contando con una Raspberry Pi 4 como computadora embebida, una cámara NoIR y un sensor de distancia LIDAR Lite v3, donde utilizan técnicas de visión por computadora livianas para la detección, minimizando el consumo de batería.

El aporte más significativo de esta investigación es el diseño del sistema de aterrizaje asistido o como se denomina en el artículo *visión-based landing* que integra cuatro etapas. La primera establece una altitud segura para iniciar el descenso, verificando constantemente el estado de aterrizaje del controlador (LANDED_STATE en MAVLink) y corrigiendo la altura si esta fuera de rango. La segunda etapa implementa la detección de un helipuerto mediante visión, identificando un código QR que contiene la palabra HELIPAD. Una vez detectado, se inicializa un rastreador visual que centra el objetivo em el eje de la cámara, aplicando una técnica denominada *pure pursuit* para guiar la aproximación. Finalmente, el sistema ejecuta un descenso controlado sobre la plataforma.

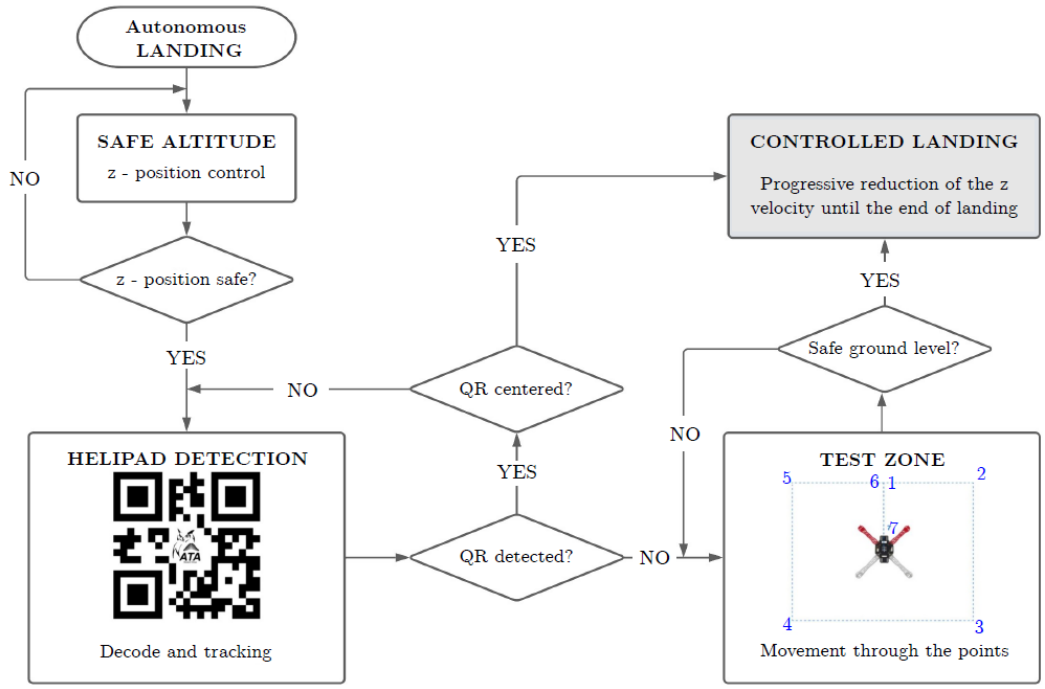


Figura 15: Algoritmo de aterrizaje guiado

Los resultados mostraron que el sistema podía ejecutar aterrizajes precisos en un 90% de los ensayos, reduciendo errores de posición respecto al GPS/Imu del autopiloto y manteniendo un consumo energético cercano a los 3.5 W. Esto evidencia que la visión por computadora aplicada a RPAs no requiere de técnicas avanzadas de inteligencia artificial ni hardware especializado de alto costo.

En el contexto de esta memoria, se validará la posibilidad de desarrollar un sistema similar de aterrizaje autónomo para las pruebas en hardware real, basado en una cámara RGB. El procesamiento de imagen se realizará utilizando la librería OpenCV, cuyo consumo energético se mantiene por debajo de 5 [W] [30], comparable a la reportada en el estudio.

3.4 High-Level Modular Autopilot Solution for Fast Prototyping of Unmanned Aerial System [31]

En este proyecto, se presenta una arquitectura modular denominada MASPX4, diseñada para acelerar el prototipado de algoritmos complejos en UAVs y al mismo tiempo, garantizar redundancia y seguridad en vuelo. La motivación principal surge de las limitaciones que presentan los autopilotos de código abierto como PX4, los que, si bien son altamente confiables para el control básico, la integración de algoritmos avanzados suele requerir modificar el código fuente, lo cual no solo puede perjudicar su desempeño en vuelo, sino que también se traduce en largos tiempos de compilación y depuración.

El sistema combina una Raspberry Pi con NAVIO2, que ejecuta el autopiloto PX4 como respaldo, siendo el encargado de las funciones de control básicas del RPA, un mini computador Intel NUC el cual corre Simulink y permite implementar directamente algoritmos avanzados sin necesidad de simplificarlos. La comunicación entre los sistemas es mediante una conexión UDP robusta sobre Ethernet, lo que asegura baja latencia. Adicional a todo esto, el diseño incluye un mecanismo que en caso de fallo, transfiere el control al autopiloto PX4 lo que refuerza la confiabilidad del sistema.

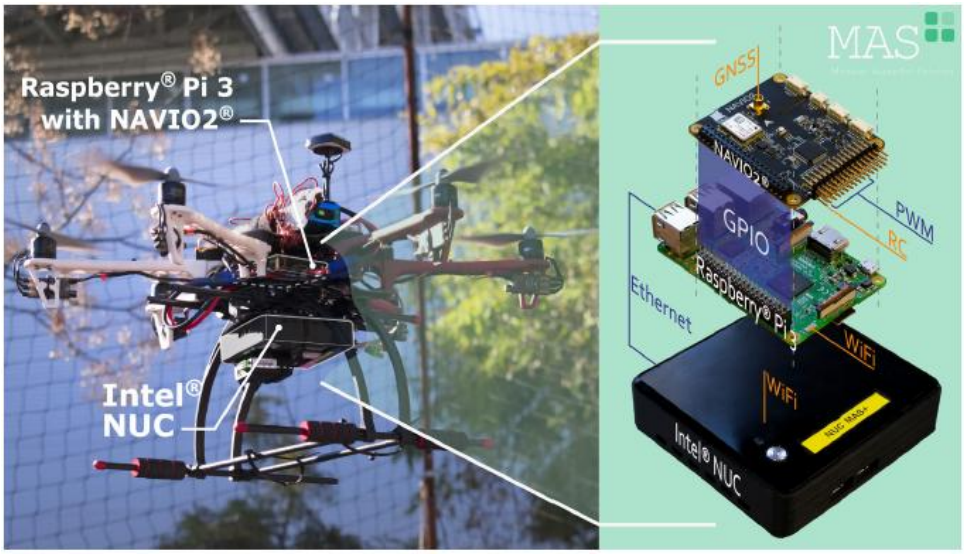


Figura 16: Plataforma de validación en vuelo con vista en detalle de la configuración de hardware

Los resultados experimentales en más de 50 horas de vuelo real mostraron que la solución propuesta redujo los tiempos de prototipado entre un 50-75%. Entre los casos evaluados se incluyen un controlador PID en cascada, un controlador linear y un módulo de navegación autónoma, todos ejecutados bajo condiciones de vuelo similares.

La relevancia de este estudio para la presente memoria es doble. Primero, confirma la viabilidad de arquitecturas hibridas en las que un sistema de cómputo adicional aporta flexibilidad al autopiloto y potencia para integrar nuevas funcionalidades. Adicionalmente, se demuestra que este enfoque reduce de manera significativa los tiempos de desarrollo y facilita la experimentación con los RPA, aspecto clave en ámbitos académicos.

4 CAPITULO 4: Entorno Simulado

4.1 Puesta en marcha del entorno simulado

Para la implementación del entorno SITL fue necesario instalar un software capaz de ejecutar una máquina virtual con Ubuntu Linux como sistema operativo base. Inicialmente, se consideró replicar el procedimiento descrito en 3.1, el cual utilizaba el firmware ArduCopter como autopiloto simulado y la biblioteca DroneKit como interfaz de control externo. Sin embargo, foros y la página oficial de Dronekit en GitHub [32], indican que su desarrollo activo fue interrumpido desde el 2017-2019, lo que implica que se dejó de ofrecer soporte para versiones recientes de Python y repositorios modernos.

Durante pruebas preliminares, la instalación de las dependencias de Dronekit produjo errores de compilación y conflictos con las librerías base de Python 3.9 y superiores, lo que dificultaba mantener un entorno estable, siendo un problema si se piensa extender el sistema a la computadora embebida.

Ante estas limitaciones, se optó por utilizar MAVSDK-Python, descrita en 2.5.4, oficialmente mantenida por el ecosistema MAVLink y ofreciendo las mismas capacidades de comunicación y control que Dronekit, pero con soporte continuo para versiones actuales de Python.

Debido a que MAVSDK mantiene compatibilidad nativa con el firmware PX4, mientras que Dronekit está enfocado en el uso exclusivo de ArduPilot, la adopción de MAVSDK implicó migrar el desarrollo hacia PX4 como firmware de autopiloto principal. Esta decisión resulto fundamental a la hora de comenzar a desarrollar el proyecto pues permitió contar con documentación actualizada y acceso a una comunidad desarrolladora activa, facilitando la ejecución de etapas posteriores del proyecto.

4.1.1 Configuración base de la máquina virtual (VM)

El entorno se configuró sobre Oracle VirtualBox [33], el cual es un software de código abierto que permite a sus usuarios ejecutar múltiples sistemas operativos en un mismo computador, creando máquinas virtuales, los cuales son computadoras basadas en software que pueden ejecutar sistemas operativos, como Linux, distintos al del computador nativo.

El entrono se configuró con Ubuntu 22.04 LTS de 64 bits, con una asignación de 4 núcleos de CPU, 8GB de memoria RAM y 50 GB de almacenamiento dinámico. Estas especificaciones permitieron asegurar un rendimiento estable, sobre todo en términos de memoria pues las bibliotecas utilizadas y logs descargados consumen un espacio considerable dentro del disco.

Se empleo una instalación limpia del sistema operativo con las siguientes dependencias iniciales, visualizadas en el listado 1 que permiten establecer la base donde se ejecutara SITL.

```
sudo apt update && sudo apt upgrade -y

sudo apt install -y git build-essential cmake ninja-build exiftool astyle \
python3-pip python3-dev python3-jinja2 python3-numpy python3-yaml \
python3-empy python3-toml python3-jsonschema \

libxml2-utils libxslt1-dev unzip zip curl wget pkg-config
```

Listado 1: Comandos de instalación de dependencias iniciales en Ubuntu 22.04.

4.1.2 Instalación y puesta en marcha de SITL

El firmware del autopiloto simulado, PX4 v1.14 fue clonado desde el repositorio oficial [34] y compilado mediante los scripts desarrollados por la comunidad PX4-Dev. Este repositorio constituye el núcleo del exosistema PX4, que es donde se concentra tanto el código fuente del autopiloto como las herramientas de compilación, simulación y documentación técnica. La instalación se realizó mediante la ejecución del código del listado 2.

```
git clone https://github.com/PX4/PX4-Autopilot.git --recursive

cd PX4-Autopilot

bash ./Tools/setup/ubuntu.sh
```

Listado 2: Clonado y configuración inicial del repositorio oficial.

El comando `git clone –recursive` garantiza la descarga de todos los submódulos que se necesitan para la compilación del firmware y del entorno SITL, el setup de Ubuntu incluye dependencias requeridas para configurar las variables necesarias para compilar el firmware en el sistema operativo y el simulador de físicas por defecto, Gazebo.

Para dar inicio a la simulación basta con especificar el comando `make sitl gazebo_x500` el cual ejecutara el simulador de físicas con un modelo de quadcopter x500.

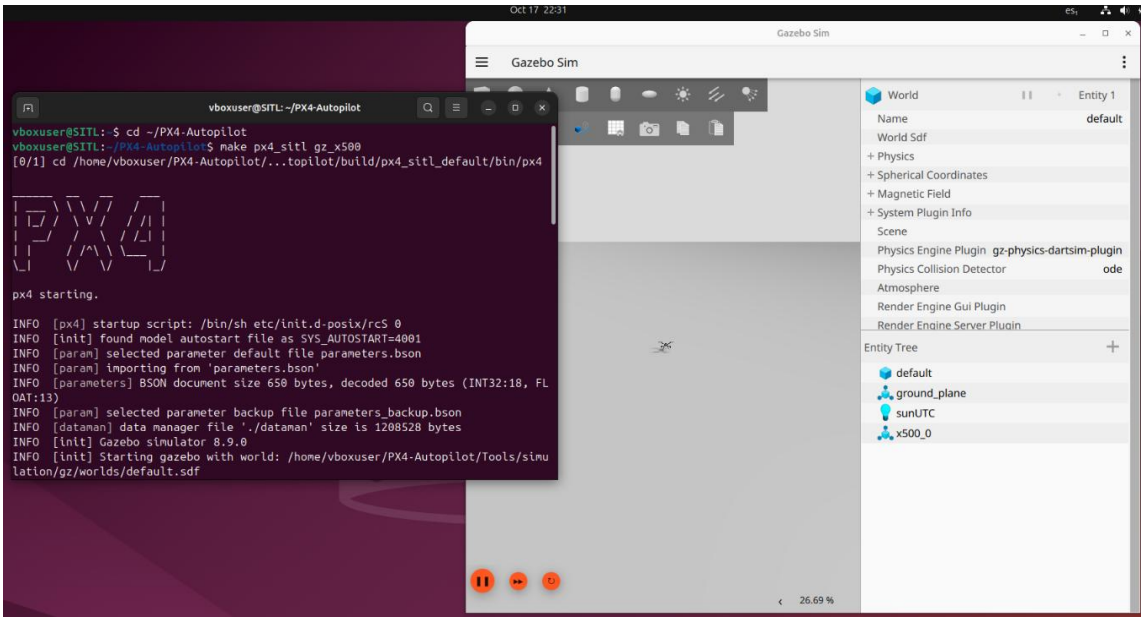


Figura 17: SITL y simulador de físicas ejecutados

La GCS como se mencionó previamente será QGroundControl (v4.3) para el monitoreo y planificación de misiones y MAVSDK-Python (v2.3.0) como biblioteca para la comunicación por MAVLink y ejecución de algoritmos. Para la instalación de ambas en Python 3.9 se ejecuta el código del listado 3.

```
sudo apt install libfuse2
wget https://d176tv9ibo4jno.cloudfront.net/latest/QGroundControl.AppImage
chmod +x QGroundControl.AppImage
./QGroundControl.AppImage
python3 -m venv mavsdk_env
source mavsdk_env/bin/actívate
pip install mavsdk==2.3.0
```

Listado 3: Instalación de QGC y configuración del entorno virtual con MAVSDK-Python.

Durante la configuración de arranque del entorno SITL, se habilitaron dos canales de comunicación MAVLink definidos en el script de inicialización `px4-rc.mavlink`, ubicado en el directorio `ROMFS/px4fmu_common/init.d-posix`. Este archivo es de gran importancia pues controla la creación de instancias MAVLink que se ejecutan al iniciar el firmware PX4, permitiendo establecer simultáneamente distintos puertos para la comunicación externa. Se estableció el puerto UDP 14540 para la conexión MAVSDK-Python y 14550 para la conexión con QGC.

4.1.3 Estructura del entorno de simulación

En la figura Figura 18 se ilustra el esquema general del entorno implementado, donde el autopiloto, código Python y estación de tierra se ejecutan dentro de la misma máquina virtual, comunicándose mediante el protocolo MAVLink sobre conexiones UDP.

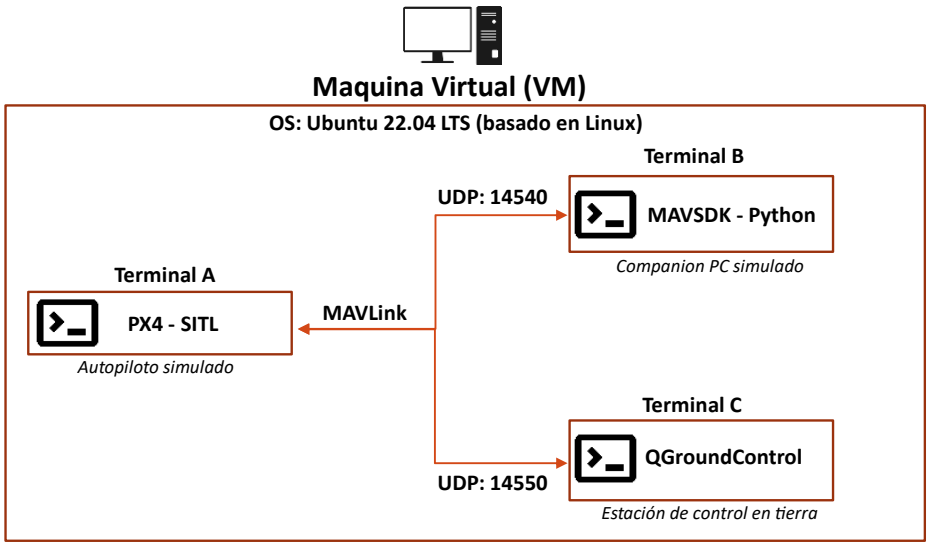


Figura 18: Estructura del entorno de simulación

En este esquema, el terminal A ejecuta el firmware PX4-SITL, que simula el comportamiento del autopiloto. El terminal B ejecuta MAVSDK-Python que cumple el rol de la computadora embebida dentro del entorno virtual, permitiendo le ejecución de algoritmos y el terminal C ejecuta QGC como estación de control en tierra. Los mensajes mediante MAVLink permiten sincronizar en tiempo real la información de estado, misiones y comandos entre los tres terminales.

4.1.4 Verificación del Entorno

Concluida la configuración del entorno de simulación, se realizó una verificación destinada a comprobar la comunicación entre los tres terminales del sistema. El objetivo de esta verificación fue confirmar que los canales MAVLink configurados en los puertos UDP se establecieran de forma estable y bidireccional dentro de la VM.

Durante la ejecución del simulador con el comando `make px4_sitl gazebo`, MAVSDK reconoció la conexión mediante el mensaje "*Connected to PX4",* mientras que QGC detecto de manera automática el vehículo, cambiando su estado de vuelo a "*Ready to fly".* En la consola del autopiloto se registró el mensaje "*partner IP:127.0.0.1",* confirmando el intercambio de datos entre los tres procesos.

Para la validación se empleó una misión simple de verificación, consistente en una trayectoria rectangular definida sobre la pista del aeropuerto Carriel Sur. El propósito fue únicamente validar que SITL reconociera *waypoints* cargados a partir de un código Python vía MAVSDK y ejecutar el modo *AUTO.MISSION* de manera continua.

Figura 19: Ejecución de la misión sobre carriel sur.

Como se puede ver en la Figura 19, la misión actualizo en tiempo real la posición del vehículo en QGC y ejecutó satisfactoriamente el comando `Mission.start_mission()` desde el código Python. Con ello se verifico que el entorno de simulación quedo correctamente operativo y que es posible enviar y ejecutar acciones desde un script en Python directamente sobre PX4. En base a esto, se detalla la implementación del control externo en la siguiente sección.

4.2 Análisis de fidelidad del entorno simulado respecto a vuelos reales

Con el propósito de tener una comparativa respecto a que tanto se asemeja el comportamiento del RPA simulado con el RPA real, se efectuó una comparación cuantitativa entre un vuelo real y una misión simulada ejecutada en PX4-Gazebo, registrado con el mismo plan de vuelo sobre el LTA.

El objetivo fue determinar el nivel de correspondencia entre ambos vuelos en términos de trayectoria, velocidad horizontal y altitud relativa como indicadores representativos del comportamiento dinámico del RPA. Ambos vuelos se basaron en el mismo conjunto de *waypoints* extraídos del autopiloto real al realizar la misión, los cuales se configuraron a distinta altura para obtener una mejor visualización del comportamiento del RPA.

Figura 20: *Waypoints* y trayectoria de la misión de vuelo

Waypoint	Altura [m]
Despegue	15
WP1	30
WP2	40
WP3	20
RTL	20 (SITL) – 30 (vuelo)

Tabla 1: Altura de cada punto de la misión

Para el análisis, en ambos casos se utilizaron los registros de vuelo almacenados en la memoria del autopiloto. Estos archivos, denominados *ulog* en el entorno PX4, contienen los parámetros y variables de vuelo registrados durante la misión, incluyendo datos de posición, velocidad, actitud y estado de los sistemas. Los registros fueron analizados inicialmente mediante PX4 Log Review [35] para una visualización rápida y preliminar del comportamiento del vehículo. Posteriormente se procesaron en MATLAB para una comparación cuantitativa más detallada.

Los datos fueron procesados aplicando una sincronización temporal mediante la variable de velocidad horizontal V_{xy}, la cual fue obtenida a partir de la magnitud horizontal en el marco local NED (sistema de coordenadas North-East-Down), Donde v_x y v_y son los campos de velocidad obtenidos directamente del ulog de PX4, al igual que el resto de las variables, esto permitió alinear eventos característicos, como el arribo a cada *waypoint*.

$$V_{xy} = \sqrt{{v_x}^2 + {v_y}^2} \tag{1}$$

4.2.1 Resultados comparativos

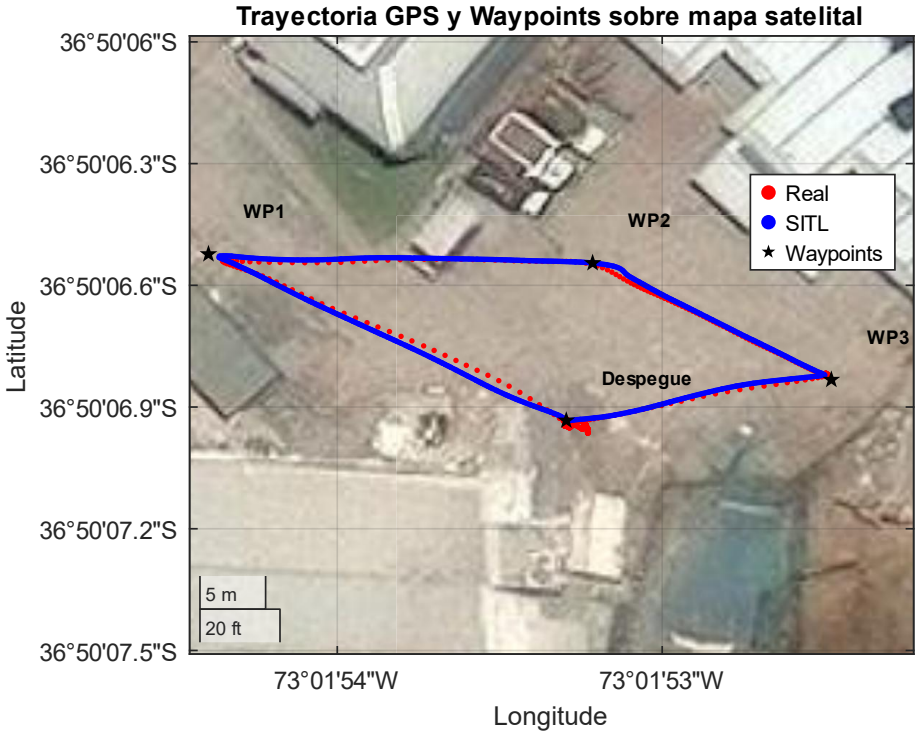


Figura 21: Comparación de trayectorias del vuelo real y simulado

De la Figura 21 se observa la comparación de trayectorias GPS entre el vuelo real y la simulación SITL, ambas proyectadas sobre un mapa satelital. Ambos recorridos siguen el mismo patrón definido por los *waypoints,* reproduciendo correctamente los puntos de despegue, tránsito y cierre de la misión.

La trayectoria simulada presente una coincidencia muy alta con el vuelo real, especialmente en los tramos rectos entre *waypoints,* con desviaciones promedio inferiores a 3 [m]. Las mayores diferencias se concentran en las zonas de viraje, donde el modelo tiene a describir curvas más suaves debido a la ausencia de viento y a la respuesta más idealizada del controlador de actitud en Gazebo. Estas variaciones son esperables, ya que en el entorno real el vehículo experimenta microcorrecciones de rumbo y perturbaciones que no están representadas en el modelo físico del simulador. Además, se puede apreciar que SITL genera telemetría de posición a una tasa significativamente mayor que el GPS del RPA real, lo que se visualiza en la “densidad” de la línea de trayectoria, diferencia que se debe principalmente a la frecuencia de muestreo de la telemetría, la cual en el caso de SITL no depende de un GPS externo, ni tiene ruido.

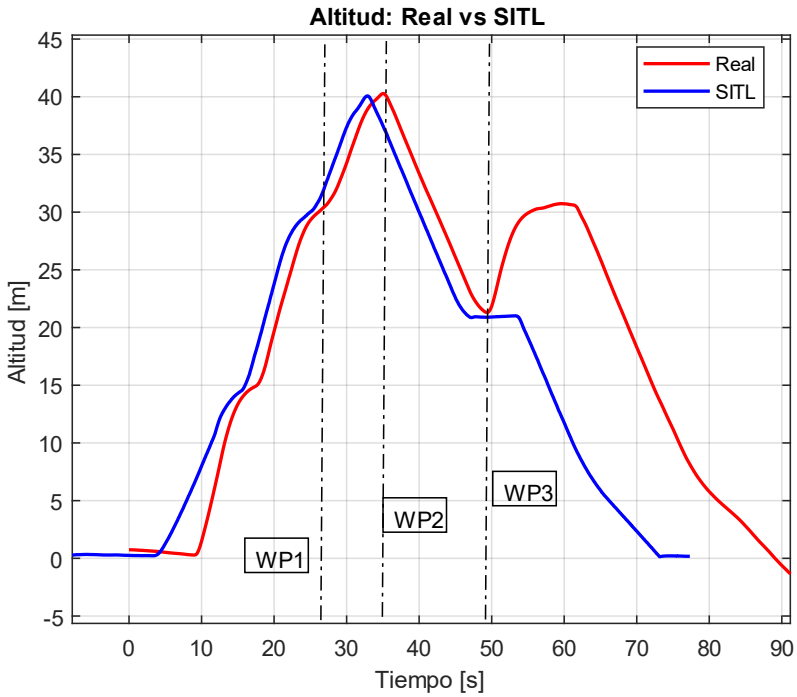


Figura 22: Comparación de altitud del vuelo real y simulado

Como se aprecia en la Tabla 1, los *waypoints* presentaban alturas distintas, por lo que el RPA debía ascender progresivamente hasta el *waypoint* 2 (40 m) y luego descender hacia el *waypoint* 3 (20 m), como se ve en la Figura 22, el perfil de altitud obtenido en el entorno simulado reproduce correctamente esta secuencia, manteniendo similitud con el vuelo real.

La principal diferencia se observa al final de la misión, donde durante RTL el autopiloto real ejecuto un ascenso a una altura de 30 [m], mientras que en el entorno SITL se mantuvo a 20 [m], generando la discrepancia visible en el tramo final. Esta condición no fue establecida manualmente y corresponde a un parámetro predeterminado del firmware, lo que resalta la importancia de verificar la configuración base antes de planificar misiones comparativas.

A pesar de esta diferencia, la coherencia general entre los niveles de altitud confirma que el modelo representa adecuadamente el comportamiento del controlador de altitud de PX4.

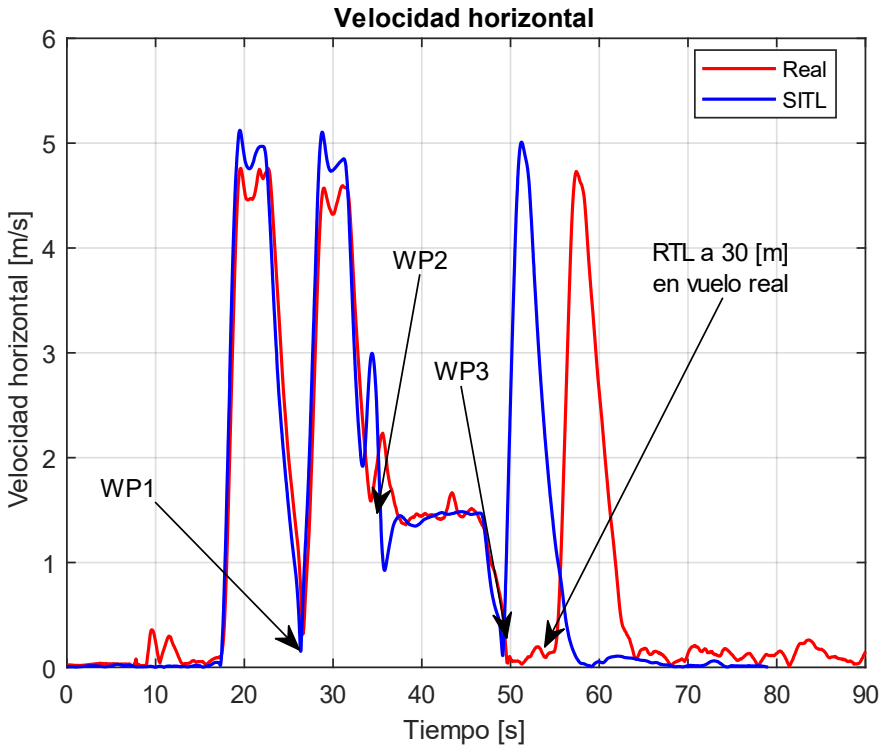


Figura 23: Comparación de velocidades horizontales del vuelo real y simulado

En la Figura 23 se muestran las curvas de velocidad horizontal para ambos vuelos, los cuales presentan una evolución similar, con tres picos principales que corresponden a los desplazamientos entre *waypoints* y zonas de velocidad reducida asociadas a las maniobras de viraje. Sin embargo, se observa una diferencia relevante en el tramo final del vuelo asociada al desfase provocado por la distinta configuración de RTL en ambos casos.

En la simulación, el vehículo ejecuta el comando de retorno inmediatamente después de alcanzar el *waypoint* 3, manteniendo la altitud de ese punto y retornando directamente hacia el origen. Esto genera un pico de velocidad adelantado respecto al vuelo real, donde el autopiloto debió ascender previamente a 30 [m] antes de iniciar el retorno, lo que provoca una breve reducción de la velocidad horizontal hasta prácticamente cero y un nuevo incremento una vez completado el ascenso. El desfase temporal entre ambos perfiles, visible en la figura, se explica justamente por esta diferencia en la fase vertical del RTL.

Fuera de ese tramo, las curvas presentan una correlación estrecha, donde se puede visualizar que la velocidad horizontal del vuelo real presenta un poco más de ruido a lo largo del tramo, mientras que la de la simulación una curva más suave, con velocidades máximas y medias muy próximas (diferencia <8%), lo que indica que al igual que en los otros casos, SITL reproduce con fidelidad la dinámica horizontal del RPA.

La Tabla 2 presenta una síntesis de las métricas comparativas, donde se aprecia la similitud en los resultados para ambos escenarios.

El error cuadrático medio (RMSE) respecto a los waypoints muestra un valor de 0.49 [m] para SITL y 0.62[m] para el vuelo real, lo que evidencia alta precisión en ambos casos.

Métrica	Vuelo real	Vuelo SITL
Altura máxima [m]	40.282	40.071
Velocidad horizontal media [m/s]	0.910	0.842
Distancia total [m]	100.1	104.4
RMSE posición c/r WPS [m]	0.623	0.490

Tabla 2: Métricas comparativas entre el vuelo real y SITL

En conjunto los resultados confirman que el entorno SITL implementado reproduce con fidelidad el comportamiento cinemático del RPA, entregando una base sólida para la siguiente etapa, en la que se incorpora la computadora a bordo como módulo de control externo, donde se probaran los métodos de interrupción propuestos en la sección 4.4.

4.3 Implementación del control externo con MAVSDK-Python

Ya teniendo el funcionamiento general del entorno de simulación verificado, se procedió a implementar un módulo de control externo, destinado a reproducir la lógica que a futuro ejecutara la Raspberry Pi en el sistema físico, permitiendo controlar el vehículo desde un script de Python y validar la transmisión de comandos mediante MAVLink.

Para ello se empleó la biblioteca MAVSDK-Python, la cual hace posible generar instrucciones autónomas desde un proceso externo al firmware. De manera similar a la carga de misión presentada en la sección anterior, MAVSDK actúa como cliente activo dentro del enlace MAVLink, pudiendo enviar comandos y modificar el comportamiento del autopiloto incluso mientras este ejecuta una misión planificada.

Es importante destacar que, aunque PX4 admite recibir comandos externos durante la ejecución de una misión, el firmware prioriza la seguridad del vuelo, por lo que ante cualquier cambio de modo o acción directa enviada por MAVSDK, la misión principal se pausa automáticamente hasta que el operador o un comando externo reanude su ejecución. Esto se puede hacer fácilmente desde QGC pues aparece el mensaje "*Resume Mission",* que permite reanudar de manera manual en caso de interrupción no deseada.

4.3.1 Lógica de programación y tareas asincrónicas

Para la lógica de programación en Python 3.9, se utiliza la biblioteca estándar *asyncio* [36] que permite manejar operaciones concurrentes sin bloquear la recepción de telemetría. Cada función implementada en MAVSDK corresponde a una tarea asincrónica (`async def`) que se ejecuta de manera independiente y puede suspenderse mediante el uso de `await` hasta recibir una respuesta del autopiloto

```
from mavsdk import System
```

```
import asyncio
async def main():
    drone = System()
    await drone.connect(system_address="udp://127.0.0.1:14540")
    async for state in drone.core.connection_state():
        if state.is_connected:
            print("Conectado a PX4")
            break
```

Listado 4: Ejemplo de estructura básica para la función asyncio

El uso de tareas asincrónicas permite mantener la comunicación con PX4 mientras se ejecutan procesos concurrentes, condición indispensable en sistemas embebidos.

El módulo de control externo se elabora mediante funciones que interactúan directamente con los servicios internos de MAVSDK, como *action, misión* y *telemetry,* los cuales permiten enviar comandos o leer información del autopiloto sin manejar directamente los mensajes MAVLink. En la Tabla 3 se adjuntan algunas de las funciones más relevantes a la hora de elaborar scripts.

Función	Descripción	Modulo MAVSDK
`connect()`	Establece el enlace MAVLink con PX4 y valida el estado de conexión.	core
`arm()`	Cambia el estado del autopiloto a *ARMED.*	action
`takeoff()`	Inicia la maniobra de ascenso controlado hasta altitud definida.	action
`upload_mission()`	Envía a PX4 una lista de *waypoints* definidos por coordenadas geográficas y altitud.	mission
`Start_mission`	Inicia misión cargada desde el primer o actual *waypoint.*	mission
`set_current_mission_item()`	Permite modificar el *waypoint* actual de la misión.	mission
`goto_location`	Envia una orden directa de navegación hacia una coordenada especifica, independiente de la misión cargada	action
`Land()`	Detiene cualquier operación activa y ordena el aterrizaje.	action

Tabla 3: Listado de funciones implementadas más relevantes.

Cada una de estas rutinas se acompaña de confirmaciones en consola que informan el estado de ejecución recibido desde PX4. Por ejemplo, tras ejecutar `await drone.action.arm()` ,PX4 responde con el mensaje "*ARMED"* y el estado cambia automáticamente en QGC.

Con respecto a la gestión de telemetría, de manera paralela al envió de comandos, MAVSDK permite suscribirse a flujos de telemetría específicos mediante *listeners* asincrónicos. Estos flujos pueden ser empleados para monitorear las variables de estado del RPA sin interrumpir la ejecución del resto del script, como se puede ver en el ejemplo del listado 5.

```
async for position in drone.telemetry.position():
    print(f"Altitud: {position.relative_altitude_m:.2f} m")
```

Listado 5: Esquema básico de lectura de un listener.

En el contexto del entorno simulado, los *listeners* permitieron registrar de forma continua variables criticas como la altitud y el modo de vuelo, facilitando la visualización en tiempo real del comportamiento del vehículo sin la necesidad de tener que extraer los datos para un análisis.

El módulo de control externo constituye la base funcional sobre la cual se elaborarán scripts que generarán un cambio sobre la misión principal del plan de vuelo.

4.4 Pruebas de interrupción y cambio de misión

Una vez validada la comunicación entre PX4-SITL y el módulo de control externo, se procedió a evaluar la capacidad del sistema para modificar una misión activa en tiempo de ejecución, teniendo como objetivo comprobar que el ecosistema MAVSDK-PX4 permite pausar, alterar o sustituir trayectorias de vuelo sin pérdida de enlace ni errores de sincronización, lo que constituye la base de un sistema con capacidad de decisión autónoma.

En términos generales, una interrupción de misión corresponde a cualquier evento que modifica la secuencia planificada de *waypoints* durante la ejecución del modo *AUTO.MISSION,*

En PX4, este tipo de interrupciones se gestiona mediante los servicios de MAVLink del módulo *mission,* que permiten pausar, reanudar, cambiar el punto actual o incluso reemplazar por completo la misión cargada. El control externo se encarga de enviar las ordenes correspondientes y de verificar la respuesta del autopiloto. A nivel operativo del RPA, una interrupción implica el siguiente flujo de acciones:

1) Detección o activación del evento que gatilla la intervención
2) Pausa de la misión en curso mediante *`pause_mission()`*
3) Ejecución de acción definida por el script
4) Reanudación de la misión original o inicio de una nueva

4.4.1 Misión base de validación y tipos de interrupción implementados

Para las pruebas de interrupción se utilizó como referencia una misión de vuelo sobre el LTA (Figura 24), a partir de coordenadas geográficas extraídas de Google Earth Pro [37]. Estas coordenadas se importaron con una misión de vuelo mediante un script Python. La misión consiste en una trayectoria rectangular con cuatro *waypoints* ubicados alrededor del LTA, a una altitud constante de 15 [m] y velocidad crucero de 5 [m/s]. Su función es servir como escenario donde se aplicaron tres tipos de interrupción propuestos, denominados salto de bloque, desvío directo y plan alternativo o “plan B”.

Figura 24: Misión utilizada para las pruebas de interrupción

Para representar diferentes niveles de intervención sobre una misión activa, se definieron tres métodos de interrupción, mostrados en la Tabla 4, los cuales difieren en su grado de modificación de la planificación y en su aplicabilidad.

Tipo	Descripción	Utilidad
Salto de bloque	Avanza la misión a el siguiente *waypoint* mediante `set_current_mission_item()`	Omitir segmentos o zonas intermedias, saltar de un punto a otro dentro de la misma misión.
Plan alternativo	Sustituye la misión actual por una nueva serie de *waypoints*	Reconfigurar la trayectoria completa del plan de vuelo.

Desvió directo	Detiene la misión y ordena al RPA a una coordenada con *goto_location()*	Responder a estímulos o ejecutar inspecciones puntuales en una zona conocida.

Tabla 4: Tipos de interrupción implementados

Estos métodos fueron seleccionados por representar distintos niveles de autonomía y respuesta que un RPA puede requerir ante condiciones cambiantes durante una misión. El salto de bloque actúa como un “ajuste táctico” del plan, modificando el orden al que se llegan a los *waypoints* sin alterar la misión global. El cambio de misión permite reconfigurar completamente la ruta ante nuevas condiciones, mientras que el desvío directo ejemplifica una reacción inmediata mediante un estímulo puntual. Además, estos tres mecanismos cumplen con viabilidad de ejecución y observación dentro del entorno SITL. En la Figura 25 se puede observar el flujo de decisión para estas misiones de interrupción.

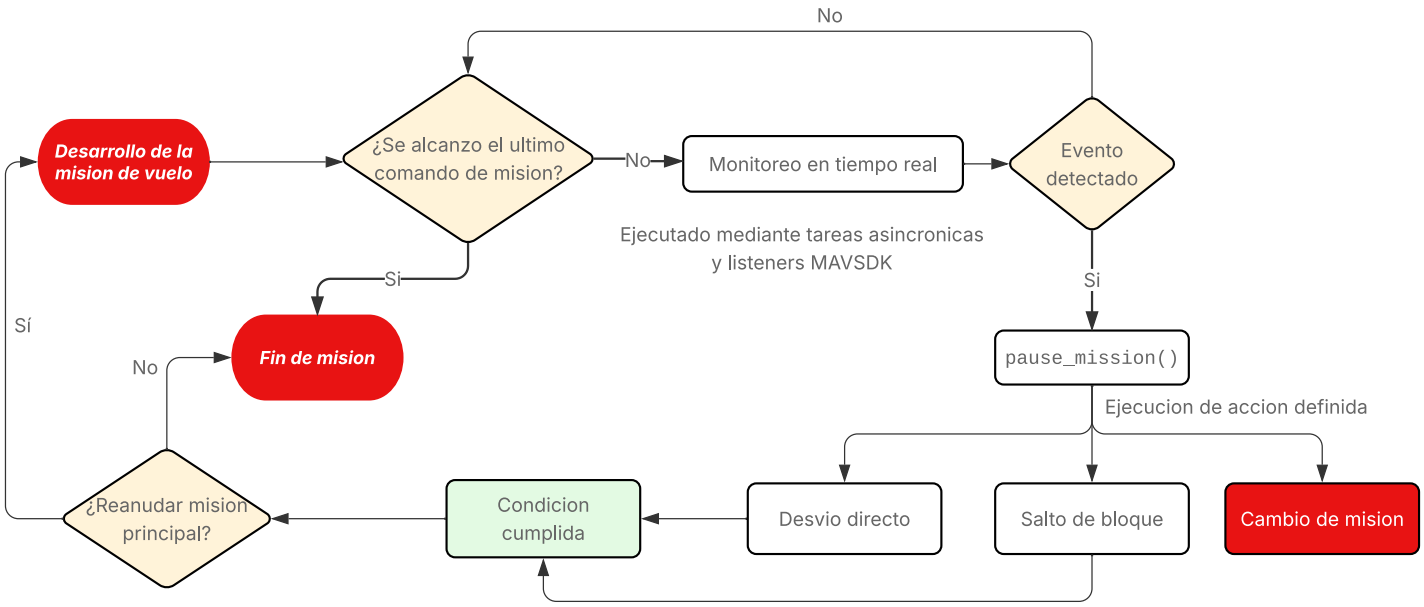


Figura 25: Flujo de decisión para la interrupción y reanudación de misiones.

5 Capítulo 5: Integración de la Raspberry Pi

5.1 Puesta en marcha de la Raspberry Pi

Para la implementación física del sistema de control externo se utilizó la Raspberry Pi 4 Model B como computadora embebida. Este modelo cuenta con un procesador Broadcom BCM2711 Quad Core Cortex-A72 (1.5 GHz), memoria RAM de 4 GB, soporte para Ethernet Gigabit y un módulo de comunicación inalámbrica Wi-Fi 2.4/5 Ghz, lo que permite establecer enlaces directos con el autopiloto y con el computador principal de control. La tarjeta de 64 bits es compatible con los entornos de desarrollo requeridos por MAVSDK y OpenCV [38].

Figura 26: Raspberry Pi 4B utilizada

El sistema operativo instalado fue Raspbberry Pi OS, grabado sobre una tarjeta micro SD de 64 GB mediante la herramienta oficial *Raspberry Pi Imager.* Durante la instalación se habilito el acceso remoto a la Raspberry Pi, permitiendo la configuración y control directamente desde el computador Host, sin la necesidad de conectar periféricos como un teclado o una pantalla. Posteriormente se realizó una actualización completa del sistema y la instalación de los paquetes necesarios para el desarrollo en Python, incluyendo control de versiones con *git* y bibliotecas para visión computacional y comunicación MAVLink.

El entorno de trabajo se configuró sobre Python 3.9.18. Para asegurar un funcionamiento estable y reproducible, se estableció un entorno de desarrollo dedicado dentro del sistema, donde se instalaron exclusivamente las dependencias del proyecto (control por MAVLink, *aasyncio* y visión computacional). Este paso adicional permite mantener versiones específicas de las librerías sin afectar el sistema operativo nativo, lo que garantiza la compatibilidad con los scripts desarrollados para la Raspberry Pi.

La comunicación principal entre la Raspberry Pi y el computador host se estableció mediante un enlace Ethernet directo, configurado con un puente de red en el sistema anfitrión para permitir el acceso simultaneo a Internet y la comunicación MAVLink con la Raspberry Pi. Este procedimiento requirió una configuración especifica dentro de la VM que ejecuta PX4-SITL, donde fue necesario crear un adaptador de red adicional en modo puenteado con la interfaz física Ethernet del computador. Se asignaron direcciones IP estáticas 192.168.137.1 (PC/VM) y 192.168.137.2 (Raspberry Pi). Finalmente, se habilito el uso compartido de internet desde la interfaz Wi-Fi del host hacia la interfaz Ethernet,

permitiendo que la Raspberry Pi obtuviera conectividad externa sin interferir con la red local de simulación.

Una vez establecido el puente, se verifico la comunicación con comandos de diagnóstico (*ping* y *ssh)* y se confirmó la recepción de paquetes MAVLink mediante monitoreo de tráfico UDP, como se puede ver en la Figura 27. Las configuraciones realizadas fueron esenciales para lograr la interacción entre el entorno SITL y la Raspberry Pi, ya que las direcciones y rutas de red debían coincidir entre la VM, el sistema anfitrión y el dispositivo embebido. Con lo establecido se obtuvo comunicación bidireccional de baja latencia, condición fundamental para las pruebas de control autónomo desarrolladas.

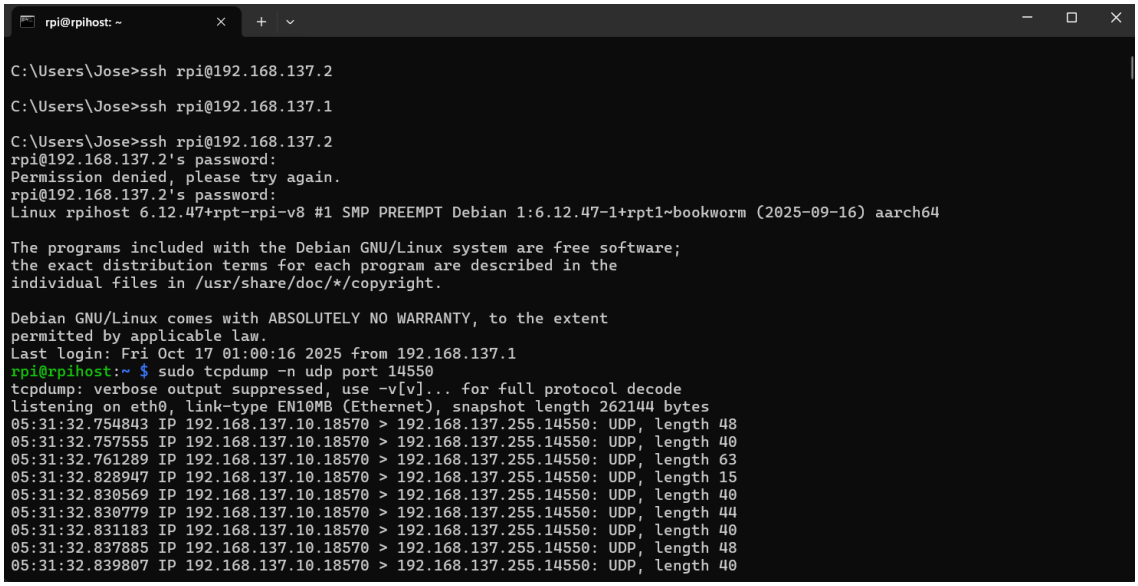


Figura 27: Terminal de la Raspberry Pi recibiendo telemetría en el puerto 14550

Para habilitar la comunicación externa del autopiloto se modificó el archivo de inicialización de PX4, ubicado en el directorio *init.d-posix,* agregando el inicio de un enlace MAVLink dedicado a la dirección IP de la Raspberry Pi sobre el puerto local 14540. De esta forma, el autopiloto transmite telemetría y recibe comandos externos del sistema embebido de forma directa sobre UDP, sin utilizar intermediarios como MAVProxy, dado que PX4 gestiona múltiples enlaces MAVLink de forma nativa permitiendo la operación simultanea con la Raspberry Pi y QGC. Esto se visualiza de mejor manera en el esquema de la Figura 28.

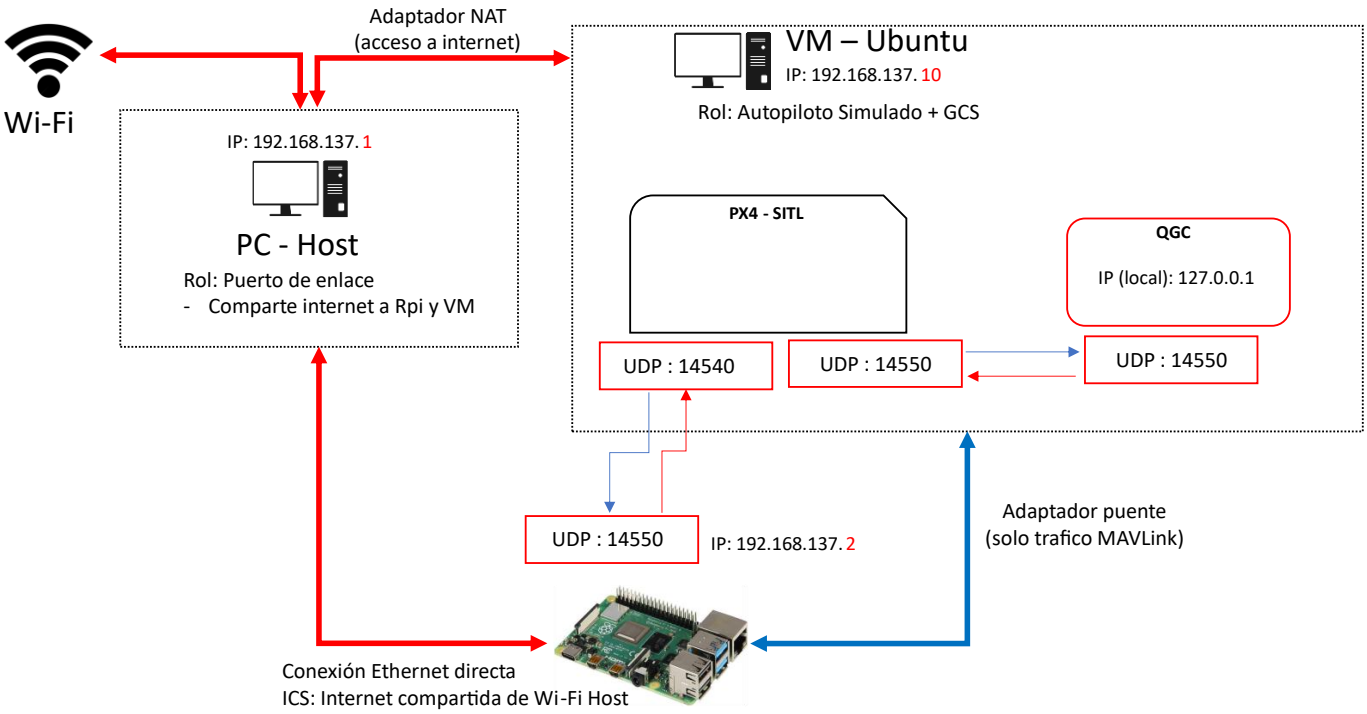


Figura 28: Esquema de comunicaciones PC Host-Raspberry Pi-VM

La conexión se verifico mediante un script básico, visualizado en el listado 6, el cual se encarga de establecer el enlace y confirmar el estado de conexión al autopiloto.

```
from mavsdk import System
import asyncio

async def main():
    drone = System()
    await drone.connect(system_address="udpin://0.0.0.0:14550")
    async for state in drone.core.connection_state():
        if state.is_connected:
            print("Conectado al autopiloto PX4")
            break
asyncio.run(main())
```

Listado 6: Script de verificación de conexión

```
rpi@rpihost:~ $ python conexionprueba.py
Vehicle type changed (new type: 2, old type: 0) (system_impl.cpp:323)
Conectado al autopiloto PX4
```

Figura 29: Visualización de la comunicación con el autopiloto

La ejecución exitosa confirmo la comunicación estable entre la Raspberry Pi y el autopiloto. Con esta configuración lista se puede empezar a desarrollar el método de activación del sistema de decisión.

Es importante señalar en este inciso, que la mayoría de las configuraciones de red, como el uso de puentes virtuales y asignación manual de rutas fueron necesarias únicamente debido a que el entorno se ejecuta dentro de una VM, lo que obliga a coordinar tres dominios (Windows, Ubuntu Linux y Ethernet). Estas configuraciones no son necesarias en un sistema nativo Linux o en la conexión directa a los puertos de telemetría del autopiloto.

5.2 Métodos de activación

Con la comunicación ya establecida, se desarrollaron y evaluaron dos métodos de activación del estímulo, orientados a generar eventos externos que desencadenan acciones autónomas en el RPA. Ambos enfoques comparten el mismo principio operativo: detectar un evento/estimulo, validar su persistencia y ejecutar una lógica de nivel superior mediante comandos MAVLink enviados desde la Raspberry Pi.

El primer método, basado en un interruptor físico, fue concebido como una representación de un estímulo equivalente al generado por un sensor embarcado (por ejemplo, un LiDAR o un sensor de temperatura), destinado a evaluar la respuesta del sistema frente a un evento externo. El segundo, fundamentado en la detección visual de marcadores ArUco, se implementó con una cámara USB genérica, conectada a la Raspberry Pi y constituye el enfoque principal de este proyecto, al incorporar una fuente sensorial real que permite escalar el sistema hacia un comportamiento autónomo basado en percepción.

5.2.1 Activación mediante interruptor físico

El método inicial consistió en la implementación de un botón tipo switch, conectado a los pines GPIO de la Raspberry Pi, actuando como un “estimulo manual” controlado por el usuario.

El switch se conectó entre el pin GPIO 17 y la línea de 3.3 V, configurado con una resistencia *pull-down* interna. El cambio de estado del interruptor (de abierto a cerrado, o viceversa) se interpreta como un evento binario que genera una interrupción del sistema, evitando la necesidad de una lectura continua del pin. Una rutina en Python captura dicha transición y ejecuta la función correspondiente mediante la librería MAVSDK, enviando un comando al autopiloto PX4. El pinout de la conexión se visualiza en el Anexo 3.

5.2.2 Activación mediante detección visual

El segundo método corresponde a la integración de detección visual mediante una cámara, basado en marcadores ArUco, la cual, para el caso de las pruebas en SITL trabaja en conjunto con el switch, pero será el enfoque adoptado sobre las pruebas en el RPA real.

Los marcadores ArUco [39] son patrones visuales cuadrados formados por una matriz binaria interna que codifica un identificador único. Su detección se realiza mediante algoritmos de visión artificial basados en la biblioteca OpenCV, los cuales reconocen la

forma, orientación y tamaño del marcador en la imagen. La idea es asociar una acción especifica dentro del sistema, basada en la detección del ArUco.

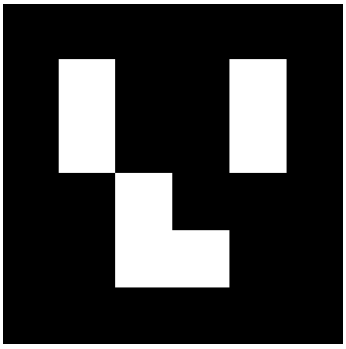


Figura 30: ArUco con ID = 3

El proceso de detección se implementó con una cámara conectada al puerto CSI de la Raspberry Pi, operando a 30 fotogramas por segundo. Cada cuadro se procesa en tiempo real mediante el módulo `cv2.aruco` el cual extrae la información del marcador. Una vez detectado el sistema solo lo considera valido si el marcador se mantiene visible durante un número mínimo de cuadros, lo que evita activaciones erróneas por ruido.

Una vez confirmada la detección, la Raspberry Pi determina la acción en base al comando asociado al identificador del marcador y envía la orden MAVLink al autopiloto. El uso de estos marcadores ofrece una solución ligera y energéticamente eficiente, adecuada para implementar en este caso y respaldado por la literatura.

La conexión final de la VM, con la Raspberry Pi, el switch y la cámara se visualizan en la Figura 31.

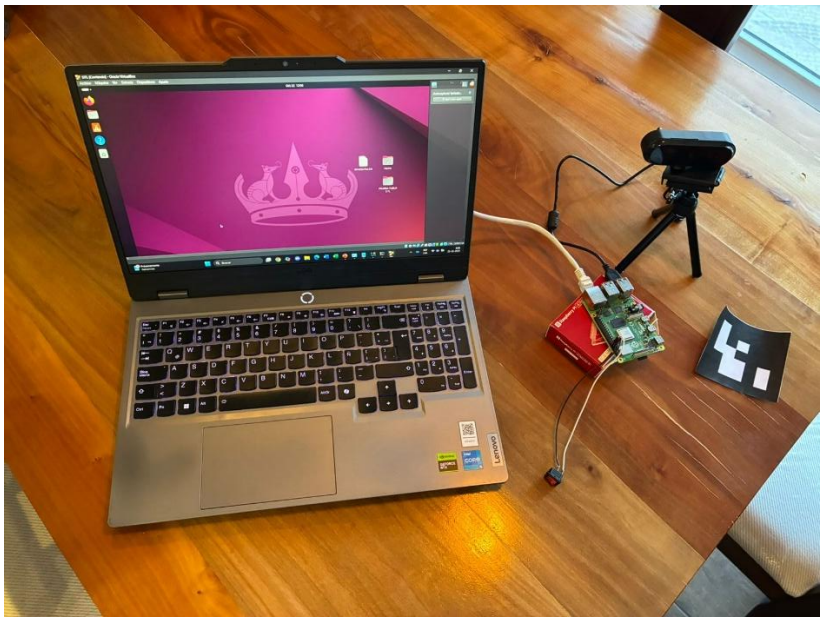


Figura 31: "banco de pruebas SITL-Raspberry Pi"

5.3 Validación RPI-SITL mediante pruebas de interrupción

Utilizando las pruebas de interrupción descritas en el Capítulo 4 y empleando la Raspberry Pi como módulo de decisión, se buscó comprobar la capacidad para detectar estímulos externos y ejecutar, sin intervención del operador, los comandos MAVLink necesarios para modificar una misión activa en tiempo real.

Se empleo la misma misión sobre el LTA vista en la Figura 24 y los tres métodos de interrupción ya descritos. La diferencia principal radica en la fuente activación, ahora no de un terminal MAVSDK si no de un switch físico o el sistema visual ArUco.

Durante cada ensayo, la Raspberrry Pi ejecuto un script MAVSDK que permanecía en escucha constante de eventos, una vez detectado, se ejecutaba la acción correspondiente y se registraban los tiempos de detección, ejecución y confirmación del comando.

5.3.1 Resultados de las pruebas de interrupción

Los resultados confirman que la Raspberry Pi puede realizar las mismas acciones de control que se podían realizar desde la terminal MAVSDK dentro de la VM, pero ahora de manera autónoma y en tiempo real.

En el Caso 1, sin estímulos, la misión se completó sin interrupciones sirviendo como referencia temporal del plan de vuelo.

En el Caso 2, la activación del switch ejecuto un cambio de misión hacia “plan B” con una latencia inicial de aproximadamente 13.8 segundos, asociada al reinicio momentáneo del enlace MAVLink durante la primera pausa de misión, un error recurrente al intentar sobrescribir una acción sobre el autopiloto, que se soluciona fácilmente al reenviar la orden dentro del script.

En el Caso 3, la detección del marcador ArUco ID 22 desencadeno un desvió directo manteniendo latencias promedio entre 0.2 y 0.5 segundos, tras la estabilización del enlace.

Finalmente, el Caso 4 combino ambos estímulos dentro de una misma misión, alternando entre interrupciones visuales y por switch sin perdidas de enlace por sincronización. Esto prueba que en un vuelo más de un sensor desde la Raspberry Pi podría enviar un comando al autopiloto. Los registros de comunicación generados por la Raspberry Pi se encuentran en el anexo 2, estos incluyen un *timestamp* de detección, tipo de evento y latencia medida.

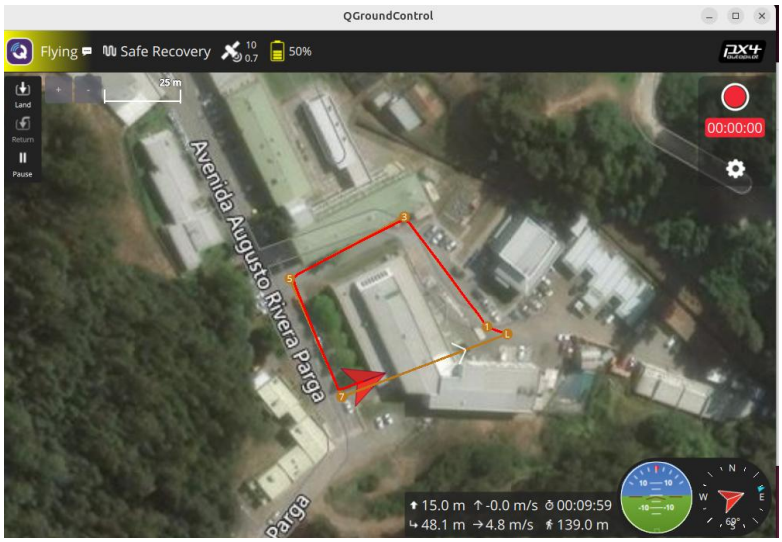


Figura 32: Caso 1, ejecución sin estímulos

Se puede visualizar que la misión sigue la trayectoria de los *waypoints* sin interrupción, en un tiempo aproximado de 73 segundos.

Figura 33: Caso 2, carga de misión "Plan B" al activar switch

En el Caso 2, la misión alternativa correspondió a una serie de *waypoints* ubicados a mayor altitud que la trayectoria original, permitiendo distinguir visualmente el cambio de plan en QGC. Para que el operador pueda visualizar esta nueva misión en la interfaz, es necesario descargar nuevamente el plan de vuelo desde el autopiloto en la sección de planificación. Este comportamiento ocurre debido a que PX4 ejecuta la navegación directa internamente sin modificar la misión cargada, por lo que la visualización en la GCS permanece estática hasta que se actualice manualmente.

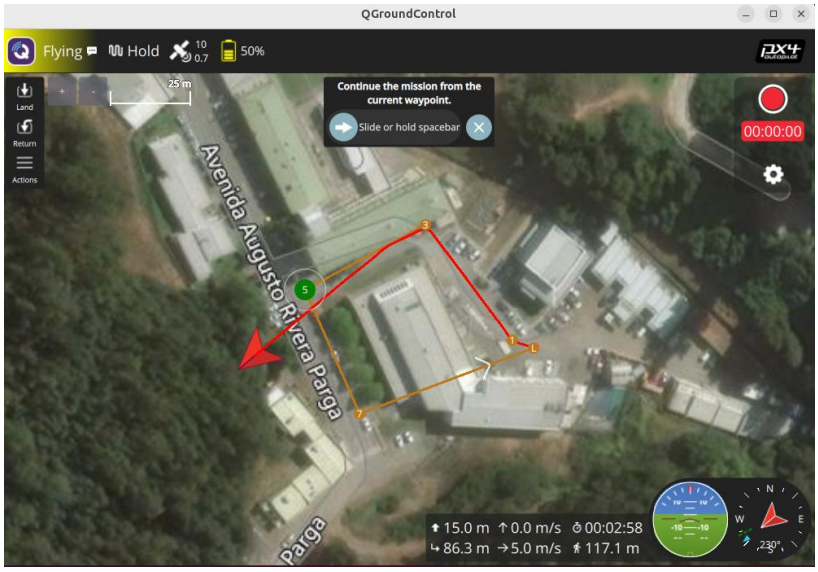


Figura 34: Caso 3, desvió directo por detección de ArUco.

Lo mismo que en el caso anterior ocurre para el caso 3, la visualización del nuevo *waypoint* a seguir no se actualiza en la interfaz de forma automática. En este caso el punto de destino se encontraba próximo al cerro colindante al LTA, lo que permitió verificar que el desvió directo fue interpretado por el autopiloto, pese a no reflejarse en la interfaz.

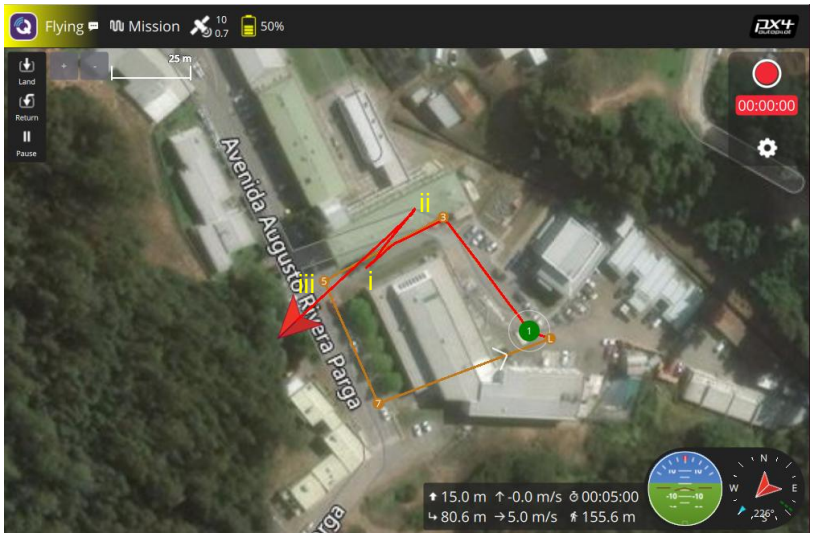


Figura 35: Caso 4, combinación de switch con ArUco.

En este caso, el sistema encadeno tres interrupciones: primero la detección de ArUco lo cual activo el desvió (i), luego el switch sobrescribió la misión con el plan B (ii) y finalmente una nueva detección visual volvió a ejecutar el desvió (iii). La trayectoria resultante se muestra con claridad en la Figura 35. Debido a que el marcador permanece visible durante varios fotogramas, el sistema puede detectar el mismo estimulo repetidamente, por ello es necesario establecer una validación por persistencia para evitar saturar el autopiloto con órdenes, lo cual se puede visualizar en el Anexo 2.4.

Un aspecto relevante observado durante la última prueba es la prioridad en la ejecución de estímulos simultáneos, la cual depende directamente del orden en que los comandos MAVLink son recibidos por PX4, ya que el autopiloto ejecuta únicamente el último mensaje valido recibido. En la configuración actual, el script evalúa primero la detección visual y luego el estado del interruptor, por lo que, si ambos eventos se activan casi al mismo tiempo, el comando del switch se envía al final del ciclo y sobrescribe la acción previa. Si se requiere establecer una prioridad fija, como privilegiar siempre la detección visual, basta con anidar las condiciones en el código (estructurar la lógica de tal manera que un estímulo excluya a otro).

5.4 Pruebas de actitud controladas por la Raspberry pi

Con el propósito de caracterizar el comportamiento dinámico del RPA, se realizaron pruebas de control de actitud en el entorno PX4-SITL, donde la Raspberry Pi envió *setpoints* de actitud (roll, pitch, yaw) a PX4 en modo *OFFBOARD*, los cuales fueron perseguidos por los controladores internos del autopiloto en respuesta a la detección visual de marcadores ArUco.

Esta etapa tuvo como objetivo registrar el comportamiento angular del modelo simulado, de modo que sirva posteriormente como punto de comparación con las pruebas en el banco de ensayos (como en el vuelo de prueba de la sección 4.4.1, se compararan en un mismo gráfico).

El control se implementó mediante un algoritmo Python con la librería MAVSDK, donde se asocia un marcador ArUco a un eje de rotación y a una amplitud determinada: ±10° para Roll / Pitch y ±20° para Yaw. De la misma manera que en las pruebas de interrupción, la Raspberry Pi equipada con una cámara genérica detecta en tiempo real los marcadores y activa la maniobra correspondiente cuando el estímulo se mantiene visible durante un número mínimo de cuadros consecutivos.

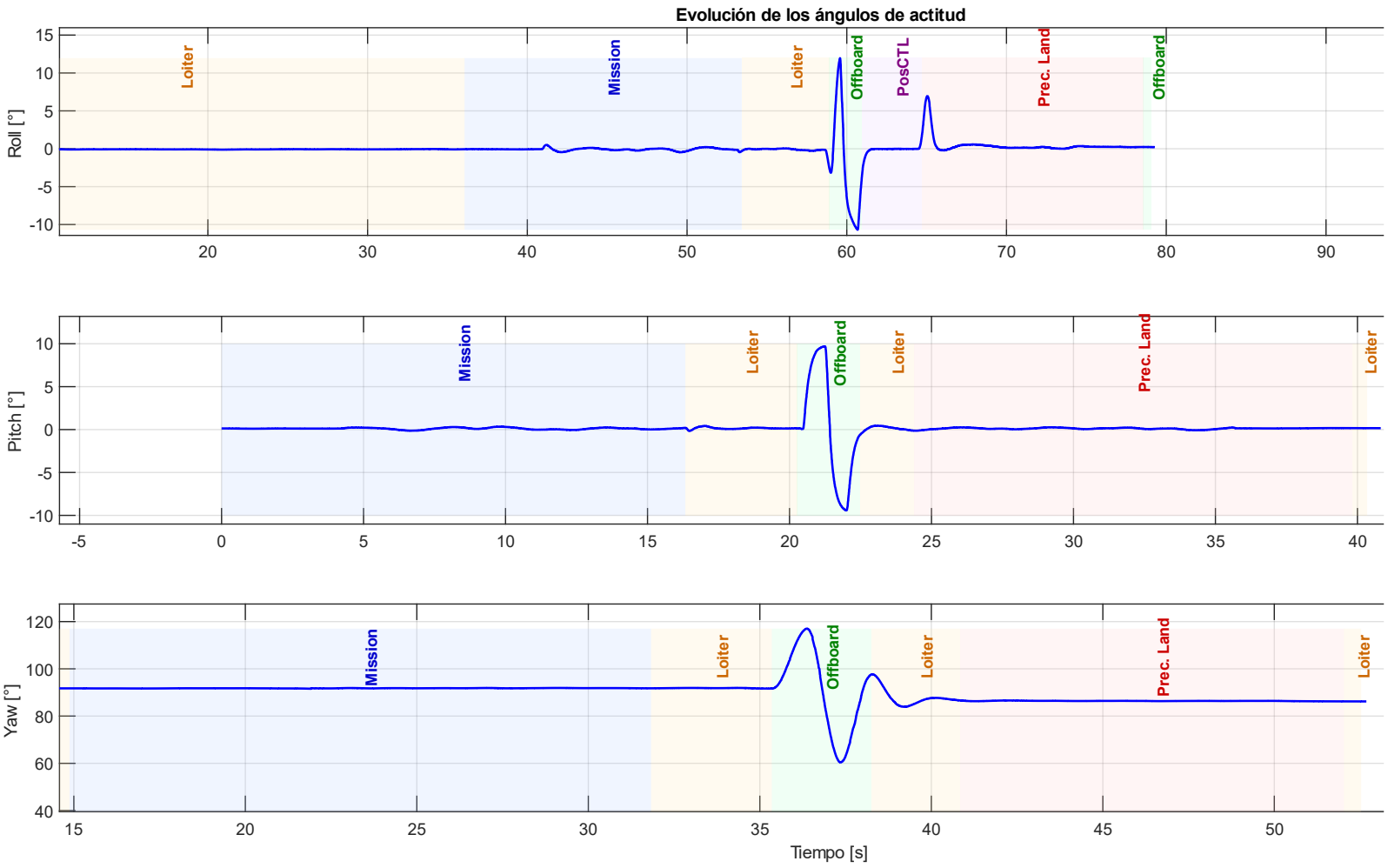


Figura 36: Ángulos de actitud

Los resultados obtenidos de la prueba de actitud muestran que el sistema logró ejecutar correctamente los comandos de control enviados desde la Raspberry Pi, generando variaciones controladas en los tres ejes. El pitch presento oscilaciones de ±9.5°, mientras que el roll alcanzo ±11°, lo que evidencia una respuesta estable frente a los comandos de actitud. En el caso del yaw, el vehículo mantuvo una referencia inicial cercana a 91.8° y registro desviaciones entre 116.9° y 61.1°, confirmando que la rotación se efectuó en torno a su orientación global.

En conjunto, estos resultados validan la correcta interpretación de las ordenes MAVLink por parte del autopiloto en modo *OFFBOARD,* demostrando que el sistema de decisión puede inducir maniobras controladas y reproducibles a partir de un estímulo visual.

6 Capítulo 6: Implementación en hardware físico

6.1 Puesta en marcha del sistema embebido

La puesta en marcha del sistema embebido requirió la integración estructural, eléctrica y funcional entre la Raspberry Pi y el autopiloto, en el RPA físico. Para poder incluir ambos componentes dentro de un mismo *frame* fue necesario imprimir diversas piezas ya diseñadas, con ajustes mínimos, que permiten dar estabilidad y accesibilidad a los puertos.

La Tabla 5 resume las piezas fabricadas y su función principal dentro del sistema.

Pieza Impresa	Función
Cama del autopiloto	Alojamiento rígido del autopiloto.
Cama de la Raspberry Pi	Soporte estable para la computadora embebida, con espacio para su tarjeta SD y aislamiento de vibraciones.
Paredes laterales	Base para contención de la tapa superior y protección a impacto.
Base del modulo	Plataforma de fijación general.
Tapa superior	Base para la ubicación del GPS.

Tabla 5: Piezas impresas para la puesta en marcha

La mayoría de las uniones realizadas fueron utilizando pernos M3 en los aprietes, los cuales son compatibles con los insertos estándar del marco del RPA.

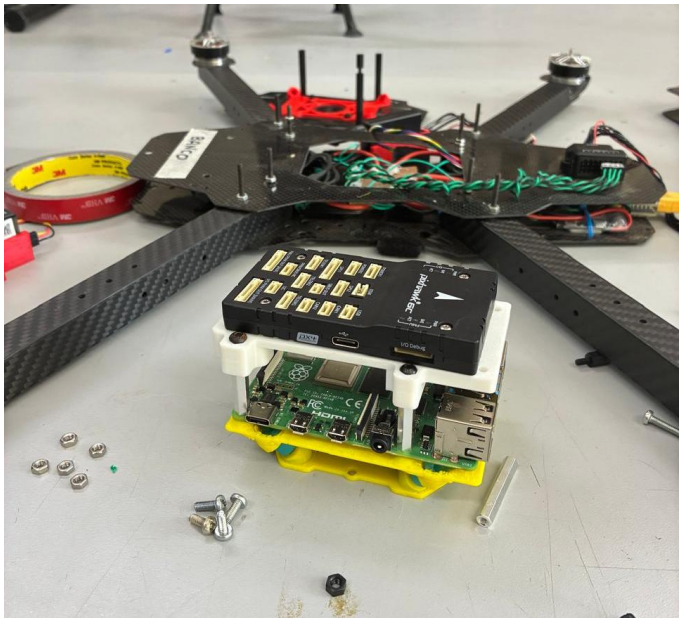


Figura 37: Autopiloto y Raspberry Pi en sus respectivas camas.

6.1.1 Integración de la Alimentación

Para la alimentación de la Raspberry Pi, inicialmente se obtuvo a partir de una toma directa de corriente en la pared, con lo que se pudo configurar el sistema operativo y realizar las pruebas en SITL. Sin embargo, para poder realizar pruebas en el banco, fue necesario diseñar una extensión a la alimentación del autopiloto, de manera que el voltaje suministrado a la Raspberry Pi provenga de la batería LiPo.

El diseño consiste en la implementación de un módulo reductor de voltaje tipo Step-Down DC-DC LM2596 [40]. Este módulo fue necesario implementarlo ya que la alimentación del sistema proviene directamente de la tensión cruda de la placa de distribución de poder (PDB), la cual reparte el voltaje completo de las baterías, variando entre 16.8 a 14 [V] durante la operación del RPA. Dado que este voltaje excede los límites de la Raspberry Pi por un gran margen, se incorporó el reductor, el cual permite convertir la tensión *VBAT* en una salida regulada y estable de 5 [V]. El esquema de la Figura 38 ilustra como se distribuye la energía.

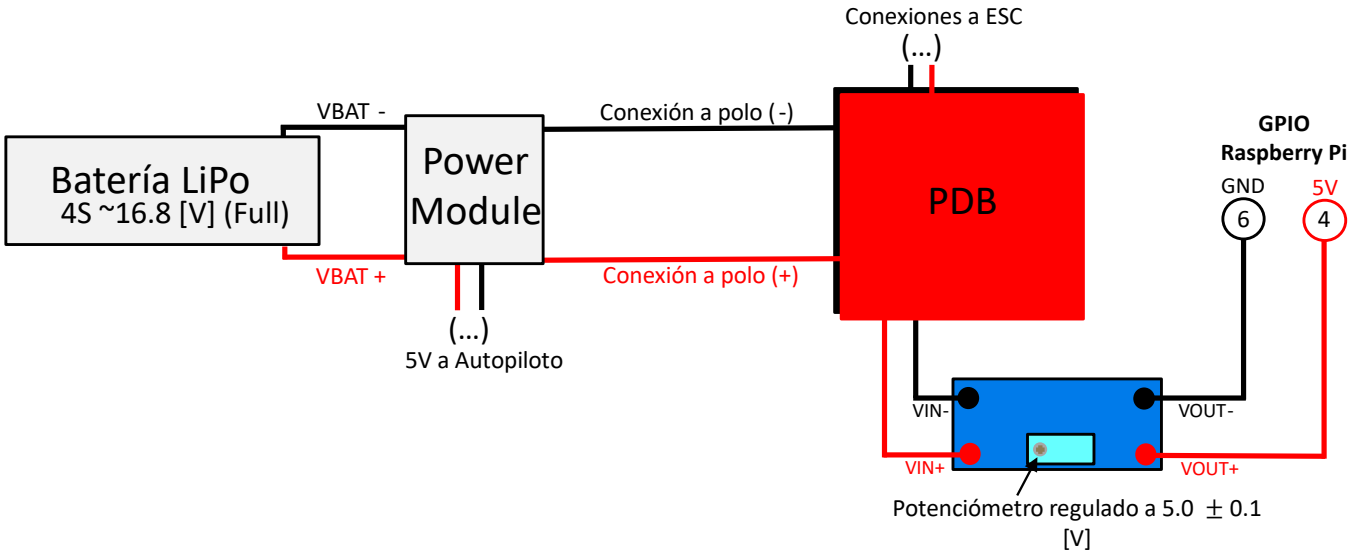


Figura 38: Distribución de la alimentación del RPA

En la figura también se puede visualizar el Power module que da la energía al autopiloto y el potenciómetro del regulador, el cual se calibra mediante el giro de un pequeño perno, por seguridad se fue probando con un multitester la salida de energía para verificar que se encontrara dentro de los rangos de voltaje admisibles por la Raspberry. El error que presentaba el reductor es bastante despreciable y en general para todos los rangos operativos de la batería, es más que suficiente para proporcionar una salida estable, lo cual se expresa en el manual del fabricante para una salida de 5 [V], como se puede ver en la Figura 39.

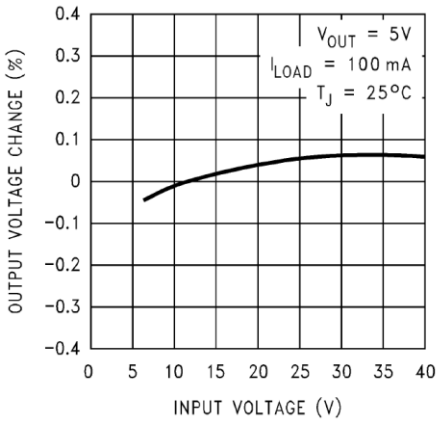


Figura 39: % Del error de salida en el voltaje del reductor, para una salida de 5 [V]
[40]

6.1.2 Integración de comunicaciones (Conexiones + QGC)

El paso más importante de la puesta en marcha fue la integración de comunicaciones entre la Raspberry Pi y el autopiloto. La configuración requirió tanto la conexión física entre los dispositivos como la configuración de parámetros internos en PX4 para que se lograra establecer comunicación bidireccional.

La conexión física se estableció utilizando el puerto TELEM2 del autopiloto, esta se puede visualizar en la Figura 40. Este puerto está diseñado para proporcionar comunicación serial bidireccional y se configuró para operar a 57600 baudios, valor adecuado para los enlaces MAVLink. En la Raspberry Pi, la comunicación fue enlazada mediante la interfaz */dev/serial0*, habilitando la UART principal del dispositivo. Para hacer esto se tuvo que activar manualmente el puerto serial desde la configuración del sistema operativo de la Raspberry Pi y deshabilitar el uso de la consola por defecto sobre UART.

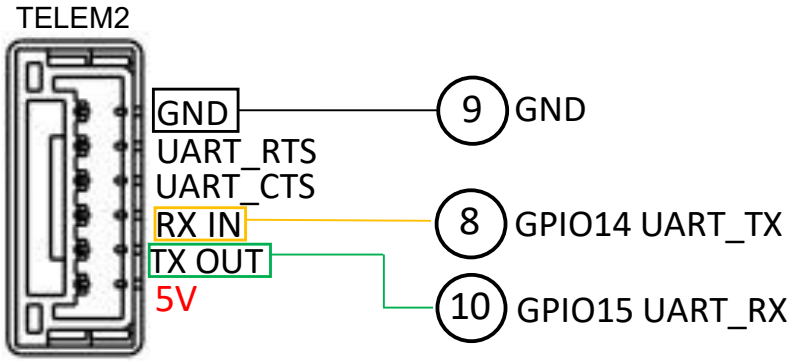


Figura 40: Conexión a puerto TELEM2 del autopiloto.

Para el autopiloto, se debieron configurar diversos parámetros desde QGC que permitieran reconocer a el puerto TELEM2 como un enlace MAVLink activo, ya que viene por defecto desactivado. Esos parámetros se incluyen en la Tabla 6.

Parámetro	Descripción	Valor
MAV_2_CONFIG	Asocia instancia MAVLink 2 a una interfaz física del autopiloto	TELEM2
SER_TEL2_BAUD	Velocidad de transmisión del puerto TELEM2	57600
MAV_1_MODE	Modo del enlace MAVLink	0 (Modo estándar)
MAV_0_FORWARD	Reenvió de mensajes MAVLink (Permite coexistencia de Raspberry Pi + QGC)	1

Tabla 6: Parámetros configurados para el puerto TELEM2 desde QGC.

Una vez configurado los parámetros, para comprobar la comunicación se decidió validar la comunicación con el autopiloto directamente desde la terminal de la Raspberry Pi, utilizando MAVProxy [41], una herramienta de línea de comandos que permite inspeccionar y decodificar en tiempo real los mensajes transmitidos por el autopiloto.

```
mavproxy.py --master=/dev/serial0 --baudrate 57600
```

Listado 7: Comando de ejecución MAVproxy

Al ejecutar el código, se observó inmediatamente el *heartbeat* del autopiloto, el modo de vuelo, mensajes de estado y telemétrica continua, confirmando que los pines TX, RX estaban bien conectados, la UART de la Raspberry Pi estaba funcionando, la configuración de baudios era correcta y que el autopiloto respondía a comandos MAVLink sin errores.

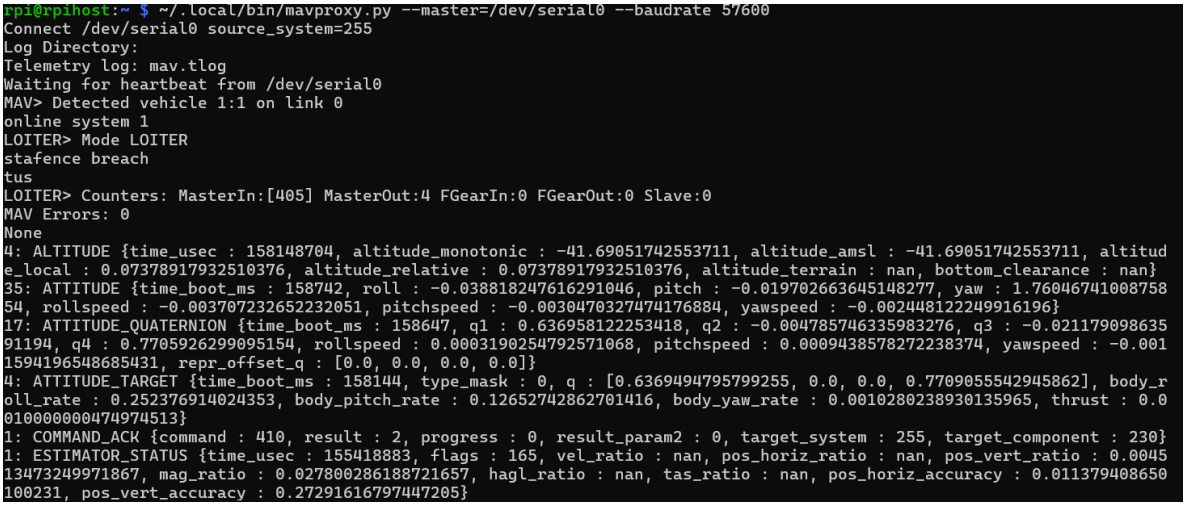


Figura 41: Ejecución de comando "Status" mediante MAVproxy

6.1.3 Integración del Radio Control

Para el control manual del RPA y como sistema de seguridad durante las pruebas en el banco, se integró un radio control Radiomaster TX16S, el cual se encontraba previamente

configurado por el usuario anterior. La emisora operaba correctamente con el receptor de telemetría instalado en el vehículo, por lo que no fue necesario modificar la asignación principal de canales ni la curva de sticks, dentro de la radio, pero si se debió realizar la calibración en QGC.

Figura 42:Radio control TX16S

El único ajuste relevante consistió en la correcta asignación del switch a los modos de vuelo: *stabilized, manual y altitude* utilizados en las pruebas de banco, lo que permite cambiar rápidamente de vuelo en caso de un comportamiento no deseado durante las pruebas.

En los ajustes de parámetros, se configuró *COM_RC_IN_MODE,* a "*RC and joystick with fallback*" lo que permite la coexistencia entre la operación del radio control por un operador humano y el control autónomo de una computadora embebida. El parámetro asegura que en caso de pérdida del enlace o falla de la sesión *OFFBOARD,* el autopiloto retorne automáticamente el control al radiocontrol, lo que en un vuelo real resultaría de vital importancia.

6.1.4 Integración funcional de la cámara en el sistema

La integración funcional se enfocó en lograr habilitar por completo el flujo de detección visual sobre la Raspberry Pi. De manera adicional a la cámara USB utilizada en las primeras pruebas en la VM, se empleó una cámara oficial de Raspberry Pi, conectada mediante el Bus CSI, ofreciendo menor latencia y mejor compatibilidad en vuelo. La cámara utilizada es una Raspberry Pi HQ, con montura intercambiable y focal variable.

Figura 43: Cámara Raspberry Pi HQ (IMX477)

A diferencia de una USB convencional, que se reconoce automáticamente como un dispositivo de video estándar, la cámara HQ utiliza el bus CSI que requiere el uso del sistema *libcamera* para acceder al sensor. Esto implica que los datos de imagen no se entregan como un flujo de video tradicional, por lo que fue necesario adaptar el código e incorporar un “hilo productor de imágenes” que transforma cada captura en un formato compatible con la librería de OpenCV.

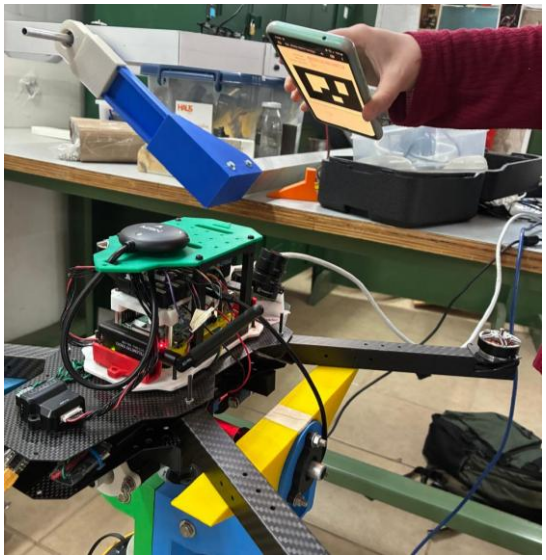


Figura 44: Integración funcional de la cámara en el RPA

Con la cámara ya instalada se realizó una prueba de validación visual, la que consiste en detectar los marcadores ArUco en una pantalla y enviar una respuesta a la terminal de la Raspberry Pi que diga “detectado”, cada vez que el marcador entraba al campo de visión. Esto permite saber si la captura de imágenes es estable y como se debe ajustar el enfoque de la cámara a una distancia tal que reconozca el patrón de los marcadores.

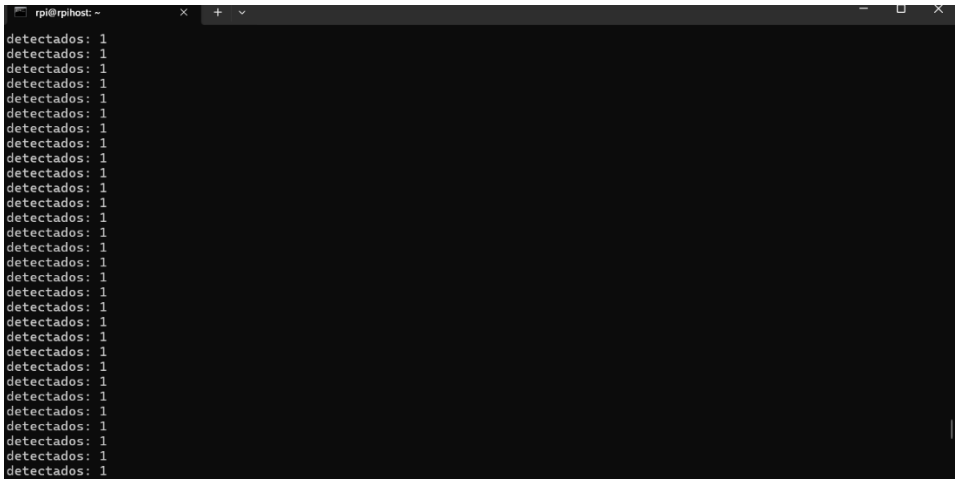


Figura 45: Detección confirmada por terminal de la Raspberry Pi

Con esta integración validada, la cámara HQ quedo operativa como sensor principal del sistema autónomo, habilitando su uso en pruebas del banco de ensayos, sin embargo, debido a la pequeña longitud del cable plano que conecta la cámara con el puerto CSI de la Raspberry, se decidió usar la cámara USB para más comodidad a la hora de hacer las pruebas.

6.2 Pruebas en banco de ensayos

Las pruebas en el banco de ensayo fueron diseñadas de tal manera que se pueda tener una idea del comportamiento del RPA al detectar un estímulo, de manera segura y reproducible, sin tener que arriesgar equipo ni usar baterías. Para la ejecución del código, este se programó como un “servicio” en la Raspberry Pi, de manera que, similar a un Arduino, la ejecución es pasiva en segundo plano mientras la Raspberry se encuentra encendida.

6.2.1 Calibración inicial y configuración para el entorno sin GPS

Para preparar el RPA para las pruebas en el banco fue necesario calibrar todos los sensores del vehículo en QGC, donde utilizando su asistente interactivo, se realizaron las calibraciones del compás, giroscopio, acelerómetro y nivelación del horizonte.

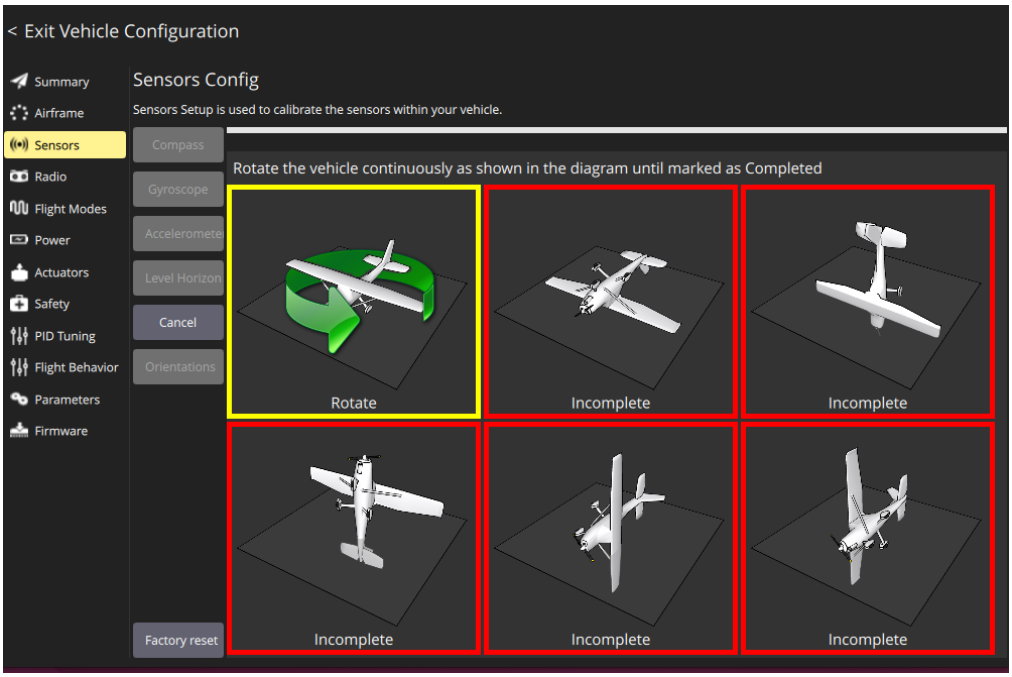


Figura 46: Calibración de los sensores.

La radio también debió ser calibrada y ajustada con los canales para los modos de vuelo descritos en 6.1.3, también el rango de operación de los actuadores y la verificación de su funcionamiento. Se omitió la calibración de la fuente de poder pues es una toma de corriente directa a la pared.

Para poder operar en un entorno sin GPS (interior del LTA), inicialmente se tomó la decisión de no conectarlo al puerto del autopiloto, pero resulto ser más problemático pues la falta del magnetómetro incorporado en el módulo provoco muchos errores a la hora de intentar armar el vehículo. Debido a esto fue necesario configurar PX4 para permitir el armado sin una señal satelital valida, para ello se debieron ajustar los siguientes parámetros para sobrepasar los chequeos de seguridad.

Parámetro	Descripción	Valor
COM_ARM_WO_GPS	Permite armar aun cuando el GPS no entrega posición valida o no adquiere fix.	1
NAV_NO_GPS	Habilita modos de vuelo no dependientes de navegación GPS	1
COM_POSCTL_NAVL	Evita que PX4 intente entrar en control de posición (Position Mode), el cual requiere señal GPS estable.	0

Tabla 7: Parámetros para permitir operación sin GPS.

6.2.2 Pruebas de actitud

Para validar la interacción entre la Raspberry Pi y el autopiloto en condiciones reales, se realizó una prueba controlada en el banco de ensayos del LTA. Este montaje permite mantener libertad rotacional en los ejes, permitiendo evaluar la respuesta de roll, pitch y yaw, lo que posibilita evaluar la respuesta del controlador.

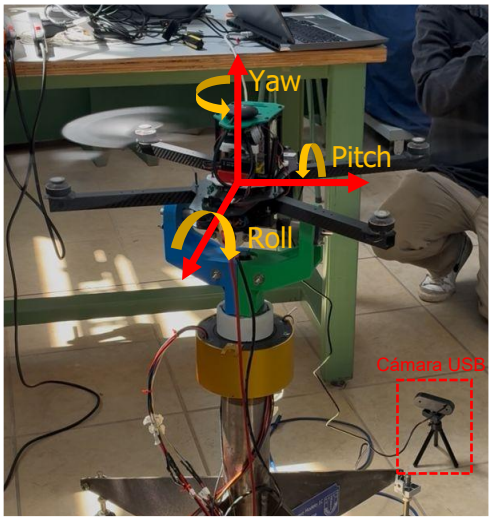


Figura 47: Prueba en banco de ensayos

La idea de esta prueba fue verificar que la Raspberry Pi pudiese generar *setpoints* de actitud y enviarlos correctamente hacia PX4 en modo *OFFBOARD*, comprobando como el autopiloto interpreta estas órdenes y ajusta sus lazos internos para alcanzar el ángulo objetivo. El vehículo se mantuvo en *Stabilized Mode* antes de ejecutar las maniobras.

Por seguridad se mantuvo en todo momento el radio control como mecanismo de respaldo, permitiendo recuperar el control manual simplemente con accionar el switch de cambio de modo de vuelo.

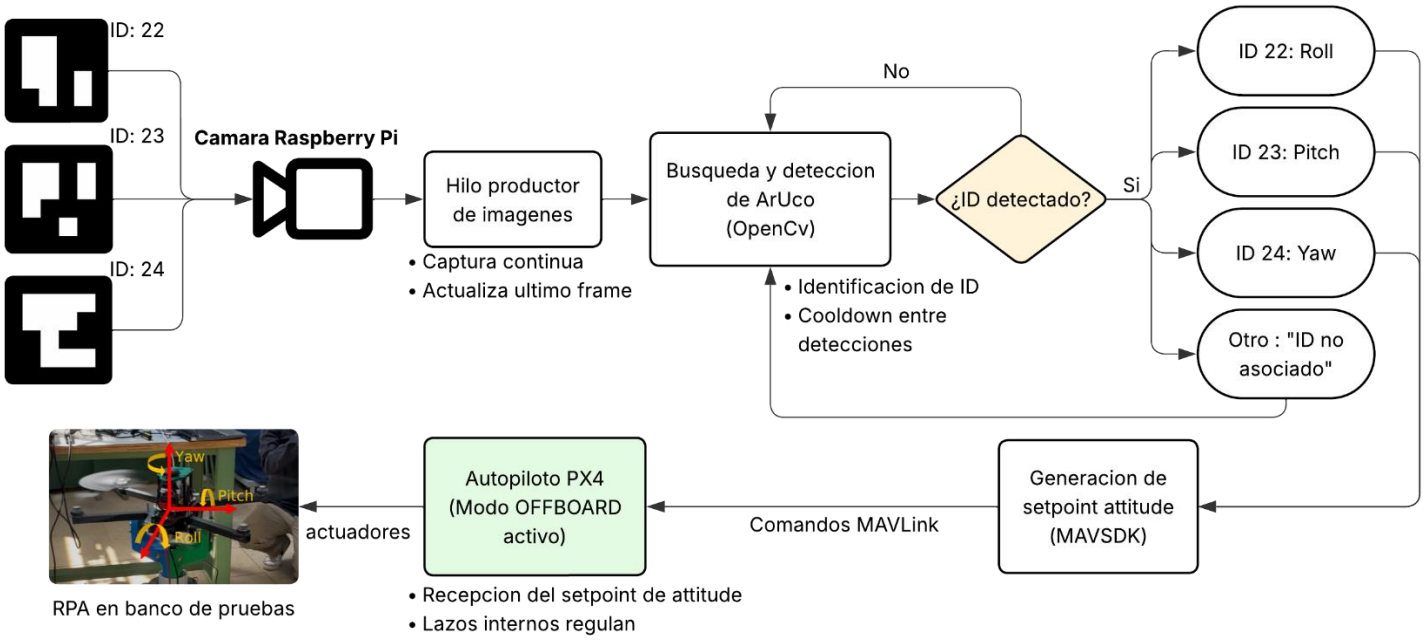


Figura 48: Lógica de la prueba de actitud

Como se puede visualizar en el esquema de la Figura 48, la prueba se basó en un flujo operativo que integra captura de imagen, detección de estímulos y generación de comandos MAVLink. La Raspberry Pi ejecuta un hilo productor encargado de mantener un flujo continuo de imágenes provenientes de la cámara USB. El propósito de este hilo es capturar imágenes de manera continua y actualizar un buffer con el ultimo frame disponible, de manera que siempre se esté “buscando” un identificador. Sobre este flujo se aplica el módulo de detección ArUco que identifica el marcador presente en el campo visual y determina una maniobra a ejecutar. La Tabla 8 resume las acciones realizadas en las pruebas de actitud

ID ArUco	Eje controlado	Angulo objetivo	Duración de la maniobra	Descripción de la acción
ID 22	Roll	$\pm 5°$	2 s	Alabeo hacia la derecha (+5°) y luego a la izquierda (-5°), volviendo luego a su actitud base
ID 23	Pitch	$\pm 5°$	2 s	Cabeceo hacia delante (+5°) y luego hacia atrás (-5°), volviendo a la actitud inicial
ID 24	Yaw	$+ 90°$ (relativos)	4 s	Rotación controlada con respecto a su orientación actual (+90°), tras lo cual el sistema regresa al ángulo original

Tabla 8: Maniobras ejecutadas en modo OFFBOARD

Además, en la lógica del código se establece un enfriamiento de 5 segundos para la detección y evitar saturar de *setpoints* el autopiloto (el cooldown empieza a correr una vez que se detecta el ArUco, pues en pruebas preliminares el buffer se llenaba de información y se ejecutaba el mismo comando independiente de si se mostraba o no el ArUco). Para la prueba los ArUcos fueron mostrados tres veces en cada caso. El código completo se puede visualizar en el Anexo 4.

6.2.3 Resultados de la prueba en el banco de ensayos

Una vez realizadas las pruebas, para facilitar la interpretación de los resultados se diseñó un código en MATLAB que permitiera visualizar el momento exacto en el que el autopiloto cambiaba de modo de vuelo activo, alternando entre *STABILIZED* y *OFFBOARD*. Esto permite identificar con claridad los intervalos exactos en que el sistema autónomo tomo el control. En la Figura 49 se puede visualizar la evolución temporal de los ángulos de actitud durante toda la secuencia de pruebas.

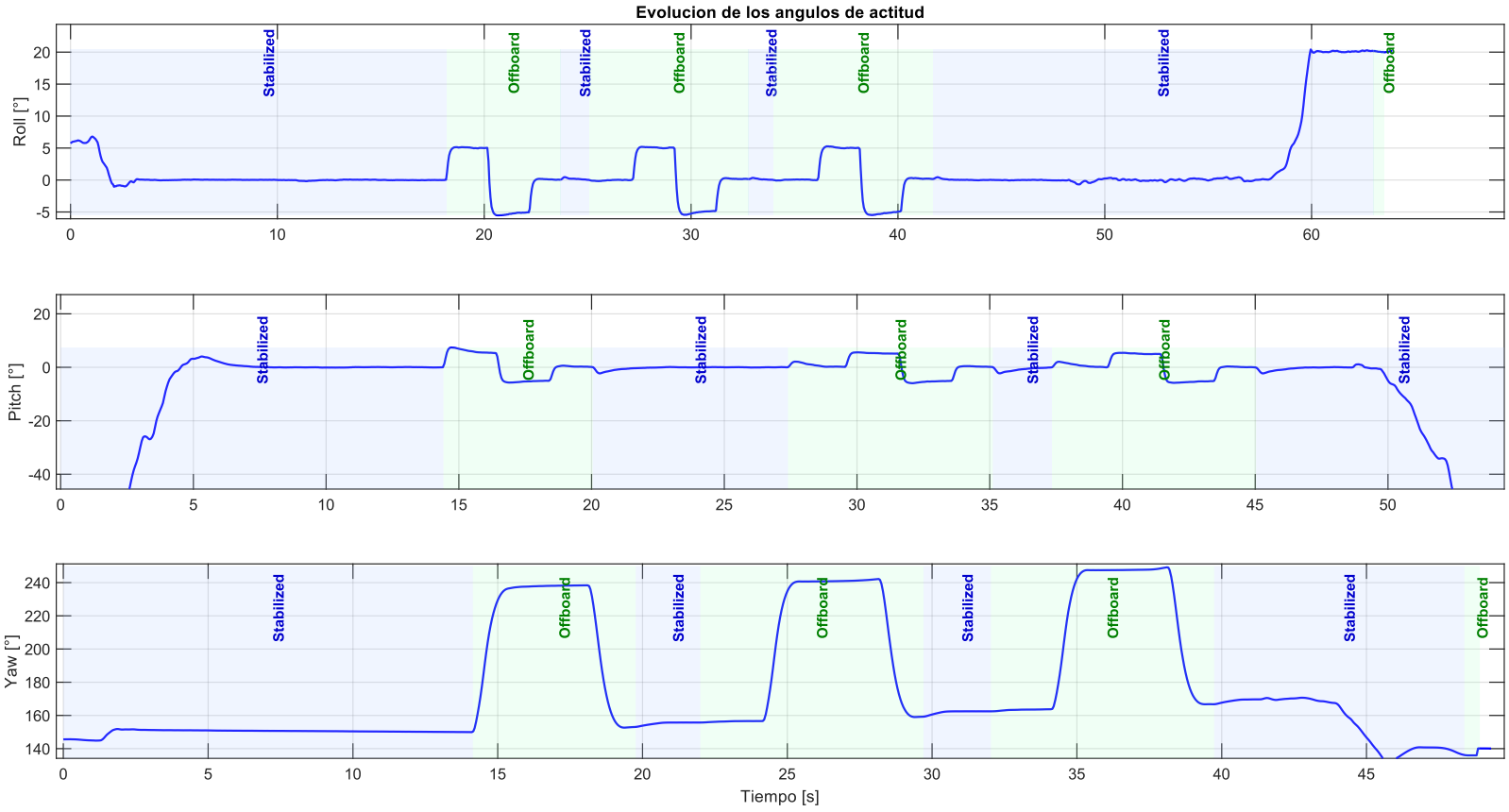


Figura 49: Evolución temporal de los ángulos de actitud.

Utilizando como referencia el tiempo en el que se llegó al *setpoint*, para cada intervalo del modo de vuelo *OFFBOARD* se hizo una estimación manual de la duración efectiva de cada maniobra. En promedio para cada maniobra los tiempos fueron: Roll = 1.68 [s], Pitch = 1.73 [s] y Yaw = 2.96 [s]. Estos valores son ligeramente menores a los programados en el código (2[s] y 4[s]), lo cual tiene sentido pues parte del tiempo útil se usa en tener que llegar al

ángulo objetivo y otra parte en volver a la actitud base, la medición solo cuenta los valores máximos, por lo que los valores son coherentes con lo programado.

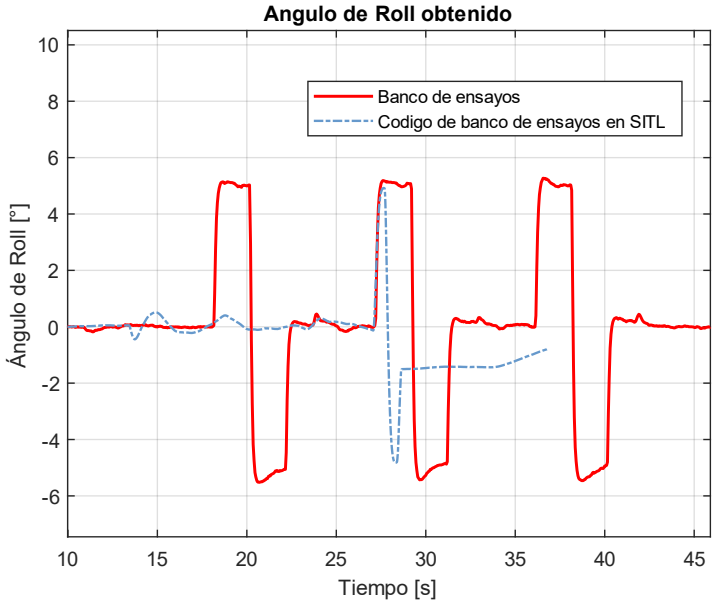


Figura 50: Angulo de Roll obtenido

La respuesta de Roll de la prueba en el banco de ensayos muestra que alcanza de forma clara los *setpoints* programados, donde el controlador corrige el error al pasarse del *setpoint* de 5 [°] llegar. Para comparar el comportamiento, exactamente el mismo código del banco de ensayos se probó en SITL (distinto al diseñado en 5.4, el cual estaba estructurado para simulación). Para una sola detección de ArUco, se puede visualizar una curva mucho más suave, la cual llega al *setpoint* pero no respeta el comando de duración de la maniobra, ni vuelve a la actitud base.

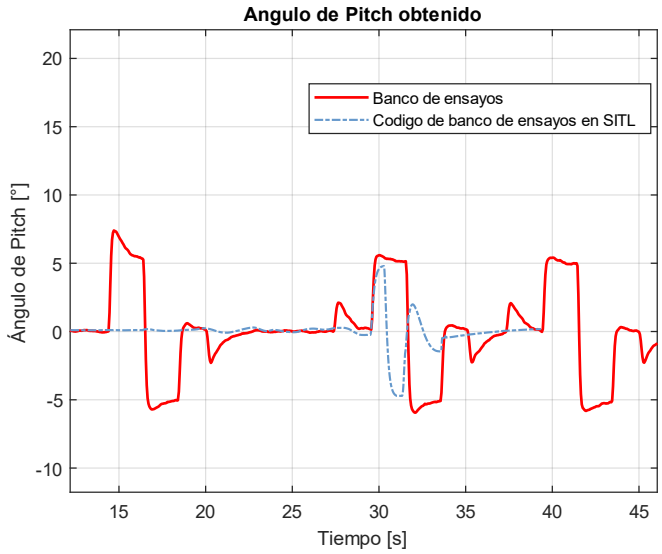


Figura 51: Angulo de Pitch obtenido

En el caso del Pitch, el controlador aparenta ser más brusco a la hora de llegar al *setpoint*, sobre todo en el primer pico, donde sobrepasa ligeramente el pitch objetivo y después corrige a lo largo de los 2[s] que dura la maniobra. Al igual que el Roll, el código al ser ejecutado en el entorno simulado muestra que es mucho más suave a la hora de llegar al *setpoint*, pero de la misma manera ignora por completo el comando de duración de la maniobra.

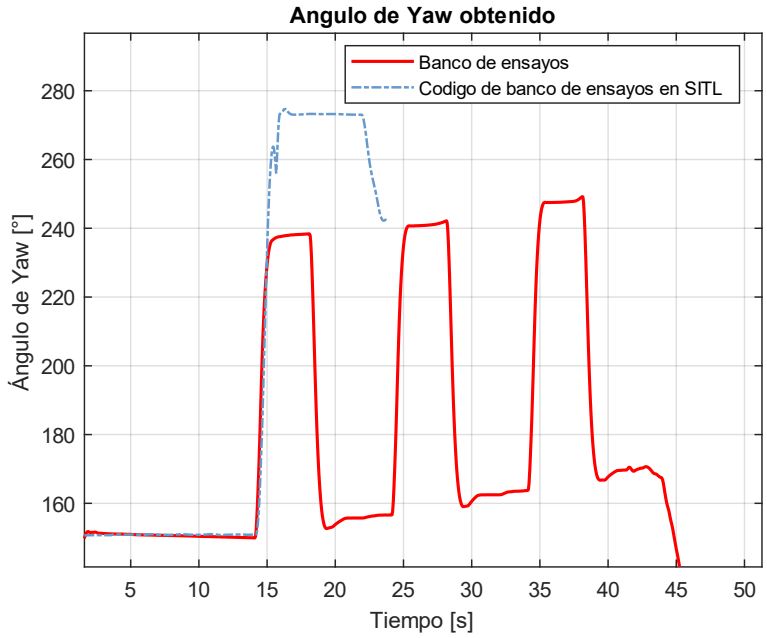


Figura 52: Angulo de Yaw obtenido

En el caso del Yaw, las tres detecciones evaluadas presentan giros de aproximadamente 88°, 84° y 84°, respectivamente, valores que son coherentes con los 90° expuestos en el código. Al finalizar cada giro, el RPA no vuelve del todo exacto a su ángulo inicial, si no que mantiene un desfase acumulado cercano a los 6°, error que se propaga en las maniobras y explica la tendencia ascendente observable, lo cual puede deberse a el propio error en la lectura del magnetómetro con respecto al norte magnético. En la prueba SITL también se aprecia la ejecución de un giro, pero es de magnitud mucho mayor (~120°). En general estos resultados muestran la incompatibilidad de un código que sirva tanto para simulación como para pruebas reales, sin realizar ajustes en su lógica.

7 Capítulo 7 Conclusión

En base al desarrollo del presente proyecto y de los objetivos propuestos, fue posible demostrar la factibilidad técnica de integrar un sistema de decisión autónoma en plataformas RPA mediante un ecosistema hibrido de simulación y hardware real. Este sistema se sustentó mediante la comunicación entre un autopiloto Pixhawk y una Raspberry Pi, a través del protocolo MAVLink.

La implementación del entorno simulado permitió reproducir el comportamiento del RPA utilizando el firmware PX4 dentro de una máquina virtual. En la ejecución de misiones autónomas, el desempeño de la simulación mostró una alta correspondencia con los vuelos reales. El análisis de la trayectoria entrego un RMSE de 0.490 [m] respecto a los *waypoints* en comparación con los 0.623 [m] registrados en el vuelo real. Este resultado confirma que el simulador de físicas, pese a ser idealizado, es capaz de aproximar con buena precisión la dinámica observada en el autopiloto físico. De la misma manera, la implementación del módulo de control externo, previo al uso de una computadora embebida, permitió demostrar que MAVSDK es fundamental a la hora de ejecutar operaciones asincrónicas y ejecutar cambios de misión desde la propia máquina virtual, lo que evidencia que es posible introducir lógica de decisión sin necesidad inicial de hardware adicional.

La puesta en marcha de la Raspberry Pi demostró que la ejecución de algoritmos externos al autopiloto es técnicamente viable, abriendo paso a la incorporación de tareas más complejas como detección visual o generación de trayectorias de vuelo. Esta arquitectura se validó con el firmware en SITL, mediante pruebas de detección, cambios de misión, comandos *goto* y modificaciones de modo de vuelo. Adicionalmente el modo *OFFBOARD* permitió enviar secuencias de *setpoints* destinadas a inducir variaciones controladas en la actitud del RPA. En estas pruebas, las respuestas del sistema estuvieron cercano a los valores definidos por el código, siendo $\pm 10°$ para Roll/Pitch y $\pm 20°$ para Yaw, lo que indico que el controlador del simulador de físicas del firmware respondía adecuadamente a los *setpoints* del modo *OFFBOARD*.

Para la validación sobre el RPA real, se confirmó que la arquitectura diseñada en SITL es trasladable al hardware físico, aunque no de manera directa ni completamente equivalente. La habilitación de comunicación de telemetría requirió configurar parámetros en el autopiloto y la Raspberry Pi, además de la configuración para su uso en entorno sin GPS. El código para las pruebas de actitud diseñadas exclusivamente para el uso en el banco, mostraron que las maniobras ejecutadas en modo *OFFBOARD* presentaron tiempos promedio de llegada al ángulo objetivo, ligeramente menores a los programados en el código. Esto considerando que parte de ese tiempo se emplea en alcanzar el valor y otra en regresar a la actitud base. La respuesta de roll alcanzo el *setpoint* con corrección inmediata tras un leve sobrepaso, mientras que en pitch el primer pico supero ligeramente el valor objetivo para luego estabilizarse dentro del tiempo establecido en el código. En el caso de yaw, se presentó un desfase acumulado cercano a los 6° al retornar a la actitud base, lo que puede ser atribuible al propio error del magnetómetro.

Para comprobar que los códigos elaborados para el banco de ensayos fueran traspasables a SITL, se decidió ejecutarlos en el entorno simulado utilizando la misma configuración de modos y *setpoints*. Sin embargo, la respuesta del simulador mostró diferencias importantes respecto del hardware real, aunque el RPA virtual alcanzo los ángulos objetivos de forma más suave y sin sobrepasos significativos, el controlador ignoro por completo la duración programada de cada maniobra y no retorno a la actitud base al finalizar. Estas discrepancias confirmaron que los códigos diseñados para hardware real son parcialmente funcionales dentro de SITL, pero se requieren ajustes diferenciados para cada entorno.

7.1 Trabajo futuro

Como trabajo futuro, se propone avanzar hacia la ejecución controlada de pruebas en vuelo, con el objetivo de validar la arquitectura completa y cuantificar el desempeño fuera del banco de ensayos. En particular se busca implementar un procedimiento de *autoland* que combine detección visual, control en modo *OFFBOARD* y supervisión del piloto mediante radiocontrol.

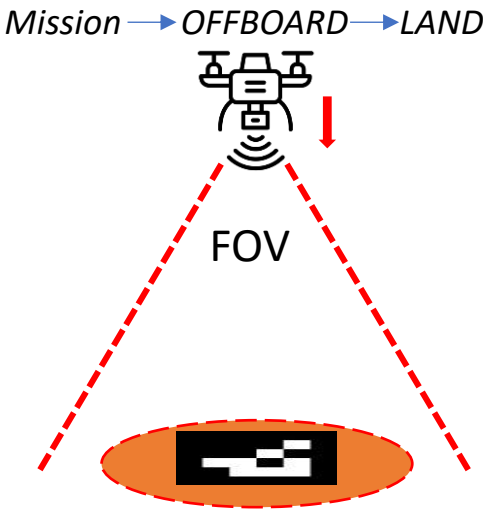


Figura 53: Prueba "*Autoland*"

Además, es necesario estudiar con mayor profundidad el autopiloto al hacer la transición al modo *OFFBOARD*, específicamente en lo referente a la gestión del empuje, ya que durante las pruebas en el banco de ensayo se observó que el cambio de modo puede provocar variaciones no deseadas en la potencia de los motores, por lo que se requiere caracterizar ese efecto y establecer un mecanismo que preserve el nivel de empuje en el modo de vuelo previo antes de iniciar la secuencia de *setpoints*.

Referencias

[1] K. Nonami, “Research and development of drone and roadmap to evolution,” Jun. 01, 2018, *Fuji Technology Press*. doi: 10.20965/jrm.2018.p0322.

[2] M. Hassanalian and A. Abdelkefi, “Classifications, applications, and design challenges of drones: A review,” May 01, 2017, *Elsevier Ltd*. doi: 10.1016/j.paerosci.2017.04.003.

[3] S. Zoting, “Unmanned Aerial Vehicle Market Size, Share, and Trends 2025 to 2034,” 2025. Accessed: Oct. 21, 2025. [Online]. Available: https://www.precedenceresearch.com/unmanned-aerial-vehicle-market

[4] R. Perez-Segui, P. Arias-Perez, J. Melero-Deza, M. Fernandez-Cortizas, D. Perez-Saura, and P. Campoy, “Bridging the Gap between Simulation and Real Autonomous UAV Flights in Industrial Applications,” *Aerospace*, vol. 10, no. 9, Sep. 2023, doi: 10.3390/aerospace10090814.

[5] C. Coopmans, M. Podhradský, and N. V. Hoffer, “Software- and hardware-in-the-loop verification of flight dynamics model and flight control simulation of a fixed-wing unmanned aerial vehicle,” in *2015 Workshop on Research, Education and Development of Unmanned Aerial Systems, RED-UAS 2015*, Institute of Electrical and Electronics Engineers Inc., Mar. 2016, pp. 115–122. doi: 10.1109/RED-UAS.2015.7440998.

[6] H. Shakhatreh *et al.*, “Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges,” 2019. doi: 10.1109/ACCESS.2019.2909530.

[7] A. Thiercelin, I. Galloway, C. Schwarzmiller, and A. Klimaj, “Dronecode Project.” Accessed: Aug. 06, 2025. [Online]. Available: https://pixhawk.org/

[8] K. M. Hosny, A. Magdi, A. Salah, O. El-Komy, and N. A. Lashin, “Internet of things applications using Raspberry-Pi: a survey,” *International Journal of Electrical and Computer Engineering*, vol. 13, no. 1, pp. 902–910, Feb. 2023, doi: 10.11591/ijece.v13i1.pp902-910.

[9] A. Koubaa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui, “Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey,” *IEEE Access*, vol. 7, pp. 87658–87680, 2019, doi: 10.1109/ACCESS.2019.2924410.

[10] A. Berndt Barriga, “Implementación y puesta a punto de banco de ensayos de 4 grados de libertad para drones multirotor POR UNIVERSIDAD DE CONCEPCIÓN FACULTAD DE INGENIERÍA DEPARTAMENTO INGENIERÍA MECÁNICA,” 2024.

[11] International Civil Aviation Organization, “Unmanned Aircraft Systems (UAS),” in *ICAO Cir 328, Unmanned Aircraft Systems (UAS)*, 2011.

[12] 3D ROBOTICS, “X8 Operation Manual,” 2016. [Online]. Available: www.Manualslib.com

[13] R. W.Beard and T. W. McLain, *Small Unmanned Aircraft Theory and Practice*. 2012.

[14] PX4 autopilot, “PX4 Autopilot,” 2025. Accessed: Sep. 15, 2025. [Online]. Available: https://docs.px4.io/main/en/

[15] Ardupilot, “Ardupilot,” 2025. Accessed: Sep. 15, 2025. [Online]. Available: https://ardupilot.org/

[16] Dronecode Project MAVLink (2024), “Mavlink Developer Guide.”

[17] P. Marwedel, *Embedded System Design*, 4th ed. 2021.

[18] J. Pramanik, A. K. Samal, S. K. Pani, and C. Chakraborty, “Elementary framework for an IoT based diverse ambient air quality monitoring system,” *Multimed Tools Appl*, vol. 81, no. 26, pp. 36983–37005, Nov. 2022, doi: 10.1007/s11042-021-11285-1.

[19] Raspberry Pi foundation, “Raspberry Pi (2025).” Accessed: Sep. 16, 2025. [Online]. Available: https://www.raspberrypi.com/

[20] ardupilot.org, “Communicating with Raspberry Pi via MAVLink,” 2025. Accessed: Sep. 16, 2025. [Online]. Available: https://ardupilot.org/dev/docs/raspberry-pi-via-mavlink.html

[21] QGroundControl, “QGroundControl -QGroundControl. QGroundControl – Drone Control – Ground Control Station for MAVLink-enabled drones. .” Accessed: Sep. 16, 2025. [Online]. Available: https://qgroundcontrol.com/

[22] Canonical ltd., “Ubuntu ROS.” Accessed: Sep. 20, 2025. [Online]. Available: https://ubuntu.com/robotics

[23] L. Joseph and A. Johny, *Robot Operating System (ROS) for absolute beginners: Robotics programming made easy*. Apress Media LLC, 2022. doi: 10.1007/978-1-4842-7750-8.

[24] N. Koenig and A. Howard, “Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator,” 2004.

[25] PX4 autopilot, “Gazebo Simulation PX4 Guide (main).” Accessed: Sep. 20, 2025. [Online]. Available: https://docs.px4.io/main/en/sim_gazebo_gz

[26] Dronecode Foundation, “MAVSDK (main / v3) | MAVSDK guide.” Accessed: Sep. 28, 2025. [Online]. Available: https://mavsdk.mavlink.io/main/en/index.html

[27] Python Software Foundation, “Python Documentation.” Accessed: Sep. 28, 2025. [Online]. Available: https://www.python.org/doc/

[28] Inc. GitHub, “GitHub Docs.” Accessed: Sep. 28, 2025. [Online]. Available: https://docs.github.com/

[29] L. D. Ortega, E. S. Loyaga, P. J. Cruz, H. P. Lema, J. Abad, and E. A. Valencia, “Low-Cost Computer-Vision-Based Embedded Systems for UAVs,” *Robotics*, vol. 12, no. 6, Dec. 2023, doi: 10.3390/robotics12060145.

[30] Raspberry Pi Foundation, “Raspberry Pi Dramble Power Consumption Benchmarks,” 2025. Accessed: Oct. 21, 2025. [Online]. Available: https://www.pidramble.com/wiki/benchmarks/power-consumption

[31] C. R. De Cos, M. J. Fernandez, P. J. Sanchez-Cuevas, J. A. Acosta, and A. Ollero, “High-Level Modular Autopilot Solution for Fast Prototyping of Unmanned Aerial Systems,” *IEEE Access*, vol. 8, pp. 223827–223836, 2020, doi: 10.1109/ACCESS.2020.3044098.

[32] dronekit, “DroneKit-Python library for communicating with Drones via MAVLink.,” 2017. Accessed: Oct. 16, 2025. [Online]. Available: https://github.com/dronekit/dronekit-python

[33] Oracle, “VirtualBox,” 2025. Accessed: Oct. 16, 2025. [Online]. Available: https://www.virtualbox.org/

[34] Dronecode Foundation, “PX4 Drone Autopilot,” 2025. Accessed: Oct. 21, 2025. [Online]. Available: https://github.com/PX4/PX4-Autopilot

[35] PX4 Team, “PX4 Flight Review.” Accessed: Oct. 21, 2025. [Online]. Available: https://review.px4.io/

[36] Python Software Foundation, “asyncio — Asynchronous I/O,” Oct. 2025. Accessed: Oct. 21, 2025. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[37] Google, “Google Earth Pro,” 2024.

[38] doxygen, “Open Source Computer Vision”, Accessed: Oct. 21, 2025. [Online]. Available: https://docs.opencv.org/4.x/d9/d6a/group__aruco.html

[39] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, “Automatic generation and detection of highly reliable fiducial markers under occlusion,” *Pattern Recognit*, vol. 47, no. 6, pp. 2280–2292, Jun. 2014, doi: 10.1016/j.patcog.2014.01.005.

[40] Texas Instruments, “LM2596 SIMPLE SWITCHER ® Power Converter 150-kHz 3-A Step-Down Voltage Regulator,” 2023. [Online]. Available: www.ti.com

[41] Ardupilot, “MAVProxy,” 2024. Accessed: Nov. 14, 2025. [Online]. Available: https://ardupilot.org/mavproxy/

Anexo

1. carta Gantt

RESULTADO	OBJETIVOS	ACTIVIDADES	HITO	ACTIVIDAD (indique el mes en que se iniciará la actividad)	DURACIÓN DE LA ACTIVIDAD (meses)	(indique el mes en el que se logrará el hito respecto de la duración total del proyecto)
	Objetivo 1.- Identificar arquitecturas y					
Resultado 1 : OE1	metodologías previas	1.- Recolección y análisis de papers		1	17	
		2.- Síntesis del estado del arte		1	5	
		3.- Definicion de la problemática y Metodologia		3	3	
		4.- Redaccion de Informe I	Hito 1 - Informe Avance I	1	5	5
	Objetivo 2.- Validar entorno SITL	1.- Configuiracion SITL		3	4	
		2.- Misiones de prueba en Simulador		3	4	
		3.- Integracion Raspberry Pi		7	5	
		4.- Redaccion de informe II	Hito 2 - Informe Avance II	8	3	10
	Objetivo 3.- Consolidar el ecosistema con pruebas					
Resultado 2: OE2	en simulacion y hardware	1.- Validacion de algoritmo en entorno SITL + Rpi		7	5	
		2.- Desarrollo de algoritmo de deteccion e implementacion de sensor		11	3	
		3.- Implementacion en RPA físico en condiciones controladas	Hito 3 - Mision de vuelo	13	2	
		4.- Analisis comparativo, pruebas extra		14	3	
	Objetivo 4.- Entrega Final					
Resultado 3: OE3	y Defensa	1.- Redaccion informe Final	Hito 4 - Informe Final	13	6	18
		2.- Defensa	Hito 5 - Defensa	19	1	

SEMANAS

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

2. Registros de pruebas de interrupción en Raspberry Pi

2.1 Caso 1, sin estimulo

```
04:34:40 | Inicio de sesión de prueba OE2
04:34:53 | Autopiloto listo
04:34:53 | Sistema listo. Inicia misión LTA y espera estímulos.
04:35:53 | 1 min sin estímulo (Caso 1 completado)
04:35:53 | Fin de ejecución. Log guardado.
```

2.2 Caso 2, carga de misión "Plan B"

```
05:36:07 | Inicio de sesión de prueba OE2
05:36:24 | Autopiloto listo
05:36:25 | Sistema listo. Inicia misión LTA y espera estímulos.
05:37:09 | Detección switch (flanco ON) t=1760323029.182
05:37:09 | INTERRUPCIÓN POR SWITCH → Plan B
05:37:09 | Error al pausar misión (intento 1): <AioRpcError of RPC that terminated with:
    status = StatusCode.UNAVAILABLE
    details = "recvmsg:Connection reset by peer"
    debug_error_string = "UNKNOWN:Error received from peer {grpc_status:14, grpc_message:"recvmsg:Connection reset by peer"}"
>
05:37:09 | Reintentando conexión con PX4...
05:37:20 | Autopiloto listo
05:37:22 | Pausa de misión exitosa (intento 2)
05:37:23 | Plan B iniciado correctamente
05:37:23 | Latencia estimada: 13.828s
05:38:50 | Interrumpido manualmente por el usuario.
```

2.3 Caso 3, desvió directo por detección de ArUco

```
06:37:03 | Inicio de sesión de prueba OE2
06:37:13 | Autopiloto listo
06:37:13 | Sistema listo. Inicia misión LTA y espera estímulos.
```

06:37:45 | Detección ArUco ID22 (t=1760326665.918)

06:37:45 | INTERRUPCIÓN POR ARUCO → GOTO objetivo

06:37:45 | Error al pausar misión (intento 1): <AioRpcError of RPC that terminated with:

status = StatusCode.UNAVAILABLE

details = "recvmsg:Connection reset by peer"

debug_error_string = "UNKNOWN:Error received from peer {grpc_message:"recvmsg:Connection reset by peer", grpc_status:14}"

>

06:37:45 | Reintentando conexión con PX4...

06:37:58 | Autopiloto listo

06:38:00 | Pausa de misión exitosa (intento 2)

06:38:00 | Comando GOTO enviado correctamente

06:38:00 | Latencia estimada: 15.054s

06:38:04 | Detección ArUco ID22 (t=1760326684.017)

06:38:04 | INTERRUPCIÓN POR ARUCO → GOTO objetivo

06:38:04 | Pausa de misión exitosa (intento 1)

06:38:04 | Comando GOTO enviado correctamente

06:38:04 | Latencia estimada: 0.173s

06:39:15 | Interrumpido manualmente por el usuario.

2.4 Caso 4, mision combinada

07:03:09 | Inicio de sesión de prueba OE2 (versión sobrescritura)

07:03:23 | Autopiloto listo

07:03:26 | Sistema listo. Inicia misión LTA y espera estímulos.

07:04:02 | Detección ArUco ID22 (t=1760328242.558)

07:04:02 | INTERRUPCIÓN POR ARUCO → GOTO (sobrescribir misión)

07:04:02 | Error al limpiar misión (GOTO): <AioRpcError of RPC that terminated with:

status = StatusCode.UNAVAILABLE

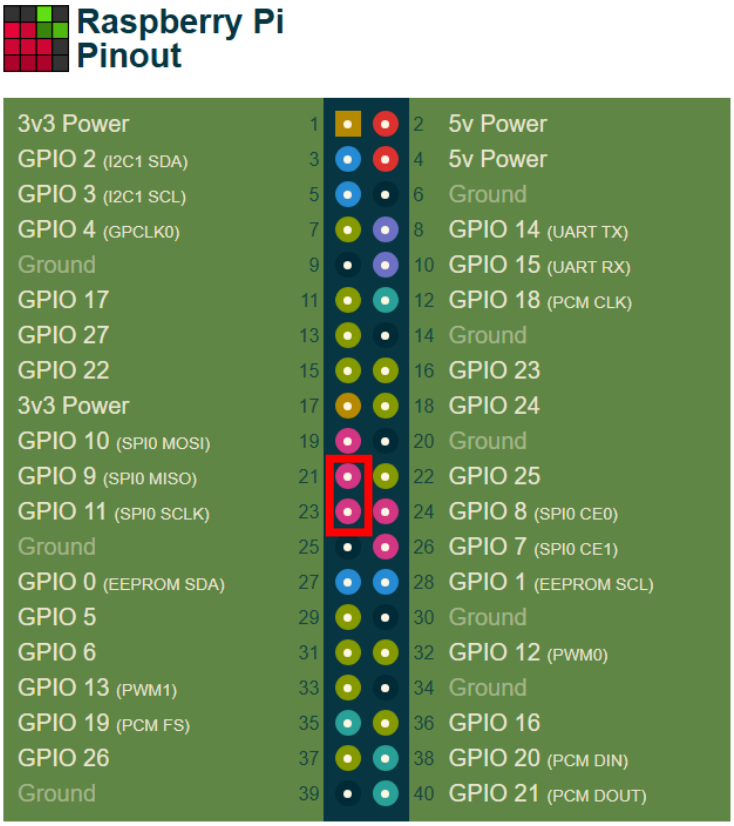
details = "recvmsg:Connection reset by peer"

debug_error_string = "UNKNOWN:Error received from peer {grpc_message:"recvmsg:Connection reset by peer", grpc_status:14}"

```
>
07:04:15 | Autopiloto listo
07:04:17 | Misión GOTO iniciada (sobrescritura exitosa)
07:04:17 | Latencia estimada: 15.403s
07:04:20 | Detección ArUco ID22 (t=1760328260.992)
07:04:20 | INTERRUPCIÓN POR ARUCO → GOTO (sobrescribir misión)
07:04:21 | Misión GOTO iniciada (sobrescritura exitosa)
07:04:21 | Latencia estimada: 0.459s
07:04:25 | Detección ArUco ID22 (t=1760328265.402)
07:04:25 | INTERRUPCIÓN POR ARUCO → GOTO (sobrescribir misión)
07:04:25 | Misión GOTO iniciada (sobrescritura exitosa)
07:04:25 | Latencia estimada: 0.512s
07:04:28 | Detección ArUco ID22 (t=1760328268.977)
07:04:28 | INTERRUPCIÓN POR ARUCO → GOTO (sobrescribir misión)
07:04:29 | Misión GOTO iniciada (sobrescritura exitosa)
07:04:29 | Latencia estimada: 0.322s
07:04:32 | Detección ArUco ID22 (t=1760328272.315)
07:04:32 | INTERRUPCIÓN POR ARUCO → GOTO (sobrescribir misión)
07:04:32 | Misión GOTO iniciada (sobrescritura exitosa)
07:04:32 | Latencia estimada: 0.441s
07:04:35 | Detección ArUco ID22 (t=1760328275.768)
07:04:35 | INTERRUPCIÓN POR ARUCO → GOTO (sobrescribir misión)
07:04:36 | Misión GOTO iniciada (sobrescritura exitosa)
07:04:36 | Latencia estimada: 0.426s
07:04:51 | Detección switch (flanco ON) t=1760328291.717
07:04:51 | INTERRUPCIÓN POR SWITCH → Plan B (sobrescribir misión)
07:04:52 | Plan B iniciado (nueva misión cargada)
07:04:52 | Latencia estimada: 0.452s
07:05:09 | Detección switch (flanco ON) t=1760328309.439
07:05:09 | INTERRUPCIÓN POR SWITCH → Plan B (sobrescribir misión)
07:05:10 | Plan B iniciado (nueva misión cargada)
```

```
07:05:10 | Latencia estimada: 0.813s
07:05:59 | Detección ArUco ID22 (t=1760328359.170)
07:05:59 | INTERRUPCIÓN POR ARUCO → GOTO (sobrescribir misión)
07:05:59 | Misión GOTO iniciada (sobrescritura exitosa)
07:05:59 | Latencia estimada: 0.269s
07:06:26 | 3 min sin estímulo (Caso 1 completado)
07:06:27 | Fin de ejecución. Log guardado.
```

3. Pinout Raspberry Pi



4. Código banco de ensayos

```
import asyncio
import cv2
import threading
import time
from mavsdk import System
from mavsdk.offboard import OffboardError, Attitude
```

```
SERIAL_PORT = "serial:///dev/serial0:57600"
CAMERA_DEVICE_INDEX = 0
ARUCO_DICT_TYPE = cv2.aruco.DICT_4X4_50
COOLDOWN_S = 5.0
ROLL_POSITIVO_S = 2.0; ROLL_NEGATIVO_S = 2.0; ANGULO_ROLL_GRADOS = 5.0
PITCH_POSITIVO_S = 2.0; PITCH_NEGATIVO_S = 2.0; ANGULO_PITCH_GRADOS = 5.0
YAW_RELATIVO_GRADOS = 90.0; YAW_DURACION_S = 4.0


class DroneVisionController:
    def __init__(self):
        self.drone = System()
        self.last_action_time = 0
        self.camera = cv2.VideoCapture(CAMERA_DEVICE_INDEX)
        self.latest_frame = None
        self.camera_lock = threading.Lock()
        self.is_running = True
        self.camera_thread = threading.Thread(target=self._camera_producer_loop, daemon=True)


    def _camera_producer_loop(self):
        print("[Cámara] Hilo productor iniciado.")
        if not self.camera.isOpened(): print("[Cámara-ERROR] No se pudo abrir stream."); return
        while self.is_running:
            ret, frame = self.camera.read()
            if ret:
                with self.camera_lock: self.latest_frame = frame
            else: time.sleep(0.1)
        self.camera.release(); print("[Cámara] Hilo productor detenido.")


    def get_latest_frame(self):
        with self.camera_lock: return self.latest_frame.copy() if self.latest_frame is not None
else None


    async def connect_and_prepare(self):
        await self.drone.connect(system_address=SERIAL_PORT)
        print("Esperando conexión con PX4...");
        async for state in self.drone.core.connection_state():
            if state.is_connected: print("Conectado al autopiloto PX4"); break
        print("Iniciando hilo de la cámara..."); self.camera_thread.start(); await
asyncio.sleep(2)
```

```
        if self.get_latest_frame() is None: raise RuntimeError("La cámara no está produciendo
frames.")
        print("Cámara funcionando y produciendo frames.")



    async def _get_current_attitude_euler(self):
        """Devuelve los ángulos de Euler actuales del dron."""
        async for euler in self.drone.telemetry.attitude_euler():
            return euler.roll_deg, euler.pitch_deg, euler.yaw_deg


    async def _run_offboard_maneuver(self, maneuver_coro):
        """Wrapper seguro que gestiona el ciclo de vida de Offboard."""
        print("--- Iniciando secuencia de maniobra Offboard ---")
        try:
            # 1. Obtener actitud base
            base_roll, base_pitch, base_yaw = await self._get_current_attitude_euler()
            print(f"Actitud     base     leída:     R:{base_roll:.1f},     P:{base_pitch:.1f},
Y:{base_yaw:.1f}")


            # 2. Calentamiento con la actitud base
            print("Calentando para Offboard...")
            attitude_base = Attitude(base_roll, base_pitch, base_yaw, 0.5)
            for _ in range(40):
                await self.drone.offboard.set_attitude(attitude_base)
                await asyncio.sleep(0.05)


            # 3. Iniciar Offboard y ejecutar maniobra
            print("Iniciando modo OFFBOARD...")
            await self.drone.offboard.start()
            await maneuver_coro(base_roll, base_pitch, base_yaw)


        except OffboardError as e:
            print(f"[ERROR] Offboard falló durante la secuencia: {e._result.result}.")
        except Exception as e:
            print(f"[ERROR] Ocurrió una excepción inesperada: {e}")
        finally:
            # 4. GARANTÍA DE SALIDA: Este bloque se ejecuta SIEMPRE.
            print("Secuencia finalizada. Saliendo de modo OFFBOARD...")
            try:
```

```
                await self.drone.offboard.stop()
                print("Modo OFFBOARD detenido exitosamente.")
            except OffboardError as e:
                print(f"[ERROR] No se pudo detener Offboard limpiamente: {e._result.result}")
        print("--- Secuencia de maniobra Offboard completada ---")




    async def _do_roll_maneuver(self, base_roll, base_pitch, base_yaw):
        print(f"Ejecutando Roll sobre la base R:{base_roll:.1f}, P:{base_pitch:.1f},
Y:{base_yaw:.1f}")
        # Fase 1: Roll Positivo
        actitud_positiva = Attitude(base_roll + ANGULO_ROLL_GRADOS, base_pitch, base_yaw, 0.5)
        print(f"Fase 1: Enviando Roll a {actitud_positiva.roll_deg:.1f}°")
        await self.drone.offboard.set_attitude(actitud_positiva)
        await asyncio.sleep(ROLL_POSITIVO_S)

        # Fase 2: Roll Negativo
        actitud_negativa = Attitude(base_roll - ANGULO_ROLL_GRADOS, base_pitch, base_yaw, 0.5)
        print(f"Fase 2: Enviando Roll a {actitud_negativa.roll_deg:.1f}°")
        await self.drone.offboard.set_attitude(actitud_negativa)
        await asyncio.sleep(ROLL_NEGATIVO_S)

        # Fase 3: Volver a la actitud base para una transición suave
        actitud_base_final = Attitude(base_roll, base_pitch, base_yaw, 0.5)
        print("Restaurando actitud base...")
        await self.drone.offboard.set_attitude(actitud_base_final)
        await asyncio.sleep(0.5)

    async def _do_pitch_maneuver(self, base_roll, base_pitch, base_yaw):
        print(f"Ejecutando Pitch sobre la base R:{base_roll:.1f}, P:{base_pitch:.1f},
Y:{base_yaw:.1f}")
        # Fase 1: Pitch Positivo
        actitud_positiva = Attitude(base_roll, base_pitch + ANGULO_PITCH_GRADOS, base_yaw, 0.5)
        print(f"Fase 1: Enviando Pitch a {actitud_positiva.pitch_deg:.1f}°")
        await self.drone.offboard.set_attitude(actitud_positiva)
        await asyncio.sleep(PITCH_POSITIVO_S)

        # Fase 2: Pitch Negativo
```

```
            actitud_negativa = Attitude(base_roll, base_pitch - ANGULO_PITCH_GRADOS, base_yaw, 0.5)
            print(f"Fase 2: Enviando Pitch a {actitud_negativa.pitch_deg:.1f}°")
            await self.drone.offboard.set_attitude(actitud_negativa)
            await asyncio.sleep(PITCH_NEGATIVO_S)

            # Fase 3: Volver a la actitud base
            actitud_base_final = Attitude(base_roll, base_pitch, base_yaw, 0.5)
            print("Restaurando actitud base...")
            await self.drone.offboard.set_attitude(actitud_base_final); await asyncio.sleep(0.5)

    async def _do_yaw_maneuver(self, base_roll, base_pitch, base_yaw):
        print(f"Ejecutando Yaw sobre la base R:{base_roll:.1f}, P:{base_pitch:.1f},
Y:{base_yaw:.1f}")
        yaw_objetivo = (base_yaw + YAW_RELATIVO_GRADOS) % 360
        print(f"Yaw objetivo: {yaw_objetivo:.1f}°")

        actitud_yaw = Attitude(base_roll, base_pitch, yaw_objetivo, 0.5)
        await self.drone.offboard.set_attitude(actitud_yaw)
        await asyncio.sleep(YAW_DURACION_S)

        # Fase 2: Volver a la actitud base
        actitud_base_final = Attitude(base_roll, base_pitch, base_yaw, 0.5)
        print("Restaurando actitud base...")
        await self.drone.offboard.set_attitude(actitud_base_final); await asyncio.sleep(0.5)

    async def run(self):
        await self.connect_and_prepare()
        dict_ = cv2.aruco.getPredefinedDictionary(ARUCO_DICT_TYPE); params =
cv2.aruco.DetectorParameters_create()
        print("\n Sistema listo..."); print("Ctrl+C para salir.")
        while self.is_running:
            frame = self.get_latest_frame()
            if frame is None: await asyncio.sleep(0.1); continue
            corners, ids, _ = cv2.aruco.detectMarkers(frame, dict_, parameters=params)
            cooldown_passed = (time.time() - self.last_action_time) > COOLDOWN_S
            if ids is not None and cooldown_passed:
                id_detectado = ids[0][0]; print(f"Macador ArUco con ID {id_detectado}
detectado.")
                self.last_action_time = time.time(); is_armed = False
```

```
                async for armed in self.drone.telemetry.armed(): is_armed = armed; break
                if not is_armed: print("[WARN] Dron no armado."); continue
                if             id_detectado             ==             22:             await
self._run_offboard_maneuver(self._do_roll_maneuver)
                elif             id_detectado             ==             23:             await
self._run_offboard_maneuver(self._do_pitch_maneuver)
                elif             id_detectado             ==             24:             await
self._run_offboard_maneuver(self._do_yaw_maneuver)
                else: print(f"ID {id_detectado} no asociado a prueba.")
                print("Reanudando detección...")
            await asyncio.sleep(0.1)


    def stop(self):
        self.is_running = False
        if self.camera_thread.is_alive(): self.camera_thread.join()
        cv2.destroyAllWindows(); print("Programa finalizado.")


if __name__ == "__main__":
    controller = DroneVisionController()
    try: asyncio.run(controller.run())
    except KeyboardInterrupt: print("\nInterrupción por teclado detectada.")
    finally: controller.stop()
```