



UNIVERSIDAD AERONÁUTICA EN QUERÉTARO

**DISEÑO Y SIMULACIÓN DE UN SISTEMA DE VISIÓN
ARTIFICIAL PARA EL RECONOCIMIENTO DE
COMPUERTAS EN CONJUNTO CON UN ALGORITMO
DE SEGUIMIENTO DE TRAYECTORIA DE VUELO
DENTRO DE UN AMBIENTE VIRTUAL DE CÓDIGO
ABIERTO PARA UN CUADRICÓPTERO AUTÓNOMO**

T E S I S

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN ELECTRÓNICA Y CONTROL DE SISTEMAS DE
AERONAVES

PRESENTA

AXEL RAMIREZ LINAREZ

TUTOR

MCSD MOISÉS TORRES RIVERA

Querétaro, mayo 2022

AGRADECIMIENTOS

A mis padres,

que sin ellos no sería quien soy y no estaría donde estoy; los quiero. Les dedico el esfuerzo, la escritura y la propiedad intelectual de este trabajo.

Mamá, gracias por ser la persona más bondadosa que he conocido, por apoyarme incondicionalmente y por motivarme a alcanzar mis metas.

Papá, te agradezco por haberme ayudado a formar carácter, por ayudarme a creer en mi persona y por haberme apoyado durante todo este tiempo.

A mis hermanas,

que hacen que la vida tenga sentido, orden y dirección para mi persona.



RESUMEN

En este trabajo se integra un conjunto de software de código libre con el objetivo de proponer e implementar un ambiente de simulación para software in the loop, que permita la validación de un sistema propio para la misión de vuelo de un quadrotor; el sistema en cuestión se encuentra integrado por un lado, por un algoritmo de visión por computadora, que cumple con la función de detectar un tipo de compuerta especial que utilizado en competencias de drones autónomos para delimitar un circuito de vuelo, y por otro lado, por un algoritmo de seguimiento de trayectoria basado en waypoints, en donde un programa desarrollado desde cero, se comunica con el firmware de un piloto automático simulado, de tal forma que se envían comandos de vuelo específicos para que el dron sea capaz de volar a través de una serie de compuertas que definen la trayectoria a seguir.

Como se mencionó anteriormente, el sistema de misión de vuelo propuesto está conformado por una serie de aplicaciones y paquetes, entre los cuales se tienen los siguientes:

OpenCV: una paquetería robusta para aplicaciones de visión artificial, con ella se implementó la detección de compuertas

PymavLink: una librería que cuenta con una API basada en MAVLink, con la cual se implementó el seguimiento de trayectoria del dron

Gazebo: un ambiente de simulación para robots, en donde se elaboró el circuito de vuelo

ArduPilot: un firmware para pilotos automáticos que ofrece herramientas para rea-

lizar simulación de software in the loop.

ROS 2: la nueva versión del framework de desarrollo de aplicaciones de robótica; parte esencial para la integración de los algoritmos.

GNU/Linux: el sistema operativo en donde se desarrolló el proyecto en su conjunto.

Por último, se documenta con gran detalle el proceso de integración de todas las herramientas de software utilizadas, así como el comportamiento final del sistema propuesto.



GLOSARIO

Terminología

Algoritmo: conjunto finito y ordenado de instrucciones que representan la solución a un problema.

Aprendizaje profundo: del inglés '*Deep Learning*', es una subárea de la inteligencia artificial, en donde el modelo de aprendizaje se basa en una gran conjunto de capas (de entrada, salidas y ocultas) compuestas por redes neuronales artificiales. Cada capa se especializa una tarea de predicción específica, de tal forma que una máquina es capaz de aprender por sí misma, sin necesidad de intervención humana.

Arquitectura: dentro del campo de estudio de la inteligencia artificial, se refiere a las conexiones o el patrón de diseño de una red neuronal artificial.

Código abierto: también conocido como *software libre*, es un modelo de desarrollo de software que se fundamenta en la colaboración abierta, en donde cualquier usuario tiene la libertad de ejecutar, copiar, distribuir, modificar y contribuir a la mejora del software.

Comando de vuelo: dentro del contexto del firmware para pilotos automáticos, se refiere a instrucciones de alto nivel para que el piloto automático lleve a la aeronave a un estado deseado, dígase actitud, rumbo, etc.

Convolución: operador matemático que representa la integral del producto de dos funciones, en donde una de las señales se encuentra trasladada e invertida.

Entrenamiento: en el campo de estudio de la inteligencia artificial, se refiere al conjunto de métodos a partir de los cuales una máquina es capaz de aprender.

Espacio de color: también conocido como *modelo de color*, se refiere al modelo matemático utilizado para describir los distintos sistemas mediante los cuales se pueden representar los colores, a partir de arreglos de 3 o 4 parámetros, generalmente.

Firmware: es el software base que viene incluido en los dispositivos electrónicos o hardware, y se encarga de asegurar un funcionamiento básico correcto. También es conocido como *soporte lógico inalterable*.

Fotograma: cada una de las imágenes fijas, que en su conjunto forman una imagen en movimiento o video.

Framework: en español *entorno de trabajo*, es una estructura que integra tecnologías, estándares y módulos de software que sirve como base para el desarrollo software.

Hardware: corresponde a los recursos físicos que componen o integran un equipo de cómputo o dispositivo lógico.

Hardware in the loop: paradigma de simulación para la validación de sistemas embedidos en donde la planta que se desea controlar se simula a partir de un modelo matemático, mientras que el sistema de control es físico e interactúa de forma directa con la simulación de la planta.

Histograma de color: es la cuantificación de la distribución de color en una imagen, generalmente se representa con una gráfica en donde se observa la frecuencia de pixeles del mismo color.

Interfaz de programación de aplicación: del inglés *Application Programming Interface*; se trata de un conjunto de definiciones y protocolos que permiten la comunicación o integración entre diferentes aplicaciones de software.

Librería: también conocidas como *bibliotecas*, es un conjunto de módulos o métodos funcionales de software, que fueron codificados para ofrecer una funcionalidad específica y bien definida.

Machine learning: conocido en español como *aprendizaje automático*, es una subárea del campo de la inteligencia artificial en donde se implementa modelo matemático para que un sistema sea capaz de aprender a partir del procesamiento de datos sin la necesidad de especificar una programación explícita.

Máquina de estados: es un modelo que describe el comportamiento de un sistema a partir de una serie de estados finitos, en donde la transición entre cada uno depende de la entrada actual del proceso y la o las entradas anteriores.

Matiz: en el modelo de color HSV, corresponde a un ángulo dentro del rango de 0 a 360 grados, en donde cada grado está asociado a una tonalidad de color en específico.

Multiplataforma: dicho de una aplicación de software que se encuentra disponible para su ejecución en distintos sistemas operativos o sistemas.

Odometría: es el área que se encarga del estudio de la estimación de posición de cualquier tipo de vehículo durante su navegación.

Quadrotor: aeronave de despegue y aterrizaje vertical que es levantado y propulsado por cuatro motores.

Rapid control prototyping: paradigma de validación de sistemas en donde un prototipo físico de la planta interactúa con un modelo matemático o simulación del controlador de esta.

Red neuronal artificial: es un sistema informático que busca emular las redes neuronales biológicas a partir de funciones u operaciones matemáticas.

Saturación: en el modelo de color HSV, se refiere a la pureza del matiz, representa la distancia al eje de brillo negro-blanco.

Script: es una secuencia de comandos o instrucciones que conforman un programa informático relativamente simple.

Segmentación: dentro del campo del procesamiento de imágenes, se refiere al proceso de dividir una imagen en distintas regiones con atributos similares, logrando hacer una distinción clara entre la información de interés y la información no relevante para el análisis.

Sistema operativo: es software encargado de gestionar los recursos de hardware de un sistema informático.

Software in the loop: paradigma de validación de sistemas en donde la planta y el sistema de control se representan mediante un modelo matemático e interactuar dentro de una simulación.

Terminal de comandos: es una interfaz que le permite al usuario interactuar con un sistema de cómputo de forma explícita a base de un conjunto de instrucciones o comandos bien definidos.

Validación: en el ámbito del la gestión y desarrollo de proyectos de software se refiere a la evaluación del producto para determinar si cumple con las expectativas y requerimientos definidos por el cliente.

Valor: dentro del modelo de color HSV, se refiere al brillo del matiz y representa un desplazamiento vertical en el eje blanco-negro.

Waypoint: es un punto de referencia intermedio que conforma una trayectoria o una ruta para el desplazamiento de algún vehículo.

Abreviaturas y acrónimos

API Application Programming Interface.

HIL Hardware in the Loop.

SIL Software in the Loop.

RCP Rapid Control Prototyping.

RNA Red Neuronal Artificial.

ML Machine Learning.

DL Deep Learning.

ROS Robot Operating System

ÍNDICE GENERAL

1. Introducción	17
1.1. Antecedente históricos	17
1.2. Objetivos	18
1.2.1. Objetivo general	18
1.2.2. Objetivo específicos	18
1.3. Justificación	19
1.4. Planteamiento del problema	21
1.5. Contribuciones	21
1.6. Metodología	22
1.7. Límites y alcances	23
1.7.1. Alcances	23
1.7.2. Límites	24
1.8. Estructura de la tesis	24
2. Estado del Arte	26
3. Marco Teórico	37
3.1. Competencias de Drones Autónomos	37
3.1.1. Autonomous Drone Racing	38
3.1.2. AlphaPilot Challenge	39

ÍNDICE GENERAL

3.1.3. Game of Drones	41
3.2. Robot Operating System (ROS)	42
3.2.1. Concepto	42
3.2.2. Conceptos básicos de ROS	43
3.2.3. Las limitaciones de ROS 1	44
3.2.4. ¿Por qué ROS 2?	45
3.2.5. Diferencias entre ROS 1 y ROS 2	47
3.3. Visión Artificial	49
3.3.1. Concepto	49
3.3.2. Segmentación de color utilizando el modelo HSV	49
3.3.3. OpenCV	50
3.4. Software in The Loop	52
3.4.1. Ardupilot	54
3.5. Gazebo	57
3.5.1. Concepto	57
3.5.2. Historia	58
3.5.3. ¿Por qué Gazebo?	58
4. Resultados	60
4.1. Configuración del Framework de SITL y Ambiente de Simulación	60
4.1.1. Instalación de ROS 2	61
4.1.2. Instalación de OpenCV	65
4.1.3. Instalación de Gazebo	66
4.1.4. Instalación de ArduPilot SITL Simulator	69
4.1.5. Instalación de Pymavlink	77
4.2. Circuito de Vuelo Virtual	78
4.3. Sistema de Visión Artificial	83
4.3.1. Descripción del sistema y forma de trabajo	83
4.3.2. Sintonización de la escala de color	85
4.3.3. Algoritmo de visón artificial	90
4.4. Misión de Vuelo	92
4.4.1. Conexión entre Pymavlink y ArduPilot	93
4.4.2. Configuración de modo de vuelo	95

ÍNDICE GENERAL

4.4.3.	Secuencia de despegue	96
4.4.4.	Seguimiento de trayectoria	98
4.4.5.	Misión de vuelo	106
4.4.6.	Aterrizaje (extra)	111
4.5.	ROS	113
4.5.1.	Visión Artificial	113
4.5.2.	Seguimiento de trayectoria	116
4.5.3.	Integración	118
5.	Conclusión	121
5.1.	Desarrollos futuros	123
	Bibliografía	125

ÍNDICE DE FIGURAS

2.1. Validación del algoritmo de detección de compuertas basado en SSD7 Cabrera-Ponce et al. (2019)	27
2.2. Imagen compuesta del vuelo del dron Mellinger & Kumar (2011)	28
2.3. Dron con raqueta incorporada Mueller et al. (2013)	29
2.4. Imagen compuesta del vuelo del dron del equipo INAOEMoon et al. (2019) .	30
2.5. Resultados de la implementación del equipo TU Delft Moon et al. (2019) .	31
2.6. Propuesta desarrollada por el equipo UNIST, Moon et al. (2017)	32
2.7. Funcionamiento de la red DeepPilot Rojas-Perez & Martinez-Carranza (2020)	34
2.8. Funcionamiento de DeepPilotFoehn et al. (2021)	35
2.9. Funcionamiento del algoritmo de gestión de trayectoria desarrollado por Stevens (2021)	36
3.1. Ejemplo de puertos de comunicación de Ardupilot	56
4.1. Características del sistema en donde se desarrolló el proyecto	61
4.2. Nodos de demostración incluidos en la instalación de ROS 2	65
4.3. Mensaje de validación para la instalación de OpenCV	67
4.4. Ficha técnica de la versión de Gazebo	67
4.5. Proyecto vacío generado al inicializar Gazebo	68

Índice de figuras

4.6. Proyecto de demostración en Gazebo que incluye el plug-in para comunicarse con ROS	69
4.7. Movimiento de la simulación en Gazebo con plugin de comunicación de ROS	70
4.8. Lista de simulaciones de demostración para el uso de Gazebo con ROS 2	71
4.9. Prueba de ejecución del framework de SITL de ArduPilot	74
4.10. Terminal de información del sistema de ArduPilot	75
4.11. Consola de comunicación de ArduPilot	75
4.12. Modelo en Gazebo del dron Iris	77
4.13. Prueba de integración entre Gazebo y el SITL de ArduPilot	78
4.14. Validación de instalación de Pymavlink	79
4.15. Esquema detallado del circuito de vuelo elaborado en simulación.	80
4.16. Modelos de compuerta proveídos por (Rojas-Perez & Martinez-Carranza, 2020)	81
4.17. Modelo de dron iris con cámara frontal provisto por Johnson (2018)	82
4.18. Capturas del ambiente de simulación implementado	83
4.19. Mapa de color en escala HSV (Dey, 2020)	84
4.20. Imágenes utilizadas para la sintonización del rango de color	86
4.21. Detección de compuerta con el primer rango de color	88
4.22. Sintonización del rango de color	89
4.23. Intento de sintonización utilizando la cuarta imagen de referencia como base	91
4.24. Datos de conexión de ArduPilot	94
4.25. Datos obtenidos a partir de la conexión entre ArduPilot y Pymavlink	95
4.26. Ejecución del script para la configuración del modo de vuelo	96
4.27. Comportamiento de la simulación ante el comando de despegue	98
4.28. Respuesta en la altura para el comando de despegue.	99
4.29. Gráfica de velocidad de despegue.	100
4.30. Etapas de la prueba de seguimiento de trayectoria	103
4.31. Recorrido realizado por el dron	104
4.32. Gráficas de desplazamiento para los tres ejes	105
4.33. Comportamiento de la velocidad lineal de vuelo en los tres ejes	106

Índice de figuras

4.34. Gráfica en 3D de la trayectoria seguida por el dron.	107
4.35. Seguimiento de misión de vuelo completa	108
4.36. Recorrido realizado por el dron	109
4.37. Gráficas de desplazamiento para los tres ejes	111
4.38. Comportamiento de la velocidad lineal de vuelo en los tres ejes	112
4.39. Comportamiento de la velocidad lineal de vuelo en los tres ejes	114
4.40. Comportamiento de la velocidad lineal de vuelo en los tres ejes	115
4.41. Comportamiento de la velocidad lineal de vuelo en los tres ejes	116
4.42. Comportamiento de la velocidad lineal de vuelo en los tres ejes	117
4.43. Recorrido realizado por el dron	118
4.44. Recorrido realizado por el dron	119

ÍNDICE DE TABLAS

3.1. Especificaciones de sensores utilizados en el APC (Foehn et al., 2020)	41
3.2. Lista de distribuciones de ROS 2	47
3.3. ROS 1 vs. ROS 2	48
3.4. Parámetros del espacio de color HSV	50
3.5. Módulos de OpenCV	53
4.1. Características técnicas de la laptop Aspire E5-575	60
4.2. Relación entre etapa y su código fuente	93
4.3. Parámetros de la instrucción para seguimiento de waypoints	101
4.4. Tipos de máscara para el comando de seguimiento de trayectoria	102
4.5. Waypoints utilizados en la misión de vuelo	107



CAPÍTULO 1

INTRODUCCIÓN

1.1. Antecedente históricos

En diciembre de 1903, Orville Wright realizó el primer vuelo tripulado en la historia de la humanidad; no tuvo que pasar mucho tiempo para que el concepto de vehículo aéreo no tripulado tuviera un auge dentro de la comunidad científica y militar enfocada a la aviación.

Siendo estrictamente correctos, si se toma en consideración los vehículos capaces de generar sustentación y/o que cuentan con un medio para su control, se puede decir que el primer UAV de la historia, fue diseñado por el inglés Douglas Archibald, al fijar un anemómetro en la cuerda de un cometa, con lo cual fue capaz de medir la velocidad del viento a una altura de aproximadamente 1200 ft. Más tarde, en 1887, Archibald colocó cámaras en otra cometa, con lo cual desarrolló el primer UAV de reconocimiento, en el mundo.

Hablando específicamente de quadrotores, en 1907, Louis Breguet, un pionero francés de la aviación, junto con su hermano Jacques y su profesor Charles Richet, hicieron una demostración del diseño de un giroplano de 4 rotores. Este prototipo contaba con un motor de 30 caballos de fuerza que alimentaba los 4 rotores, cada uno de los cuales tenía 4 propelas y lograba elevarse hasta un máximo de 0.6 m.

Por otro lado, Etienne Oehmichen, un ingeniero francés, fue el primero en experi-

mentar con diseños de aeronaves de ala rotativa. En 1920, construyó y probó 6 diseños, el segundo de ellos tenía 4 motores y 8 propelas; el cuerpo de esta aeronave estaba hecho de tubos de acero y tenía 4 extremidades, en las cuales se alojaban cada uno de sus motores con 2 propelas cada uno. En su momento, este diseño destacaba en su estabilidad y controlabilidad, y para la mitad de 1920 ya había realizado más de mil vuelos de prueba. En 1924 estableció un récord mundial al volar una distancia horizontal de 360 m.

Después, en 1922 el Dr. George de Bothezat e Iván Jerome desarrollaron una aeronave con una estructura en forma de equis y motores de 6 propelas en sus extremidades. Para 1923 habían realizado hasta 100 vuelos de prueba con una altura máxima de 5 m; sin embargo, este diseño era muy complejo y rígido, dificultando su movimiento lateral y suponiendo una carga de trabajo, para alimentar la maquinaria, demasiado alta para el piloto.

Además, en 1956 se desarrolló el Convertawings Model A, el cual fue pensado para formar parte de una línea de quadrotómetros grandes para uso civil y militar. Este prototipo contaba con dos motores, que controlan el giro de dos motores, cada uno, a partir de lo anterior, el control de la aeronave se lograba al variar el empuje proporcionado por los motores.

1.2. Objetivos

1.2.1. Objetivo general

Implementar en simulación un algoritmo de detección de compuertas rectangulares mediante visión artificial junto con una metodología para el seguimiento de trayectoria de vuelo en un cuadricóptero autónomo virtual.

1.2.2. Objetivo específicos

- Diseñar un algoritmo de visión artificial capaz de identificar compuertas rectangulares
- Diseñar un algoritmo de gestión de trayectorias de vuelo para un cuadricóptero autónomo

- Diseñar un ambiente de simulación en 3D de un circuito de vuelo basado en una carrera de cuadricópteros autónomos.
- Implementar un ambiente de Software in The Loop utilizando los algoritmos y el ambiente de simulación diseñados para verificar su comportamiento en conjunto

1.3. Justificación

Lejos de ser un atractivo visual y un espectáculo con fines de entretenimiento, las competencias de drones autónomos representan el estado del arte de la robótica aplicada a vehículos con sistemas de navegación autónoma. Lo anterior se debe a que la robótica siempre se ha enfocado a la automatización de los sistemas; es decir, que los robots sean capaces de realizar tareas o recorridos sin necesidad de intervención humana, para esta última parte, se necesita de algoritmos de percepción y navegación, con los cuales los vehículos puedan ubicarse en el espacio a partir de su sistema de sensores con el que cuentan (tales como tecnología a base de láseres, cámaras estereoscópicas, tecnología ultrasónica, etc.) para que después sea capaz de trazar una trayectoria o seguir una ruta previamente definida.

Esta tecnología ha adquirido una robustez bastante significativa en los últimos años, pues existe una gran cantidad de esfuerzos y colaboraciones dedicadas al desarrollo de los mismos, incluso, se han organizado eventos y competencias con el fin de estimular y potenciar el desarrollo de este tipo de sistemas; tal es el caso de la International Conference on Intelligent Robots and systems (IROS) y AlphaPilot, dos eventos de gran magnitud, creados con el objetivo de tratar, demostrar y fomentar los avances que se tienen en el área.

Por otro lado, la implementación de un sistema robótico autónomo no es una tarea sencilla, y debido a la poca competencia en el mercado también adquiere un costo elevado. Para que un robot sea capaz de percibir el ambiente a su alrededor y desplazarse por el mismo, es necesario implementar un sistema de software capaz de coordinar la adquisición de datos proveídos por los sensores y el conjunto de actuadores que permiten que el sistema se desplace. Muchas de las soluciones desarrolladas para afrontar este desafío son privadas y no sé comparte con el público en general, además, algoritmos como el filtro de Kalman o un control PID son ampliamente utilizados en este tipo de

sistemas, por lo que existe una posibilidad bastante alta de que todas estas soluciones implementen los mismos algoritmos, lo cual conlleva un desperdicio de tiempo y esfuerzo, sin mencionar que la calidad y eficiencia de cada implementación puede variar bastante. Debido a lo anterior, soluciones de código abierto como ROS (Robot Operating System; un framework de comunicaciones para el manejo y coordinación de procesos en sistemas robóticos), pueden representar el inicio de la implementación de un estándar en el área, pues al ser de software libre permiten que toda la comunidad utilice, mejore e inspeccione los algoritmos ya implementados.

Además, la realización de pruebas con el sistema físico, para verificar y validar los algoritmos desarrollados, representa un costo muy alto en la mayoría de sistemas con los que se trabaja en el área, por lo que también es necesario disponer de algún tipo de simulador que permita realizar las pruebas sin necesidad de utilizar el prototipo físico con el que se trabajará. Existen diferentes paradigmas de simulación en los que se puede simular la planta mediante software, tales como Hardware in the loop (HIL) y software in the loop (SIL). Ambos paradigmas representan una solución al problema planteado, proveyendo resultados muy cercanos a la realidad y con una arquitectura flexible, que permite realizar una gran cantidad de pruebas o incluso entrenar algoritmos relacionados con inteligencia artificial o redes neuronales, una vez más, sin depender del sistema físico.

A partir de todo lo anterior, en este trabajo se propone el diseño y la simulación de un algoritmo de visión por computadora para la detección de compuertas rectangulares, similares a aquellas utilizadas en las competencias de drones autónomos, mismas que se utilizan para trazar un circuito de vuelo, para el cual se define un algoritmo de seguimiento de trayectoria para que un dron virtual autónomo sea capaz de completar el circuito definido. El desarrollo e implementación de los algoritmos se realiza dentro de un framework de simulación de SIL, y la gestión y comunicación entre procesos se implementan a partir de una arquitectura diseñada en ROS2, todo lo anterior bajo el paradigma de código abierto, con el fin de aprovechar las ventajas previamente mencionadas y aportar los esquemas de configuración y diseño a la comunidad.

1.4. Planteamiento del problema

Los sistemas aéreos de navegación autónomos han tenido un hito importante en los últimos años, a tal grado que, al día de hoy, existen propuestas bastante sofisticadas de sistemas para la creación de rutas para misiones de vuelo basadas en la detección de guías visuales utilizando visión artificial. Sin embargo, previo a la implementación de cualquier de propuesta, siempre existe una fase de pruebas en donde se valida el desempeño y comportamiento de la propuesta definida, para realizar lo anterior, es necesario disponer de herramientas de simulación que, valga la redundancia, permitan simular con la mayor fidelidad posible a la realidad el sistema que se desea implementar. De tal manera que si es necesario realizar algún tipo de cambio a la propuesta de solución, se realice antes de implementarla en un sistema físico, ahorrando sustento económico y acelerando el desarrollo del proyecto; esto último corresponde a algunas de las ventajas de las simulaciones tipo SIL.

En este trabajo se propone la integración de diversas herramientas de última generación con bajos requerimientos de hardware, instalación multiplataforma y de código abierto para la creación de un ambiente de simulación en 3D con físicas realistas, integrando un modelo de dron comercial y en donde sea posible implementar algoritmos de visión artificial y de seguimiento de trayectoria, de tal forma que puedan interactuar con un controlador de vuelo simulado y guiar al dron a través de un circuito de vuelo.

En todo su conjunto, lo anterior representa las bases de un framework de SIL para validar todo tipo de algoritmos asociados al vuelo autónomo de un dron.

1.5. Contribuciones

Este trabajo cuenta con 3 principales aportaciones al campo de estudio:

- El diseño y la implementación de un ambiente de simulación basado en ROS 2, para la validación de algoritmos de detección y navegación enfocados en misiones de vuelo de drones autónomos
- La documentación detallada del proceso seguido para la creación del ambiente de simulación también representa una contribución importante, pues debido a que

ROS 2 es una tecnología emergente, no existe mucha documentación relacionada con implementaciones de este tipo para el campo de estudio en cuestión.

- Por último, debido a la naturaleza de las tecnologías utilizadas para el proyecto, el código fuente, en conjunto con las instrucciones de creación y utilización del framework, se comparten bajo el paradigma de código abierto, con el objetivo de beneficiar a la comunidad y usuarios de ROS, así como formar parte de la descentralización del conocimiento.

1.6. Metodología

En primera instancia, se realiza una revisión bibliográfica intensiva acerca del estado del arte en cuanto a drones guiados por visión artificial, con el objetivo de visualizar las soluciones ya implementadas y conceptualizar la arquitectura necesaria para el sistema, sus componentes, los algoritmos de visión artificial y sistemas de gestión de trayectoria empleados, así como la configuración necesaria para realizar la integración de todo lo anterior.

Posteriormente, se define el esquema general del proyecto estableciendo el algoritmo de visión artificial a utilizar, la metodología para el seguimiento de la misión de vuelo, el ambiente de simulación, la interfaz de comunicación para la adquisición de datos e imágenes provenientes de la simulación, el modelo de dron a simular y las librerías necesarias para integrar el ambiente de simulación.

Establecido lo anterior, se implementa la arquitectura diseñada para el ambiente de simulación y se realizan vuelos cortos enviando comandos de vuelo básicos al controlador de vuelo simulado, con el objetivo de observar el comportamiento del modelo de dron ante esta serie de comandos, dentro del circuito de vuelo definido. A partir de lo anterior, se extraen imágenes de la trayectoria de vuelo del dron por medio de una cámara simulada a bordo del modelo del dron y se registran los datos obtenidos a partir de los sensores simulados dentro del entorno virtual; se utilizan las imágenes recopiladas y los datos recopilados para el desarrollo de los algoritmos propuestos, en conjunto con el análisis de su desempeño.

Cuando el algoritmo de visión artificial proporciona una identificación adecuada del tipo de compuerta utilizada y el algoritmo de seguimiento de trayectoria permite que el

dron complete una misión de vuelo de prueba, se implementan los algoritmos con base en la arquitectura definida. Se realiza la validación del algoritmo dentro del mismo circuito de vuelo.

A lo largo de la ejecución de la simulación, existe un intercambio de información constante entre el dron virtual y los algoritmos implementados, la simulación envía imágenes obtenidas durante el vuelo del dron y el algoritmo de visión artificial las analiza, de tal forma que es capaz de identificar y aislar las compuertas dentro del fotograma, si es que existe alguna dentro del campo de visión y a la suficiente distancia como para ser detectada; por otro lado, el gestor de trayectoria se encuentra realizando la lectura de los sensores y el estado del piloto automático, con el fin de guiar el vuelo del dron con base en la misión de vuelo establecida, indicando el waypoint pertinente según la ubicación del dron durante su vuelo. En pocas palabras, al final de este trabajo, el dron es capaz de completar el circuito de vuelo y detectar las compuertas que lo delimitan a lo largo de su recorrido.

Por último, se reportan los resultados obtenidos y los problemas enfrentados a lo largo de cada una de las etapas de desarrollo del proyecto, así como las posibles mejoras a futuro para el proyecto en su conjunto.

1.7. Límites y alcances

1.7.1. Alcances

Se implementa un algoritmo de visión por computadora a partir de la detección del color de las compuertas que conforman un circuito de vuelo dentro de un ambiente de simulación personalizado; se realiza la conversión del espacio de color del fotograma captado por la cámara de un dron simulado al modelo de color HSV, y después de un proceso de calibración del rango de color, se utilizan métodos de apertura y cerradura para aislar el objeto de interés para su detección. Por otro lado, se utiliza la librería PymavLink para establecer comunicación con un piloto automático simulado, de tal forma que a partir de una serie de scripts se indican comandos de vuelo que sirven para definir la trayectoria de vuelo de un dron virtual. A partir de lo anterior, se define una arquitectura que integra ambas propuestas.

La arquitectura es evaluada dentro de un entorno de simulación realizado en simu-

lador Gazebo 11, en donde se virtualiza un circuito o pista de obstáculos compuesta por compuertas rectangulares de distintas alturas y color sólido, colocadas en distintas posiciones y orientaciones a lo largo del circuito. Se utiliza ROS2 para coordinar el envío de datos entre la simulación de Gazebo 11 y un nodo independiente de ROS2 para cada algoritmo. Además, se documenta de forma detallada la configuración realizada para la creación del ambiente de simulación, especificando la integración entre Gazebo, ROS, ArduPilot, OpenCV y Pymavlink. Por último, el proyecto en su conjunto se distribuye bajo el paradigma de código abierto.

1.7.2. Límites

A diferencia de las contribuciones y proyectos más populares dentro de la comunidad de las carreras de drones autónomos, en donde se utiliza ROS 1 y Gazebo en su versión 9, en este proyecto se implementa la última versión estable de ROS2, Foxy, y la versión más actual del simulador Gazebo, que al momento de escritura del trabajo es la versión 11. Entonces, algunos plugins tanto de ROS como de Gazebo, que facilitan la interacción entre ambos y la implementación de otras metodologías más sofisticadas, no se encuentren disponibles en estas versiones, lo que significa una limitante para la expansión a futuro del proyecto.

Por otro lado, dentro del ambiente de simulación, no se evalúan condiciones de vuelo poco ventajosas como viento en contra, lluvia o cualquier otra condición climática adversa. Además, la complejidad en el arreglo de compuertas para el circuito es baja, de tal forma que se conoce la composición exacta del circuito previo al vuelo del dron y se utiliza la ubicación ya conocida de las compuertas para la definición de la misión de vuelo; adicionalmente, las compuertas se encuentran siempre de forma perpendicular a la vista de la cámara del dron y se utiliza un solo modelo de dron y tipo de compuerta a lo largo del desarrollo de todo el proyecto.

1.8. Estructura de la tesis

El presente trabajo se divide en 5 grandes partes; la introducción relata el contexto histórico que sirvió como base para llegar al estado del arte que se tiene hoy en día, además, dentro de este mismo capítulo se establecen los objetivos y se clarifican todas

aquellas partes que le dan sustento a la motivación detrás de la creación del proyecto, tales como la justificación, metodología, etc.; la segunda parte corresponde al estado del arte del enfoque tecnológico abarcado, en él se dan ejemplos específicos y prácticos de aplicaciones contemporáneas en donde se hace uso de las tecnologías e implementaciones más sofisticadas y avanzadas que se tienen hasta el momento dentro del campo de aplicación; la parte del marco teórico tiene por objetivo fundamentar y detallar de la forma más explícita posible la implementación desarrollada para el proyecto, desglosando el funcionamiento base de los algoritmos utilizando así como dando un panorama general del acercamiento que se siguió para solucionar el problema planteado, además, brinda una descripción bastante detallada de las tecnologías utilizadas así como sus ventajas, desventajas y prestigio actual; adicionalmente, dentro de apartado de resultado se encuentra descrito a gran detalle el procedimiento desarrollado para la implementación del proyecto, así como el comportamiento, los resultados y los problemas enfrentados en cada uno de las partes, esta sección representa la culminación de la aportación del proyecto y funciona como guía para entender con mayor detalle el contexto bajo el cual se trabajó, así como el progreso gradual observado durante la elaboración del proyecto.

Por último, haciendo la función de cierre, se incluye un capítulo dedicado a la conclusión del trabajo, en donde se realiza una recapitulación del trabajo realizado de tal manera que se completa con los resultados observados y se da una retroalimentación objetiva mencionando las posibles mejoras para continuar el trabajo en un futuro o utilizar las bases de este para la creación de otros proyectos.

Dicho lo anterior, se da por finalizada la primera parte del trabajo, dando preámbulo a la siguiente, el estado del arte.

CAPÍTULO 2

ESTADO DEL ARTE

Las competencias de drones autónomos han adquirido un grado alto de relevancia en la última década, dentro del marco teórico del presente trabajo se describe con profundidad el contexto histórico, así como la motivación y los requerimientos establecidos para dos de las competencias, más significativas, de drones autónomos, el IROS Autonomous Drone Race y el AlphaPilot AI Drone Innovation Challege. En este capítulo se presentan algunas de las soluciones propuestas en estas competencias, al igual que trabajos con enfoques más prácticos o que no se encuentran directamente relacionados con las carreras de drones autónomos.

Dentro de las competencias anteriormente mencionadas, existen dos problemas esenciales a los que se enfrentan los equipos que participan en estos retos, la detección de objetos y la gestión de trayectoria de vuelo a partir de la detección realizada. Los circuitos que tiene que completar los drones están compuestos por compuertas de distintas formas y tamaños, y en algunos casos, se adicionan obstáculos dinámicos, los vehículos desarrollados por los participantes tienen que ser capaces de detectar estos objetos haciendo uso exclusivo de los sensores con los que están equipados (cámaras, sensores ultrasónicos, tecnología láser, etc.).

Con base en lo anterior, Cabrera-Ponce et al. (2019) desarrollaron un algoritmo para la detección de compuertas en tiempo real basado en aprendizaje profundo. Su implementación se basó en una arquitectura de red neuronal convolucional (RNC) con una

arquitectura base de Single Shot Detector de 7 capas, Ferrari (2018). La arquitectura base tiene un diseño optimizado para la detección de objetos, permitiendo un tiempo de entrenamiento reducido y una velocidad de detección alta; esta se modificó de tal forma que se eliminaron las últimas dos capas de convoluciones, haciendo posible una detección mucho más rápida que la propuesta base y disminuyendo la complejidad de la red. El entrenamiento de la red se realizó con un total de 3418 imágenes obtenidas a partir de un entorno simulado y entornos reales. Además, para observar el desempeño de su implementación compararon su arquitectura con otras propuestas, SSD7, SSD300 y SmallerVGG, en simulaciones y ambientes de exteriores e interiores. Los resultados muestran que su propuesta logra un tiempo de detección promedio más bajo y porcentaje de confianza más alto que las otras arquitecturas.

La figura 2.1 muestra algunos de los resultados obtenidos por Cabrera-Ponce et al. (2019), en el conjunto de imágenes se logra apreciar la influencia que tienen los datos de entrenamiento utilizados sobre el desempeño obtenido; el primer tercio de imágenes, las del extremo izquierdo, se obtienen a partir de un entrenamiento basado exclusivamente en imágenes de simulación; los resultados de la parte central fueron obtenidos entrenando la red con imágenes de compuertas reales, y por último, el último conjunto de resultados se obtuvo combinando imágenes de simulación e imágenes reales para el entrenamiento. Como es posible apreciar, la RNC tuvo un mejor entrenamiento al combinar imágenes capturadas en simulación e imágenes de un ambiente real.

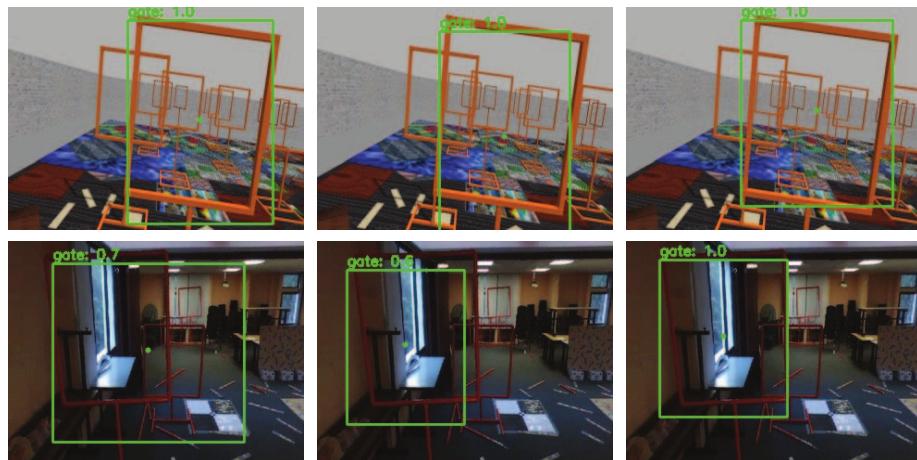


Figura 2.1: Validación del algoritmo de detección de compuertas basado en SSD7 Cabrera-Ponce et al. (2019)

Por otro lado, Mellinger & Kumar (2011) presentaron un diseño de control y generación de trayectoria de vuelo en ambientes de interiores para un quadrotor. Su implementación es capaz de generar una trayectoria óptima y ángulos para la guiñada del vehículo, en tiempo real, a partir de matrices de rotación para el marco de referencia del vehículo y una secuencia de posiciones en tres dimensiones. La propuesta fue diseñada con el objetivo de que el quadrotor sea capaz de navegar de forma segura a través de corredores angostos, manteniéndose en los límites de velocidad y aceleración. Además, implementaron un control no lineal que asegura el seguimiento de las trayectorias generadas; las propuestas se pusieron a prueba con un prototipo físico que se hizo volar a través de un circuito construido por aros, los cuales indicaban la trayectoria que el quadrotor debía de seguir. La figura 2.2 muestra el recorrido realizado por el dron durante una de las pruebas.



Figura 2.2: Imagen compuesta del vuelo del dron Mellinger & Kumar (2011)

Adicionalmente, Mueller et al.(2013)Mueller et al. (2013) diseñaron un algoritmo de bajo consumo computacional para la generación de trayectorias de intersección para la ruta de vuelo de un quadrotor. La implementación tuvo como propósito que el quadrotor fuera capaz de interceptar una pelota en vuelo, con una raqueta montada en su chasis, la figura 2.3 muestra el dron utilizado. El algoritmo de generación de trayectoria se usó en un sistema de control predictivo, en donde miles de trayectorias eran generadas y evaluadas por el controlador, y después, la trayectoria más óptima era seleccionada por el algoritmo. Se destaca el bajo coste computacional pues se utilizó el hardware de una laptop estándar para evaluar cerca de un millón de trayectorias por segundo.



Figura 2.3: Dron con raqueta incorporada Mueller et al. (2013)

Las propuestas anteriores representan ejemplos de soluciones individuales para cada uno de los problemas mencionados. Sin embargo, existen implementaciones que solucionan ambos problemas en un solo trabajo, y corresponden a aquellas que fueron desarrolladas como propuestas para participar en las competencias.

El trabajo realizado por Moon et al.(2019) Moon et al. (2019) compila una serie de propuestas destacadas, y describe con detalle los algoritmos de visión artificial, odometría, control de vuelo, etc., desarrollados para el IROS 2017 por los equipos más sobresalientes de la competencia.

Se presentan 5 propuestas distintas. Iniciando por el equipo ganador de la competencia, los participantes del Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE); implementaron un control PID para la altura, curso y ángulo de deslizamiento del dron, además, obtuvieron la localización espacial del dron y su orientación a partir de un algoritmo de deep learning basado en ORB-SLAM. Su algoritmo de odometría asume que el suelo de la pista es plano, por lo que al conocer la altura y ángulo de la cámara de vuelo, les fue posible generar una trayectoria de vuelo adecuada para el cruce de las compuertas. La figura 2.4 presenta la participación del equipo INAOE, en donde se logra observar la implementación de su algoritmo de detección, su sistema de odometría y su propuesta para la generación de trayectorias de vuelo.

Por otro lado, el equipo de la Universidad de Zúrich (UZH) propuso una solución basada en la elaboración de un modelo 3D del circuito de vuelo, el cual utilizó para definir una serie de waypoint para la navegación del dron, a cada waypoint se le asoció un vector de velocidad; lo anterior, en conjunto con un sistema de odometría visual, permitieron que

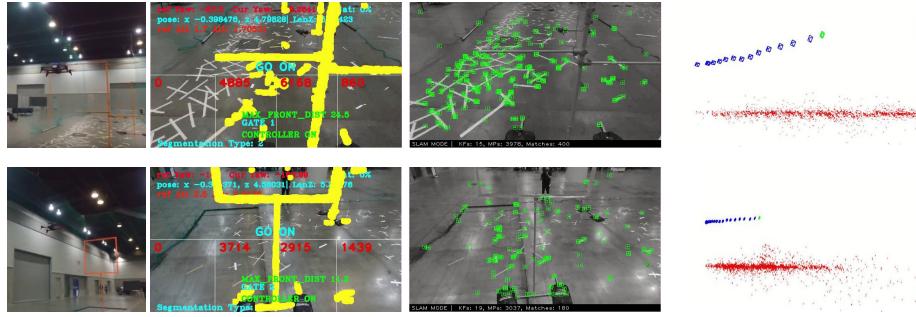


Figura 2.4: Imagen compuesta del vuelo del dron del equipo INAOEMoon et al. (2019)

el dron del equipo de UZH navegará de forma autónoma a través del circuito. El principal reto para esta implementación fue la alineación de la pista con el marco de referencia del dron, para solucionar lo anterior, utilizaron un sensor de profundidad junto con un mapa de referencia, de tal forma que minimizan la distancia entre la nube de puntos de la pista y el conjunto de puntos proveídos por el sensor.

El tercer lugar del IROS 2017, le perteneció a la Universidad Técnica de Delft (TU Delft). Este equipo buscó enfocar su propuesta en drones de tamaño compacto (< 50 cm), teniendo como objetivo un vuelo rápido, ágil y de bajo costo computacional. Lo anterior contempla ciertas limitaciones inherentes en cuanto a la cantidad de sensores integrados en el vehículo y la gama de la computadora de vuelo que se puede utilizar. Debido a lo anterior, el equipo TU Delft optó por utilizar una máquina de estados para la gestión de la trayectoria de vuelo, en vez de algoritmos complejos de SLAM u odometría visual. De forma más detallada, el algoritmo propuesto fue una máquina de estados de alto nivel en donde cada estado está asociado a un comportamiento específico definido para cada parte del circuito; esto representa una ventaja, pues la máquina de estados es muy eficiente, computacionalmente hablando. Por otro lado, la máquina de estados necesita la posición y orientación del dron con respecto a la compuerta que está a punto de cruzar; la detección de compuerta se realizó utilizando un algoritmo basado en la detección del color de estas; es decir, no utilizaron RNC por el alto consumo de recursos, y en su lugar hicieron uso de funciones básicas de visión artificial para realizar una segmentación del color de la compuerta, y de esta forma realizar su detección. A partir de lo anterior, se detectó las esquinas de las compuertas, y utilizando la geometría conocida de las mismas, fue posible determinar la posición y orientación con respecto a la compuerta. Adicionalmente, el equipo también implementó una detección basada en un histograma de colores,

el cual detecta los vértices que considera más verticales para detectar la compuerta más cercana, y a diferencia del algoritmo anterior, esta propuesta esta pensada para realizar la identificación cuando no toda la compuerta es visible, por lo que ambos algoritmos se complementan para realizar la detección de las compuertas.

En la figura 2.5 se muestran algunas capturas realizadas durante la participación del equipo; en ellas se aprecia la precisión obtenida para la detección de compuertas. En la primera columna de imágenes se observa el algoritmo de detección por color, mientras que en la segunda columna se aprecia las barras verticales producto del algoritmo con el histograma de colores. Por último, la imagen de la tercera columna muestra el dron del equipo Tu Delft cruzando a través de una compuerta de forma autónoma.



Figura 2.5: Resultados de la implementación del equipo TU Delft Moon et al. (2019)

Después, el equipo del Instituto Avanzado de Ciencia y Tecnología de Corea (KAIST) propuso una combinación de una arquitectura de deep learning para la detección de compuertas y un algoritmo de guía por línea de vista (Line Of Sight Guidance) para la generación de la trayectoria de vuelo. El equipo KAIST implementó una red neuronal convolucional de 7 capas convolucionales, basada en la arquitectura de ADRNet, para el procesamiento de imágenes y la detección de compuertas en tiempo real. Esta arquitectura logró la inferencia a una velocidad de 28.95 fps en una computadora de placa única NVIDIA TX2. Por otro lado, KAIST logró la generación de maniobras precisas para el pase a través de compuerta con un algoritmo de LOS Guidance. Este algoritmo es muy utilizado en aeronaves de ala fija, y fue modificado ligeramente para que se adaptara a la dinámica de un quadrotor. La propuesta desarrollada por KAIST representa un buen acercamiento para navegar en situaciones de alta incertidumbre, pues no depende del mapa del circuito; sin embargo, lo anterior es ineficiente, computacionalmente hablando, en circuitos en donde se cuenta con los detalles y composición del recorrido de vuelo a

priori.

Por último, en cuanto al IROS 2017, el equipo de la Universidad Nacional de Ulsan (UNIST) implementó una red neuronal profunda para la detección de las compuertas del circuito, y a partir del procesamiento de las imágenes, lograron generar controles de vuelo para el desplazamiento horizontal, vertical y las acciones rotacionales. Los comandos de vuelo son generados en forma de un mensaje de tipo MAVLink para que la computadora de vuelo los pueda interpretar, controlando el dron de forma directa. La arquitectura de la red neuronal está basada en la red Google Inception de Xia et al. (2017), la cual representa el estado del arte de las arquitecturas para detección y clasificación. Bajo esta implementación, el dron es capaz de volar a través de las compuertas con dos pasos; el dron se encuentra en una posición inicial y su cámara tiene que tener en su campo de visión a la compuerta a atravesar; se establece una línea recta con respecto al centro de la compuerta y el dron vuela tomando esa trayectoria como referencia. Una vez alineado con la recta, el dron vuela a través del centro de la compuerta. La figura 2.6 muestra un diagrama general de los pasos descritos anteriormente para la implementación del equipo UNIST.

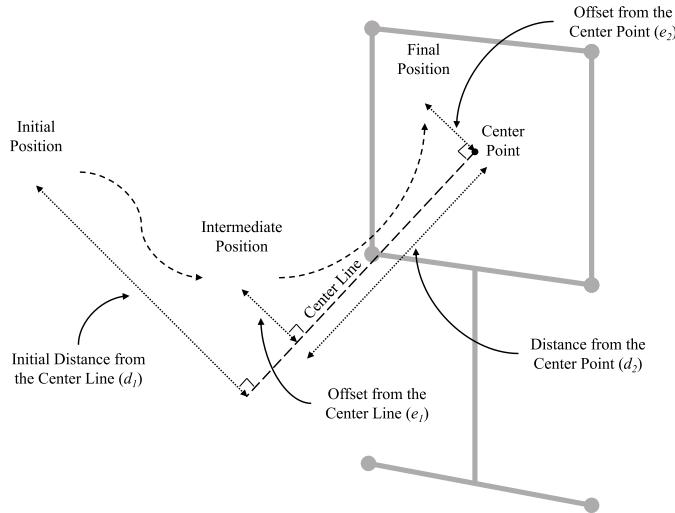


Figura 2.6: Propuesta desarrollada por el equipo UNIST, Moon et al. (2017)

A partir de lo anterior, es evidente el impacto y la importancia que han adquirido las redes neuronales profundas dentro de las competencias de drones autónomos. Otro ejemplo de este tipo de implementación fue presentado por Kaufmann et al. (2018); desarolla-

ron un sistema de visión artificial y seguimiento de trayectoria, pensado para ambiente dinámicos, en donde se requiere de un vuelo ágil y una estimación de estados adecuada, que permita una rápida y correcta definición de la trayectoria de vuelo para el dron. Para lograr lo anterior, implementaron una red neuronal convolucional basada en la RedNet de Loquercio et al. (2018), acoplada a un algoritmo de planificación de trayectoria; la red neuronal recibe imágenes directamente de la cámara de vuelo del dron y realiza un mapeo de tal forma que genera un waypoint y una referencia de velocidad deseada, lo cual es utilizado posteriormente por un algoritmo planificador para generar el segmento de trayectoria más corto y los comandos para los motores, de tal forma que el dron pueda alcanzar su destino. Esta implementación no requirió del conocimiento previo del circuito de vuelo, pues todos los cálculos son realizados en tiempo real durante el vuelo. Esta propuesta se implementó en simulación y en un ambiente físico en donde algunas de las compuertas del circuito cambiaban de posición durante el vuelo, además, su desempeño se comparó con el obtenido con un vuelo realizado por pilotos con distinta experiencia de vuelo. Los resultados muestran que este sistema tuvo un desempeño más bajo en comparación con la habilidad y destreza de los pilotos humanos; sin embargo, este sistema ejemplifica una buena implementación de un algoritmo de percepción robusto en conjunto con una arquitectura moderna de machine learning y algoritmos de velocidad y estabilidad de vuelo.

Por otro lado, Rojas y Martínez (2020) Rojas-Perez & Martinez-Carranza (2020) presentaron una propuesta bastante llamativa. Propusieron una arquitectura de red neuronal convolucional para procesar imágenes obtenidas a partir de una cámara montada en el dron y generar comandos de vuelo que permiten que el dron centre su trayectoria y pase a través de compuertas de un circuito de vuelo basado en el IROS. Utilizaron una ambiente de simulación elaborada en Gazebo para la implementación de su algoritmo de visión artificial y la obtención de datos para su entrenamiento. Como parte de su contribución, propusieron una nueva arquitectura de red neuronal profunda, que toma como entrada un mosaico de imágenes compuesto por un arreglo de tomas capturadas por la cámara del dron durante su vuelo, lo anterior permite tener cierta temporalidad del comportamiento del dron y deducir la acción de vuelo más adecuada para cruzar a través de una compuerta. Además, utilizaron ROS para la gestión de los procesos en la computadora de vuelo y se hace la comparación con la arquitectura de otras redes neuronales profun-

das. La figura 2.7 muestra un diagrama general de la implementación de Rojas-Perez & Martinez-Carranza (2020) en 4 pasos; primero se adquiere la imagen captada por la cámara del dron; después, se utiliza la imagen captada para insertarla en un mosaico junto con otras imágenes captadas durante el recorrido del dron; el mosaico generado es ingresado a la red neuronal artificial; por último, se genera el comando de vuelo y se hace pasar a través de un filtro para disminuir el ruido presente durante el procesamiento de las imágenes.

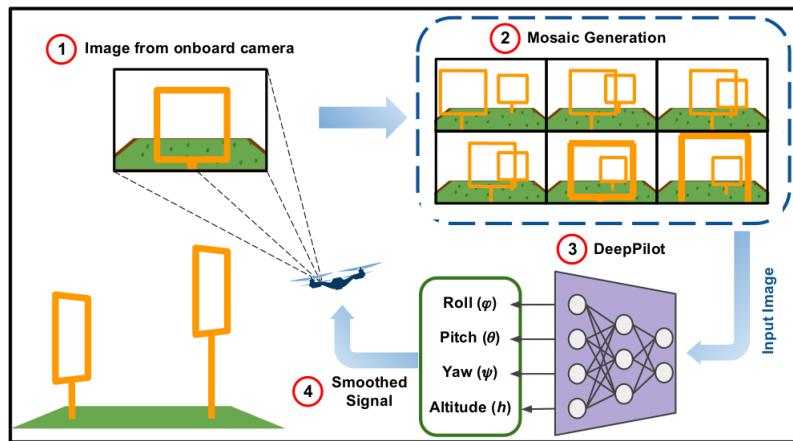


Figura 2.7: Funcionamiento de la red DeepPilot Rojas-Perez & Martinez-Carranza (2020)

En cuanto a soluciones particulares para algoritmos de seguimiento de trayectoria, Foehn et al. (2021) propone una metodología para el cálculo de trayectoria de vuelo en un tiempo óptimo durante el vuelo de un cuadricóptero autónomo, que permite explotar completamente la potencia ofrecida por los actuadores. Además, se plantea una formulación que optimiza la trayectoria de vuelo a lo largo del seguimiento de la trayectoria, se compara la propuesta con otras propuestas y se valida el algoritmo implementado en un prototipo físico. Por último, se pone a prueba la implementación desarrollada enfrentándola a un piloto de drones experimentado, y los resultados argumentan que la solución implementada logra obtener un desempeño superior al del piloto humano.

Finalmente, mencionando un ejemplo que va más allá de las competencias de drones autónomos, Stevens (2021) propone el diseño de un quadrotor ligero y dimensiones reducidas, equipado con una serie de sensores y subsistemas que hacen posible su vuelo autónomo en ambientes con alta densidad de vegetación y obstáculos. La contribución del trabajo se enfoca en el desarrollo del proyecto a partir de requerimientos de efectivi-



Figura 2.8: Funcionamiento de DeepPilotFoehn et al. (2021)

dad y seguridad, definiendo un diseño de dron que utiliza componentes comerciales de bajo costo. La estimación de estados y la guía de vuelo, generada a través de un sistema de visión artificial, se obtienen a partir de la computadora de vuelo a bordo del vehículo, sin ningún tipo de cálculo previo al vuelo. Además, un sistema de flujo óptico permite sensar la velocidad para la estimación de posición, y los efectos derivados por el derrape se compensan utilizando un GPS. Los resultados de la implementación demuestran que el sistema propuesto y los algoritmos desarrollados, son capaces de llevar a cabo una evasión dinámica de obstáculos durante el vuelo. La implementación de esta propuesta se realizó tanto en simulación como en físico. La figura 2.9 muestra el desempeño en simulación del algoritmo de generación de trayectoria propuesto en este trabajo, se realizaron varias pruebas con las trayectorias de vuelo generadas, las trayectorias exitosas son las curvas de color rojo y verde oscuro, mientras que el resto de trayectorias provocaron que el dron chocara con un obstáculo, y se encuentran tachadas en uno de sus extremos.

A manera de resumen, se analizaron algunos de los algoritmos e implementaciones más sofisticadas hechas en el campo de estudio, al momento de la elaboración de este trabajo. Algunos de los trabajos se enfocaban únicamente en el problema de la detección de compuertas mientras que otros implementaban soluciones específicas para la generación de trayectorias de vuelo, y en otros casos, como los algoritmos desarrollados para las competencias, las implementaciones buscaban superar ambas problemáticas. Dicho lo anterior, en la descripción de algunos de los trabajos, no se mencionó de forma explícita; sin embargo, la gran mayoría de las propuestas mencionadas pasaron por una etapa de validación llevada a cabo en algún ambiente de simulación, de hecho, algunas de las competencias enfocadas a esta área, evalúan el desempeño de los sistemas propuestos dentro de la simulación como primera etapa para seleccionar a los equipos que participantes, esto se menciona con mayor detalle dentro del marco teórico del presente documento.

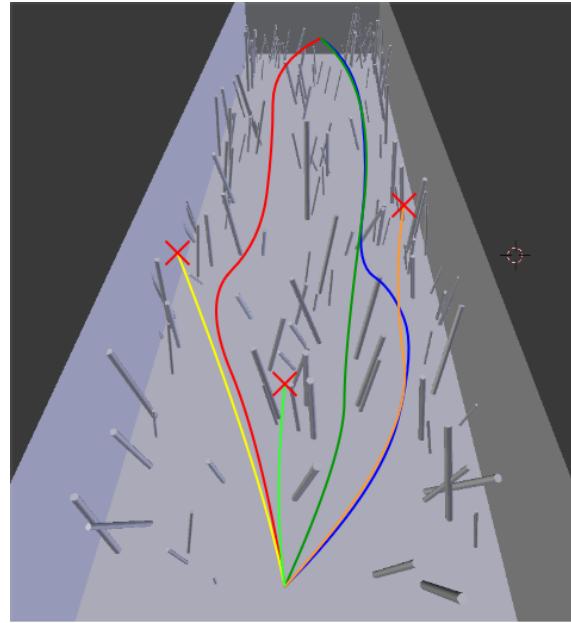


Figura 2.9: Funcionamiento del algoritmo de gestión de trayectoria desarrollado por Stevens (2021)

Dicho lo anterior, dentro de los ambientes de simulación utilizados por los equipos participantes para entrenar y validar sus algoritmos, existen algunas soluciones bastante específicas enfocadas exclusivamente al vuelo de drones autónomos, tales como Flight-Goggle desarrollado por Guerra et al. (2019), Flightmare creado por Song et al. (2020) o AirSim (Shah et al., 2017). Sin embargo, los ambientes anteriores requieren un hardware con poder computacional bastante alto, en especial exigen una tarjeta gráfica dedicada; además, pese a que FlightGoggle y Flightmare ofrecen compatibilidad con ROS, solo lo hacen con ROS 1. En este trabajo se utiliza Gazebo en conjunto con ROS 2 para la implementación de los algoritmos de desarrollados, por lo que se propone un ambiente de simulación abierto que utiliza tecnologías de última generación, con requerimientos de hardware básicos.

CAPÍTULO 3

MARCO TEÓRICO

3.1. Competencias de Drones Autónomos

Las carreras de drones se han convertido en un deporte bastante popular en los últimos años. Resulta increíble pensar que, haciendo uso de única y exclusivamente una cámara de vuelo, los pilotos son capaces de abstraer la información necesaria del ambiente para ejecutar maniobras de vuelo con alta precisión y agilidad.

A partir de lo anterior, la comunidad científica, en especial aquella dedicada al campo de la robótica, se ha visto bastante interesada en sustituir al piloto humano por meras unidades de cómputo y componentes electrónicos; es decir, hoy en día existe la tendencia a automatizar el vuelo de estos vehículos aéreos no tripulados, de tal manera que, a partir de computadoras de placa única, sensores y algoritmos sofisticados de visión artificial, odometría y gestión y control de trayectoria de vuelo, se pueda obtener el mismo desempeño de vuelo que el otorgado por un piloto humano, e incluso, en algún punto, superarlo de manera significativa.

Sumando a lo ya expresado, se han creado una serie de instituciones y eventos con el fin de financiar, potenciar y motivar el desarrollo tecnológico en este campo emergente, dando lugar a lo que se conoce como *carreras de drones autónomos*. Dentro de los eventos o competencias más significativas se encuentra el *Autonomous Drone Racing (ADR)*(Moon et al., 2017), llevado a cabo cada año en la Conferencia Internacional de Sistemas y Robots

Inteligentes (IROS, por sus siglas en inglés), el *AlphaPilot Challenge (APC)*(Foehn et al., 2020), organizado por Lockheed Martin en colaboración con Nvidia y la Liga de Carrera de Drones (DRL); y *Game of Drones (GOD)*(Madaan et al., 2020), gestionada por Microsoft para la Conferencia Anual de Sistemas de Procesamiento de Información Neuronal (NeurIPS) de 2019.

Los eventos anteriores representan un punto de encuentro a nivel internacional que ha permitido dirigir los esfuerzo e intelectos alrededor del mundo, a la propuesta de soluciones, ya sea de forma parcial o general, para el dilema ya expresado; y de hecho, es ahí en donde se ha presentado el estado del arte de este enfoque, pues se busca poner a prueba las implementaciones propuestas por los participantes en circuitos y retos con distintas características y composición. En las siguientes subsecciones se describe con más detalle las características, requisitos y relevancia de cada una de las competencias mencionadas.

3.1.1. Autonomous Drone Racing

A grandes rasgos, el ADR es una competencia que busca promover soluciones para vuelos autónomos ágiles en ambientes angostos de interiores. En el desafío se combinan técnicas y enfoques que buscan optimizar distintos parámetros de desempeño, como la generación de trayectoria de vuelo, el tiempo de recorrido de los circuitos, esquemas de control, detección de obstáculos, localización y mapeo, entre otros.

El ADR debutó como evento en la edición de 2016 del IROS, en Daejeon, Corea. A partir de entonces siguió teniendo presencia en 3 ediciones más del IROS; en 2017 con sede en Vancouver, Canadá, en 2018 en Madrid, España y en 2019 Macao, China. Cabe recalcar que el IROS per se sigue llevando a cabo, sin embargo, la última ADR tuvo lugar en la edición 2019 de este evento, posiblemente por las restricciones derivadas en 2020 por la pandemia provocada por el virus SARS-CoV-2; además, la edición 2021 del IROS, se llevó a cabo de forma virtual.

En general, en cada edición se propuso una única pista, dentro de una zona techada, con 5 pruebas de vuelo para los equipos participantes; velocidad de vuelo en línea recta a través de una serie de compuertas incompleta, vuelo en curva cerrada, recorrido de un circuito en zigzag, recorrido de un circuito en espiral y a través de compuertas cerradas y vuelo por un corredor con obstáculos dinámicos.

En la edición de 2016, las compuertas fueron identificadas con un número embebido en un código QR, para facilitar su localización. El tamaño de los drones se limitó a un volumen máximo de 1 m x 1 m x 1 m; a los equipos se les compartieron detalles estructurales sobre el circuito antes de la competencia, por lo que les era posible generar mapas que les pudieran auxiliar en la navegación del dron. Además, se les permitió el uso de cualquier tipo de sensor, siempre y cuando este estuviera montado en el chasis del vehículo; se utilizaron distintos tipos de sensores para su participación, incluyendo lidares, láseres, radares y sensores ultrasónicos.

En cada edición del ADR, las compuertas utilizadas para delimitar el circuito han conservado un característico color naranja; en cada evento los circuitos cumplieron con los requerimientos y pruebas mencionadas anteriormente, excepto en la edición 2019 en donde el circuito estuvo compuesto por dos grupos de compuertas LED, alfombras con patrones y luces controladas; además, el tamaño de este circuito fue reducido para producir una pista mucho más angosta, con el objetivo de incrementar la dificultad en el desafío (Rojas-Perez & Martinez-Carranza, 2021).

Para cada equipo, la competencia comenzaba con el despegue del dron de forma manual, este era posicionado en un punto de inicio y en cuanto se diera la señal, los equipos cedían el control del dron a su sistema de piloto automático; es decir, se tenía que suspenderse toda clase de interacción humana con el sistema de vuelo del dron, y permitir que navevara de forma autónoma hasta completar el circuito.

3.1.2. AlphaPilot Challenge

Como se mencionó, el APC es otra competencia enfocada en las carreras de drones autónomos, fue presentado como un reto de innovación con un gran premio de \$1 millón de dólares para el equipo ganador; la iniciativa fue creada y lanzada al público por Lockheed Martin en conjunto con La Liga de Carrera de drones, en 2019. El objetivo del desafío fue desarrollar un dron completamente autónomo que pudiera navegar por un circuito de vuelo utilizando visión por computadora; a diferencia de otras competencias, el APC no solo buscaba poner a prueba la capacidad de navegación de los sistemas, sino que, se buscaba explotar por completo los sistemas de vuelo, de tal forma que se buscó evaluar también la velocidad de vuelo y agilidad de las maniobras en una pista más grande y compleja en comparación con la de ADR.

Entonces, en el APC se buscó implementar soluciones más complejas que permitieran percibir el ambiente del dron por medio sistemas de visión artificial y que los sistemas de control de vuelo fueran capaces de llevar al límite la velocidad de navegación de este; el objetivo era claro, se buscaba ampliar el estado del arte y desarrollar implementaciones que pudieran competir con el desempeño de los mejores pilotos humanos.

Más de 400 equipos participaron en la etapa de selección de esta primera edición del APC, y solamente los mejores 9 equipos clasificaron para poder participar en la competencia. La segunda fase del reto consistió en 3 carreras de clasificación, de donde se seleccionaron a 6 equipos finalistas. La etapa final de la competencia se disputó con un único circuito, donde los equipos compitieron por llevarse el gran premio de \$1 millón de dólares. Los ganadores de cada etapa y carrera de selección fueron filtrados a partir del tiempo que les tomó completar los circuitos; cada participante contó con 3 intentos para completar los circuitos tan rápido como les fuese posible y sin ningún otro competidor o adversario en la pista.

En cada carrera, los drones empezaban en un podio desde donde tenían que despegar y navegar por una secuencia de compuertas con formar distintas y en un orden predefinido. A diferencia del ADR, a los equipos del APC solo se les informó de la estructura de la pista momentos antes de competir; es decir, no podían hacer uso de un mapa detallado para definir la trayectoria de vuelo que tenía que seguir su dron, sino que, tenía que implementar soluciones que se pudieran adaptar en tiempo real a la posición de las compuertas. Se había estimado una longitud de aproximadamente 300 m para cada pista, sin embargo, debido a dificultades técnicas, la pista de mayor longitud fue la de la carrera final, con una longitud aproximada de 74 m(Foehn et al., 2020).

Por último, las características del dron utilizado fueron estandarizadas, por lo que a cada equipo se les otorgó el mismo modelo de dron. Además, a todos los equipos se les facilitó una computadora de vuelo *NVIDIA Jetson Xavier*, para la interfaz con los sensores y actuadores de navegación y también fungió como la unidad de procesamiento para llevar a cabo el vuelo autónomo. El arreglo de sensores a bordo de dron se conformó por un par de cámaras estero con vista frontal a 30°, una unidad de medición inercial (IMU), un telémetro láser (LRF); la tabla 1 muestra las especificaciones técnicas de los sensores utilizados. Por último, los drones también estaban equipados con un controlador de vuelo encargado de controlar el empuje y la velocidad angular.

Sensor	Modelo	Frecuencia	Detalles
Cámara	Leopard Imaging IMX 264	60Hz	resolución de 1200 x 720
IMU	Bosch BM1088	430Hz	rango: ± 24 g, ± 34.5 rad/s resolución: $7e^{-4}$ g, $1e^{-3}$ rads/s
LRF	Garmin LIDAR-Lite v3	120Hz	rango: 1 – 40 m resolución: 0.01 m

Tabla 3.1: Especificaciones de sensores utilizados en el APC (Foehn et al., 2020)

3.1.3. Game of Drones

En la tercera edición de la Conferencia Anual de Sistemas de Procesamiento de Información Neuronal (NeurIPS), en 2019, el equipo de desarrollo de AirSim(Foehn et al., 2020) en conjunto con la Universidad de Stanford y la Universidad de Zúrich buscaron fomentar el avance de las tecnologías utilizadas den las carreras de drones, gestionando la competencia Game of Drones.

De manera similar a las competencias anteriores, el GOD buscó explotar el potencial de los algoritmos de machine learning y visión por computadora, junto con los avances en cuestión de técnicas de planificación de trayectoria, control y estimación de estado de quadrotores; sin embargo, a diferencia de los otros eventos, el GOD se basó completamente en la utilización de un simulador de vuelo con gráficas foto-realistas para la implementación de las propuestas desarrolladas por los equipos.

El simulador utilizado para la competencia fue Microsoft AirSim(Shah et al., 2017), el cual fue desarrollado con el objetivo de hacer más accesible el ámbito de las carreras de drones para ingenieros e investigadores, que tiene un conocimiento bastante amplio sobre algoritmos de machine learning e inteligencia artificial, pero que quizás no estén tan familiarizados con el hardware utilizado por estos sistemas de robótica. AirSim se define como un ambiente de simulación para multi-rotores, que integra un motor de físicas de consumo reducido, controlador de vuelo, sensores iniciales, y gracias al uso del motor gráfico Unreal Engine (UE), cuenta con un ambiente con gráficos foto-realistas. Además, también ofrece una API (Application Programming Interface) que permite interactuar y comunicarse con los algoritmos de machine learning, y también provee datos sobre el progreso del recorrido, el desempeño del dron y la habilidad de imponer reglas o normas para la carrera, asociadas a infracciones por colisiones y descalificaciones dentro de la competencia.

La competencia se enfocó en control y planificación de trayectoria, visión artificial y evasión de obstáculos (otro dron oponente). Lo anterior se llevó a cabo en tres niveles con base en el enfoque:

Nivel 1 - Planificación de trayectoria: cada circuito estuvo limitado a dos drones a la vez, en donde uno era el perteneciente al equipo participante y el otro era un dron oponente implementado por el staff de Microsoft. El objetivo fue atravesar todas las compuertas en el menor tiempo posible, evitando colisionar con el dron oponente. La posición de las compuertas y de ambos drones fue proveída a través de la API del simulador. El dron oponente contaba con un algoritmo de trayectoria óptima y volaba con una serie de waypoints generados al azar para cruzar por la sección transversal de cada compuerta.

Nivel 2 - Percepción: en esta modalidad, la posición de las compuertas contenía ruido, no había dron oponente y la siguiente compuerta a cruzar no siempre se encontraba a la vista; la posición proveída por la API ayudaría a dirigir al dron en la dirección correcta, sin embargo, el vehículo tenía que valerse de su algoritmo de visión artificial para completar el circuito de manera satisfactoria.

Nivel 3 - Percepción y planificación de trayectoria: esta modalidad resultó de la combinación de los dos niveles anteriores. A los participantes se les preveía con datos sobre la posición de las compuertas y había un dron adversario; el objetivo era completar el circuito evitando cualquier colisión con el adversario.

Por último, la competencia consistió de dos etapas, una de clasificación y una ronda para los finalistas. Se registraron 117 participantes, pero solamente 16 calificaron para la competencia.

3.2. Robot Operating System (ROS)

3.2.1. Concepto

De acuerdo con su sitio oficial(Open-Robotics, 2021c), ROS (del inglés, Robot Operating System) es un conjunto de herramientas y librerías de software para robótica desarrolladas por Open Robotics(Open-Robotics, 2021b) bajo el paradigma de software libre u open-source. Este entorno de trabajo destaca por contener algoritmos de última generación y herramientas de desarrollo avanzadas, que permiten la creación, implementación y reutilización de código para todo tipo de proyectos de robótica.

ROS 1, la primera versión del entorno de trabajo, surgió en 2007 como un ambiente de desarrollo para el PR2 robot, un robot de servicio diseñado para trabajar con personas y creado por la empresa The Willow Garage. Sin embargo, los creadores de ROS buscaban que el entorno de trabajo no se viera limitado a un solo modelo de robot, sino que, pudiera ofrecer herramientas de software para más tipos y modelos de robots, por lo que ROS adquirió varias capas de abstracción mediante la implementación de interfaces para el manejo de mensajes, lo que dio lugar a que el software desarrollado mediante ROS pudiera ser reutilizado en más robots.

Algunas características que destacan en esta etapa temprana de ROS son:

- Gestión de un solo robot
- Sin requerimientos de aplicación en tiempo real
- Excelente conectividad a la red
- Usado principalmente en el ámbito académico y de investigación

Hoy en día, ROS es utilizado en una amplia gama de robots, desde robots con ruedas y con forma humanoide, hasta brazos industriales, vehículos aéreos y mucho más. Sin embargo, ha pasado bastante tiempo desde el lanzamiento de la primera versión de ROS, y las necesidades y estándares de la industria han cambiado al igual que el paradigma y la filosofía detrás del desarrollo de ROS. Dicho lo anterior, es evidente que ROS ha adquirido una alta relevancia desde su creación; sin embargo, existen muchas limitaciones asociadas a la manera en que ROS fue diseñado.

3.2.2. Conceptos básicos de ROS

Paquetes: son la unidad principal para organizar software en ROS. Un paquete puede contener procesos (nodos), liberarías, conjuntos de datos, archivos de configuración o cualquier otro tipo de archivo que pertenezca a un conjunto funcional. Los paquetes representan la unidad atómica en ROS.

Tipos de mensajes: descripción de mensajes, definen la estructura de los datos para los mensajes manejados por ROS.

Tipos de servicios: descripción de servicios, define la estructura solicitada o enviada para los datos utilizados en los servicios de ROS.

Nodos: son procesos que llevan a cabo cálculos. ROS está diseñado para ser modular. Un sistema robótico generalmente está compuesto por múltiples nodos encargos de distintas tareas como la lectura de un sensor, control de un actuador, ejecución de algoritmos, etc.

Mensajes: forma de comunicación entre nodos. Son estructuras de datos para el intercambio de información entre nodos.

Topics: canales de transporte por donde se envían los mensajes, a través de una dinámica de editor y subscriptor. Un nodo envía un mensaje publicándolo en un topic; el topic es el nombre utilizado para identificar el contenido del mensaje.

Servicios: tienen una función similar a los topics, sin embargo, los servicios generan una respuesta/interacción a partir de las solicitudes enviadas.

Bags: registros en donde se almacenan los datos de un mensaje enviado.

3.2.3. Las limitaciones de ROS 1

La forma en que se manejan las comunicaciones entre nodos de cómputos distribuidos en ROS 1 dificulta la integración entre dispositivos de hardware (sensores, actuadores, etc.). Para realizar una red de procesamiento distribuido en ROS 1, es necesario contar con un dispositivo maestro que inicia antes de cualquier otro nodo. Además, las comunicaciones entre nodos se llevan a cabo utilizando el protocolo de llamada XML-RPC, el cual posee una dependencia significativa cuando se implementa en cualquier sistema de recursos limitados o microcontroladores, debido a su naturaleza recursiva. En vez de lo anterior, es muy común que se utilice un controlador con un protocolo de comunicaciones propio para realizar la interacción entre los dispositivos.

En ROS 1, los nodos comúnmente utilizan la API *Node*, la cual implementa su propia función *main*, en vez de la API *Nodelet* para compilar librerías compartidas. Debido a lo anterior, el desarrollador tiene que escoger entre una de estas dos APIs, y el proceso para convertir de una API a otra no es trivial y requiere una inversión de tiempo considerable.

En cuanto al proceso de lanzamiento, el sistema de lanzamiento de ROS 1 solo inicializa un conjunto de procesos, y no provee ningún tipo de retroalimentación fuera de si el proceso fue iniciado o no. Sin embargo, es común que los desarrolladores escriban sus procesos para que esperen una cierta cantidad de tiempo o una bandera de estado, que indique que todo está bien antes de comenzar a procesar los datos.

Además, en sistemas complejos la observabilidad de los procesos y la posibilidad de una configuración dinámica se vuelven mucho más relevantes. En ROS 1 los nodos no tiene ningún estado asociado y solo algunos componentes una interfaz para obtener información o manipular el sistema durante su ejecución.

A partir de lo anterior, en 2014 una nueva versión de ROS con un enfoque y estructura distinta es anunciada por Open Robotics. ROS 2 surge como un completo rediseño para lo que había sido el entorno de trabajo hasta entonces, con esta reestructuración se busca cubrir necesidades y funcionalidades que no habían sido consideradas con ROS 1, pero que habían sido exigidas por la comunidad y la industria. Lo anterior dio lugar al desarrollo de un nuevo conjunto de paquetes con cambios en API general, arquitectura y comunicación.

3.2.4. ¿Por qué ROS 2?

Como se mencionó anteriormente, ROS surgió con la idea de satisfacer las necesidades de un único modelo de robot, y lo logró; Por otro lado, a pesar de haber ampliado el framework para funcionar para otros dispositivos, como ya se mencionó, con el paso del tiempo los desarrolladores del proyecto de algunas características importantes de las que carece ROS 1, pues si bien es cierto que en los últimos años ROS ha adquirido una importante relevancia, su uso se ha visto limitado a aplicaciones con fines académicos y no ha podido despegar en el sector industrial debido a las limitaciones con las que cuenta, dentro de ellas cabe mencionar el hecho de no estar diseñado para trabajar con sistemas en tiempo real, además de carecer de estándares de seguridad para llevar a cabo certificaciones de este tipo.

Las características mencionadas anteriormente no son para nada triviales y modificar el framework de ROS 1 para implementarlas implica realizar una gran cantidad de cambios, los cuales pueden provocar inestabilidad en el proyecto, pues son adiciones que no se planificaron dentro de la etapa de diseño del framework. Entonces, la opción más razonable para el equipo de desarrollo fue crear un nuevo framework desde cero; ROS 2.

Dicho lo anterior, se tiene definidos nuevos casos de uso que dirigen el desarrollo de ROS 2, al mencionar algunos de los más importantes se encuentran los siguientes:

- Sistemas compuestos por múltiples robots: existe la posibilidad de implementarlos

en ROS 1, pero no existe un estándar o un acercamiento unificado que permita el desarrollo para este tipo de sistemas.

- Sistemas embebidos: se tiene por objetivo que la implementación de ROS en sistemas embebidos, como computadoras de placa única y microcontroladores, no sea a través de un controlador de dispositivo, sino que, sea posibilidad configurar el dispositivo como una computadora normal.
- Sistemas en tiempo real: soporte para este tipo de sistemas de forma nativa en ROS, ofreciendo comunicación inter-proceso e inter-máquina.
- Redes no ideales: desempeño estandarizado incluso si la conectividad de red es deficiente.
- Ambientes de producción: seguir enfocando el desarrollo al área de investigación, pero facilitar la evolución de un mero prototipo a una aplicación comercial.
- Patrones para el desarrollo y estructura de sistemas: conservar la flexibilidad en el desarrollo de soluciones pero proveer estándares y herramientas de desarrollo enfocadas al manejo de un ciclo de vida y configuraciones estáticas para su lanzamiento.

ROS 2 propone una arquitectura en donde es posible implementar el protocolo de comunicación entre nodos directamente en cualquier sistema embebido, de tal forma que cualquier dispositivo ROS dentro de la red sea descubierto de forma automática por la interfaz de ROS; se implementa una comunicación más descentralizada pues ya no existe la figura de maestro y se implementa una lógica de intercambio de información del tipo DDS (Data Distribution Service), la cual está pensada para sistemas en tiempo real. Además, la forma de crear nodos en ROS 2 está pensada para que el usuario sea capaz de decidir el tiempo y la forma de lanzamiento de un nodo; cada nodo puede ser lanzado en procesos distintos para facilitar la depuración de estos, o, pueden ser integrados a un solo proceso para obtener un mejor rendimiento y aprovechar la comunicación inter-proceso.

Lo anterior también conlleva un cambio significativo dentro de la API de ROS; se rediseñó con el fin de mejorarla, de tal forma que los conceptos clave de la versión anterior se conservaron, pero se ofrece una gran mejora y experiencia al usarla. Esto significa que

Distribución	Fecha de lanzamiento	Fin de vida útil
Humble Hawksbill	Mayo 23, 2022	No especificada
Galactic Geochelone	Mayo 23, 2021	Noviembre 2022
Foxy Fitzroy	Junio 5, 2020	Mayo 2023
Eloquent Elusor	Noviembre 22, 2019	Noviembre 2020
Dashing Diademata	Mayo 31, 2019	Mayo 2021
Crystal Clemmys	Diciembre 14, 2018	Diciembre 2019
Bouncy Bolson	Julio 2, 2018	Julio 2019
Ardent Apalone	Diciembre 8, 2017	Diciembre 2018
beta3	Septiembre 13, 2017	Diciembre 2017
beta2	Julio 5, 2017	Septiembre 2017
beta1	Diciembre 19, 2016	Julio 2017
alpha1 - alpha8	Agosto 13, 2015	Diciembre 2016

Tabla 3.2: Lista de distribuciones de ROS 2

la API de ROS 1 no es compatible con la de ROS 2, y viceversa; sin embargo, se busca que el código de ambas versiones pueda coexistir en un mismo sistema e incluso que cuenten con cierto tipo de interacción; esto permite que la transición entre ambas versiones sea de forma gradual y práctica.

La tabla 3.2 presenta un resumen de las distribuciones de ROS 2 desarrolladas hasta el momento, en ella se indica la fecha de lanzamiento de cada una y el periodo en el que se dejara de ofrecer soporte para estas. Cabe destacar que cada distribución de ROS esta asociada a una única versión LTS (Long Term Support) de Ubuntu.

En cuanto a ROS 1, actualmente se encuentra activas dos distribuciones, Melodic Morenia y Noetic Ninjemys. Siendo esta última la versión final de ROS 1, cuyo objetivo es implementar Python 3 para aquellas organizaciones y empresas que necesitan seguir trabajando con ROS 1 por un tiempo. Por otro lado, el servicio de soporte técnico para esta última versión terminará en mayo de 2025, de tal forma que se tiene hasta entonces para que los usuarios de ROS 1 migren a ROS 2, si es que desean seguir utilizando un framework con un desarrollo activo por parte de Open Robotics.

3.2.5. Diferencias entre ROS 1 y ROS 2

Con base en todo lo que se mencionó en las secciones anteriores, la tabla 3.3 hace un compilado de todas las diferencias importantes entre ROS 1 y ROS 2.

Característica	ROS 1	ROS 2
Plataformas	Únicamente soporta de forma nativa Ubuntu.	Soporta Ubuntu, Mac OS X y Windows 10.
Programación	Trabaja con C++03 y Python 2; Python 3 en su última versión con plan de soporte.	Utiliza C++11 de forma extensiva y usa algunas partes de C++14 y Python a partir de la versión 3.5.
Comunicación	Utiliza una capa de comunicación diseñada desde cero (XML-RPC)	Adopta un protocolo ya definido (DDS)
Compilación	Utiliza <i>catkin</i> para compilar e instalar sus paquetes	Implementa <i>Ament</i> , un nuevo sistema de compilación que integra a <i>colcon</i> .
POO	No hay un estándares, cada implementación es única	Existen pautas definidas para escribir los nodos
Archivos de lanzamiento	Se crean utilizando XML y Python, sin embargo, de este último no hay documentación sobre como hacerlo	Permite crearlos de forma nativa en Python ofreciendo una mayor personalización en cada archivo.
Servicios	Son síncronos	Son asíncronos.
Paquetes	Se crean los paquetes y se añaden archivos de forma independiente	Se tiene que especificar si el paquete se basara en Python o C++.

Tabla 3.3: ROS 1 vs. ROS 2

Tomando como referencia la tabla 3.3 es evidente que el paradigma que rige el diseño de ROS 2 es por mucho muy superior en cuanto a estructura, practicidad y mantenibilidad; sin embargo, hoy en día muchas de las implementaciones que estas relacionadas con ROS son realizadas con la primera versión del framework, por lo que ROS 1 sigue teniendo una presencia bastante sólida en comparación con la nueva versión desarrollada. Esto tenderá a cambiar con el paso del tiempo, pero se trata de un proceso gradual que puede llegar a tomar años.

Por otro lado, la comunidad de ROS recomienda a los usuarios nuevos formar parte de la transición e iniciar por aprender los fundamentos de ROS 2. En contraste con lo anterior, aprender las implicaciones y la forma de trabajo de ROS 1 puede resultar benéfico para los nuevos usuarios también, pues otorgaría una formación más sólida con

el objetivo de tener un panorama más detallado sobre la forma de trabajo con ROS, en general.

Por último, existe otro obstáculo que dificulta la transición hacia la nueva versión de ROS, y es que para el usuario promedio puede que el porteo de proyecto de una versión a otra no conlleve tanto esfuerzo, en especial si se trata de proyectos pequeños; sin embargo, al hablar de organizaciones y empresas con proyectos a gran escala, esta actividad puede llegar a representar una cantidad considerable de tiempo y esfuerzo, por lo que es entendible que en estos casos se opte por seguir utilizando ROS 1, y que para nuevos proyectos se opte por utilizar ROS 2.

3.3. Visión Artificial

3.3.1. Concepto

La visión artificial o visión por computadora es un campo de la informática que tiene por objetivo extraer información de imágenes, y a pesar de que el concepto suene sencillo, ha representado un reto a lo largo del tiempo. Se le ha dedicado una gran cantidad de esfuerzo por parte de las mentes más brillantes y creativas de las últimas décadas, y a pesar de ello, el avance tecnológico parece indicar que aún no se logra por completo el objetivo de desarrollar una máquina de propósito general capaz de observar la realidad como lo hace el sistema conjunto del ojo y el cerebro humano.

3.3.2. Segmentación de color utilizando el modelo HSV

Generalmente, las imágenes digitales se encuentran definidas bajo un modelo de color de tres componentes: rojo, verde y azul (RGB), en donde a cada color se le asigna una matriz de las mismas dimensiones de la imagen, a tal forma que las tres matrices en conjunto conforman la concentración de color en los píxeles de la imagen en cuestión (Saravanakumar et al., 2011).

La concentración de cada color está definida por 8 bits; es decir, este parámetro puede tener un valor entre el 0 y el 255, que representan la menor y mayor concentración, respectivamente. El modelo de color RGB representa una forma sencilla para representar la concentración de color en una imagen, sin embargo, cuando se requiere buscar una

tonalidad de color especifica en una imagen, utilizando este espacio de color, la situación puede llegar a complicarse cuando existen cambios de iluminación en el ambiente.

Debido a lo anterior, cuando se quiere realizar operaciones de procesamiento de imágenes, se suele cambiar el modelo de color de la imagen de RBG a HSV, pues este espacio de color permite manejar los cambios de iluminación con más facilidad y utiliza un sistema para la definición del tono mucho más sencillo de utilizar.

De manera similar e intuitiva, el espacio de color HSV está compuesto por tres parámetros para la definición de la concentración de color: el matiz (hue; H), la saturación (saturation; S) y el valor (value; V). Estos tres parámetros son organizados en una lista en el orden mencionado, y contienen un rango de valor asociado a cada uno. A continuación, la tabla 3.4 contiene la descripción de los parámetros utilizados por este modelo de color así como su rango de valores.

Parámetro	Descripción	Rango
Matiz	La tonalidad del color	0 - 179
Saturación	La tonalidad grisácea del color	0 - 255
Valor	brillo del color	0 - 255

Tabla 3.4: Parámetros del espacio de color HSV

3.3.3. OpenCV

Concepto

De acuerdo con su documentación oficial (OpenCV-Team, 2021), OpenCV (del inglés, Open Source Computer Vision Library) es una librería multiplataforma de código abierto que fue diseñada para realizar procesamiento de imágenes en tiempo real, contiene una cantidad considerable de algoritmos de visión artificial, por lo que, es considerada un estándar para implementar aplicaciones en donde es necesario utilizar visión por computadora; está escrita en C/C++ y puede ser instalada en Linux, Windows y Mac OS X. Además, cuenta con interfaces para Python, Ruby, Matlab y otros lenguajes de programación.

Historia

La primera versión alfa de OpenCV surgió a inicios de 1999, fue desarrollada por Intel con objetivos de investigación en aplicaciones de uso intensivo de CPU. En una visita al MIT, uno de los investigadores de Intel descubrió que un grupo de estudiantes habían desarrollado una infraestructura bastante sólida de visión por computadora, de tal forma que en vez de crear sus propias librerías desde cero, los estudiantes se compartían una serie de código fuente base, con el que podían llevar a cabo sus propias implementaciones, reduciendo el tiempo de desarrollo.

De esta manera, OpenCV fue concebida como una infraestructura estándar que buscaba estar disponible para cualquier desarrollador alrededor del mundo. El desarrollo y optimización de la librería fue delegado a un grupo de expertos pertenecientes a Intel, en Rusia (Bradski & Kaehler, 2008). Vadim Pisarevsky fue el líder del proyecto y participó de forma activa en la optimización y codificación de la librería, tal que gran parte del código desarrollado por él continúa siendo el núcleo funcional de la librería. En conjunto con Vadim, Victor Eruhimov apoyó en el desarrollo prematuro de la infraestructura, al igual que Valery Kuriakin, quien gestionó el laboratorio y aportó de forma continua al proyecto.

A partir de lo anterior, OpenCV basó su desarrollo en algunos pilares u objetivos:

- Proveer una infraestructura para visión artificial, no solo abierta, sino también optimizada; evitar reinventar la rueda.
- Descentralizar el conocimiento, poniendo al alcance de cualquier desarrollador una librería base que permitiera crear código con mejor legibilidad y más portabilidad.
- Potenciar aplicaciones comerciales de visión por computadora, ofreciendo código gratuito y de uso libre, con términos y condiciones de uso que no obligaran al desarrollador a publicar su aplicación como código libre.

En cualquier proyecto de código abierto, la participación de la comunidad es de suma importancia, pues el conjunto de esfuerzos es lo que permite que el proyecto se vuelva autosustentable; OpenCV no es la excepción, pues una cantidad considerable de usuarios han contribuido a su desarrollo, a tal grado que, se puede decir que ya no pertenece a Intel, sino a su propia comunidad. Se estima que la librería ha sido descargada más de

dos millones de veces (Bradski & Kaehler, 2008), creciendo de forma continua con un aproximado de hasta 26,000 descargar por mes.

Hoy en día, OpenCV es utilizada en áreas como segmentación y reconocimiento, identificación de objetos, reconocimiento facial, seguimiento de movimiento, realidad aumentada, percepción de profundidad, calibración multi-cámara, entre muchas otras más (García et al., 2015). La explotación y el esfuerzo dedicado a la librería ha sido tal que, la infraestructura ya viene con módulos que integran de modelos estadísticos de machine learning.

Características

OpenCV fue diseñado con una estructura modular, es decir, el paquete en su conjunto está formado por una serie de librerías compartidas o estáticas. La tabla 3.5 presenta la lista de módulos incluidos en la paquetería, así como su respectiva descripción.

3.4. Software in The Loop

El paradigma de simulación de Software in The Loop (SIL) conlleva la generación de un código fuente compilado a partir de un modelo matemático simulado, de tal forma que se crea un ambiente de simulación práctico para el desarrollo y validación de estrategias de control para sistemas complejos y de gran escala (OPAL-RT-TECHNOLOGIES, 2022c).

Con SIL, los ingenieros pueden utilizar una computadora de escritorio para interactuar con la simulación y modificar su código fuente, integrando su algoritmo de control a una planta virtual, substituyendo los prototipos costosos o los bancos de pruebas complejos. El SIL posibilita las pruebas del software antes de la inicialización de la fase de prototipaje, mejorando de forma significativa los tiempos del ciclo de desarrollo del sistema.

Complementando lo anterior, la simulación de SIL permite detectar bugs o defectos en el código fuente en etapas tempranas de del ciclo de desarrollo, reduciendo los costos derivados de la solución de estos errores en etapas más avanzadas del desarrollo del sistema, cuando el número de componentes e interacciones entre ellos son más complejas y numerosas.

A manera general, la simulación de SIL contempla la simulación del comportamiento

Módulo	Descripción
Funcionalidad nuclear	Define las estructura de datos base e incluye funciones altamente sofisticadas para operaciones con arreglos multidimensionales y funciones utilizadas por otros módulos
Procesamiento de imágenes	Incluye filtros lineales y no lineales para el procesamiento de imágenes, transformaciones geométricas, conversión de espacio de color, histogramas y más.
Análisis de video	Constituido por funciones para estimación de movimiento, extracción de fondo, y algoritmos para seguimiento de objetos.
Calibración de cámaras y reconstrucción 3D	Algoritmos multi-geométricos básicos, estimación de posición de objetos, reconstrucción 3D de elementos.
Funcionalidades en 2D	Detectores de características, descriptores y descriptores enlazadores
Detección de objetos	Detección de objetos e instancias de clases especificadas.
Interfaz gráfica de alto nivel	Interfaz fácil de utilizar con funcionalidades simples
E/S de video	Interfaces para captura de video y codecs.
Otras	OpenCV cuenta con muchos más módulos, para más información consultar la documentación oficial (OpenCV-Team, 2021)

Tabla 3.5: Módulos de OpenCV

de un algoritmo de control sobre el modelo de una planta o proceso; a diferencia otros paradigmas como Hardware in The Loop (HIL, OPAL-RT-TECHNOLOGIES (2022a)) o Rapid Control Prototyping (RCP, OPAL-RT-TECHNOLOGIES (2022b)), en una simulación de SIL tanto el sistema de control como el proceso de interés son completamente simulados a partir de modelos matemáticos; es decir, no se requiere ningún tipo de prototipaje físico más que el dispositivo simulador y una computadora para interactuar con la simulación.

Algunas de las ventajas inherentes del paradigma de SIL son:

- Reducción del tiempo de desarrollo y búsqueda de errores y fallas en el código fuente

- Existe la posibilidad de reutilizar modelos en distintos proyectos
- Se mejora la eficiencia y la calidad del software de control desarrollado.

3.4.1. Ardupilot

Concepto

Ardupilot es definido por sus autores (ArduPilot-Dev-Team., 2022) como un sistema de piloto automático confiable, versátil y de código abierto, capaz de funcionar en diversos vehículos, tales como: multi-copteros, helicópteros, aeronaves de ala fija, botes, submarinos, rovers y más. Al igual que otros proyectos grandes de código abierto, su desarrollo es llevado a cabo por una gran comunidad de profesionales y entusiastas.

El firmware de ArduPilot permite desarrollar sistemas de navegación para vehículos autónomos no tripulados prácticamente de cualquier tipo, y al ser de código abierto, se encuentra en un estado de mejora continua, más aún, el equipo de desarrollo trabaja de forma colaborativa con socios comerciales para añadir funcionalidades por el beneficio de toda la comunidad. Por otro lado, si bien es cierto que ArduPilot no se dedica a la manufacturación de hardware, su firmware trabaja en una amplia gama de hardware para el control de vehículos no tripulados.

Actualmente, se estima que ArduPilot ha sido instalado en más de un millón de vehículos alrededor del mundo (ArduPilot-Dev-Team., 2022); además, integra un registro de datos avanzados, herramientas de análisis y simulación, por lo que ha pasado por un proceso de validación bastante extenso, lo que le da una fiabilidad y seguridad bastante alta para sistemas de piloto automático. Además, los usuarios de ArduPilot tienen acceso a una gran cantidad de interfaces para sensores, computadoras de compañía y sistemas de comunicación.

Por último, ArduPilot ha adquirido una relevancia tal que, es usado ampliamente en vehículos desarrollados por instituciones y corporaciones de gran importancia, tales como la NASA, Intel, Boeing y una gran cantidad de universidades y colegios de alrededor del mundo.

Framework de SIL

Por otro lado, pese a que ArduPilot está diseñado para ser instalado en hardware físico, como se mencionó en el apartado anterior, también cuenta con herramientas avanzadas de simulación, tal es el caso de su framework destinado a simulaciones de SIL, lo que permite ejecutar perfiles de pilotos automáticos de vehículos como aviones, multi-copteros o rovers, con el objetivo de observar el comportamiento del piloto automático sin necesidad de contar con el vehículo o computadora de vuelo en físico.

El framework de SIL de ArduPilot permite ejecutar ArduPilot en cualquier computadora de escritorio, sin necesidad de cualquier otro tipo de hardware especial. Esto es debido a que ArduPilot puede ser visto como un piloto automático portable que puede ser instalado y ejecutado en una gran variedad de plataformas. Dicho de esta forma, una computadora personal puede ser vista como un dispositivo más, en donde se desea instalar ArduPilot.

Es importante aclara que cuando la simulación de SIL es ejecutada, los datos *supuestamente* leídos por los sensores del piloto automático son generados a partir de las dinámicas de vuelo del simulador de vuelo. ArduPilot cuenta con una gran variedad de simuladores para distintos vehículos, y puede integrarse con simuladores externos. Lo anterior permite desarrollar aplicaciones para distintos vehículos y sensores, entre ellos:

- Aeronaves multi-rotor
- Aeronaves de ala fija
- Vehículos terrestres
- Vehículos subacuáticos
- Gimbal de cámaras
- Seguidores para antenas
- Sensores ópticos

Además, el framework de SIL provee al usuario con una suite de herramientas de desarrollo, tales como depuradores de código, analizadores estáticos, herramientas de

análisis dinámico. El ambiente de SIL fue desarrollado para ejecutarse de forma nativa en los sistemas operativos Linux y Windows.

Por último en cuanto al ambiente de simulación de ArduPilot, la figura 3.1, proporciona una perspectiva ejemplo de los puertos de comunicación utilizados para la comunicación de ArduPilot con más software.

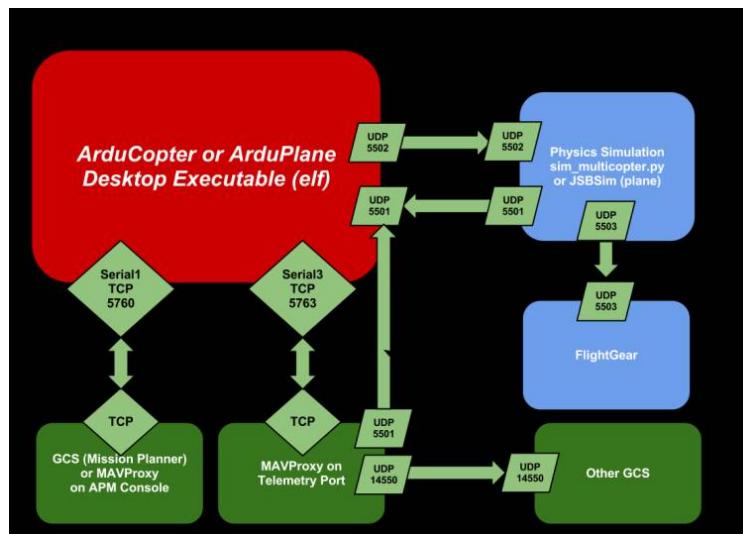


Figura 3.1: Ejemplo de puertos de comunicación de Ardupilot

PymavLink

De acuerdo con su documentación oficial (Lorenz Meier, 2021), MAVLink es un protocolo para comunicación con drones y sus componentes; sigue un patrón de diseño híbrido de publicación-suscripción y comunicación punto-a-punto, en donde la trama de datos es enviada mediante el primer paradigma y los protocolos de configuración, como misiones y parámetros de vuelo, son enviados mediante el segundo.

En otras palabras, el uso del protocolo MAVLink es necesario para establecer comunicación con la simulación del controlador de vuelo provisto por Ardupilot, y de esta forma obtener los parámetros de vuelo de dron simulado así como el envío de comandos de vuelo.

Dicho lo anterior, MAVROS (Ermakov, 2021) es un paquete de ROS que crea un nodo nativo capaz de establecer comunicación con un controlador de vuelo mediante MAVLink,

por lo que es una alternativa bastante popular dentro de la comunidad, para la integración de software y proyectos relacionados con ROS y drones; sin embargo, a pesar de que el paquete se encuentre en constante desarrollo, la versión estable del mismo solo se está disponible para ROS 1, mientras que la implementación para ROS 2 todavía se encuentra en una etapa prematura de su desarrollo al tiempo de escritura de este trabajo, ofreciéndose en una versión alfa, la cual no cuenta con todas las funcionalidades ni con la estabilidad ofrecida por la implementación de ROS 1.

3.5. Gazebo

3.5.1. Concepto

Dentro del desarrollo de sistemas, es indispensable llevar a cabo un proceso de validación que permita evaluar el cumplimiento de los requerimientos funcionales del sistema desarrollado, tanto para software como para hardware. Hoy en día existen muchas herramientas que facilitan la ejecución de pruebas de validación para sistemas complejos; en este aspecto, el software de simulación corresponde a una gran alternativa pues permite realizar las pruebas de validación pertinentes bajo distintos paradigmas, en donde es necesario (o no) contar con un prototipo físico del sistema o un modelo matemático del mismo, lo anterior agiliza el desarrollo del sistema y probar algoritmos, diseños e incluso llevar a cabo el entrenamiento de algoritmos basados en inteligencia artificial; además, permite identificar errores y fallas en etapas tempranas del desarrollo del sistema, reduciendo costos y asegurando un mayor grado de calidad en el producto final.

De acuerdo con su sitio oficial (Open-Robotics, 2014), Gazebo es un software de simulación que ofrece la posibilidad de simular de forma precisa y eficiente, conjuntos de robots en ambientes complejos de interiores y exteriores. Además, ofrece un motor de físicas robusto, gráficos de alta calidad e interfaces gráficas y programáticas convenientes. Por último, al ser un proyecto desarrollado bajo el paradigma de open-source, cuenta con una comunidad bastante amplia que distribuye el conocimiento y se encarga de darle mantenimiento a este.

3.5.2. Historia

Creado por el Dr. Andrew Howard y su estudiante Nate Koeing, Gazebo comenzó su desarrollo en otoño de 2002 en la Universidad del Sur de California (Open-Robotics, 2014). El concepto de un simulador de alta fidelidad para robots, surgió a partir de la necesidad de simular este tipo de sistemas en ambientes bajo distintas condiciones. Gazebo fue nombrado de esta forma debido a que la estructura asociada a la palabra es un claro representante de un ambiente de exteriores; sin embargo, a pesar de que la mayoría de los usuarios de Gazebo lo utilizaban para simular ambientes de interiores, el nombre persistió hasta el día de hoy.

Con el paso de los años, Nate se encargó del desarrollo de Gazebo mientras terminaba su doctorado, y en 2009, John Hsu, un ingeniero en Willow garage, logró integrar ROS y el robot PR2 en Gazebo; desde entonces, Gazebo se convirtió en la herramienta de preferencia para llevar a cabo simulaciones de robots, por parte de la comunidad y usuarios de ROS.

Después, en la primavera de 2011, Willow Garage comenzó a financiar el desarrollo de Gazebo, y más tarde, en 2012, la Open Source Robotics Foundation (OSRF; que más tarde se convertiría en Open Robotics) surgiría como un organismo independiente a Willow Garage, continuando con el desarrollo de Gazebo.

Hoy en día Open Robotics representa el equipo de desarrollo principal para Gazebo, en conjunto con la gran y diversa comunidad que ha acompañado al proyecto a lo largo de su desarrollo.

3.5.3. ¿Por qué Gazebo?

A demás de las ventajas ya mencionadas, inherentes al uso de un simulador para la etapa de validación de un sistema, Gazebo cuenta con una serie de características que lo hacen único y por las cuales destaca.

- **Proyecto de código abierto:** el código fuente de Gazebo está liberado al público, por lo que cualquier usuario experimentado puede entender su funcionamiento y contribuir a su desarrollo; además, al ser software libre, Gazebo puede ser utilizado por cualquier tipo de usuario prácticamente sin ningún tipo de restricción.

- **Simulación de dinámicas:** es posible utilizar una serie de motores de físicas de código abierto, tales como ODE (Open Dynamics Engine), Simbody y DART (Dynamic Animation and Robotics Toolkit)
- **Gráficos avanzados en 3D:** Gazebo utiliza OGRE, un motor gráfico de código abierto que permite crear ambientes de simulación con gráficos realistas, incluyendo iluminación, sombras y texturas de alta calidad.
- **Sensores y ruido:** permite generar datos a partir de una amplia gama de sensores simulados, desde láseres, cámaras, sensores de movimiento, sensores de fuerza y torque, etc.
- **Plugins:** es posible desarrollar plugins personalizados para sensores, robots y el control de ambiente, con base en las necesidades del usuario.
- **Modelos de robots:** Gazebo cuenta con un catálogo extenso de modelos de robots existentes en el mercado, que pueden ser utilizados por cualquier usuario; sin embargo, también existe la posibilidad de que el usuario cree su propio modelo de robot desde cero.
- **Comunicación TCP/IP:** las simulaciones pueden ser ejecutadas desde servidores remotos y accedidas a partir de un protocolo de comunicación basado en el envío de mensajes por socket.
- **Simulación en la nube:** Gazebo puede ser ejecutado en una máquina virtual basada en la nube utilizando CloudSim y permite la interacción con la simulación a partir de su cliente web GzWeb.
- **Herramientas basadas en la línea de comandos:** Gazebo cuenta con un conjunto de herramientas para la línea de comandos que facilitan la interacción con la simulación en ejecución y así como su control.

CAPÍTULO 4

RESULTADOS

En este capítulo se presentan los resultados obtenidos; además, se documenta de forma detallada el procedimiento realizado para configurar cada uno del software utilizado, lo anterior debido a que este trabajo también busca funcionar como una guía estructurada que permita la réplica de la implementación desarrollada.

4.1. Configuración del Framework de SITL y Ambiente de Simulación

La implementación del trabajo se realizó en una computadora portátil modelo *Acer Aspire E5-575*. A continuación se anexan las características físicas más relevantes del hardware utilizado.

Parámetro	Descripción
Procesador	Intel Core i3-7100U; Dual-core 2.40 GHz
Memoria RAM	12 GB DDR4
Disco duro	1 TB Toshiba HDD
Coprocessor de gráficos	Intel HD Graphics 620

Tabla 4.1: Características técnicas de la laptop Aspire E5-575

La figura 4.1 muestra con más detalle las características de hardware y software correspondientes a la computadora donde se llevó a cabo el proyecto.

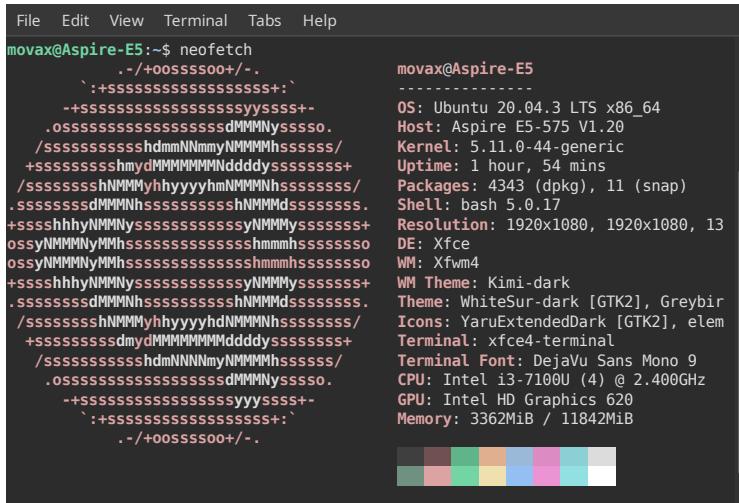


Figura 4.1: Características del sistema en donde se desarrolló el proyecto

Con respecto al software, se trabajó con las versiones más recientes y técnicamente compatibles del software que integra el sistema. Para el ambiente de simulación se utilizó *Gazebo 11* en conjunto con *ArduPilot* y el modelo ofrecido para SITL de Arducopter; Para la gestión de procesos se utilizó *ROS2 Foxy*. Además, la computadora en donde se implementó el sistema viene por defecto con el sistema operativo *Windows 10*; sin embargo, para poder integrar el software mencionado es necesario utilizar *Ubuntu 20.04.3 LTS (Focal Fossa)*, el cual fue instalado en un disco duro externo.

4.1.1. Instalación de ROS 2

La siguiente serie de comandos fue extraída de la documentación oficial de ROS2 Foxy(Robotics, 2021) y se asume que la instalación se lleva a cabo en un sistema con Ubuntu 20.04 o sus derivados (*Xubuntu*, *Kubuntu*, etc.). El proceso puede ser distinto para cualquier otra distribución de Linux o Sistema operativo no listado en la documentación oficial, o incluso puede que no sea compatible.

Cabe destacar que existen dos formas de instalar ROS 2, la primera, la forma corta, es utilizar un paquete binario pre-compilado; sin embargo, este tipo de instalación no incluye la paquetería completa de ROS 2, sino, algunos paquetes base que son más que

suficientes para comenzar a desarrollar aplicaciones en ROS. Este primer método puede ser consultado en la documentación oficial de ROS 2 (Open-Robotics, 2021a).

Por otro lado, la segunda forma de instalación, la utilizada en este trabajo, corresponde a compilar ROS 2 desde cero en el sistema. A continuación se anexa la serie de comandos y configuración que corresponden a este último método de instalación.

1. Revisar que el sistema donde se instalará ROS2 admite la codificación de caracteres *UTF-8*, mediante el siguiente comando.

```
$ locale
```

Sí la codificación se encuentra en la lista, se puede saltar al paso 3, si no, seguir se debe seguir con el resto de pasos.

2. Instalar la codificación de caracteres

```
$ sudo apt update && sudo apt install locales  
$ sudo locale-gen en_US en_US.UTF-8  
$ sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8  
$ export LANG=en_US.UTF-8  
$ locale #verificacion de instalacion
```

3. Añadir el repositorio de ROS 2 al sistema.

```
$ sudo apt update && sudo apt install curl gnupg2 lsb-release  
$ sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg  
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

4. Instalar las herramientas de desarrollo para ROS 2

```
$ sudo apt update && sudo apt install -y \ build-essential \
cmake \
git \
```

```
libbullet-dev \
python3-colcon-common-extensions \
python3-flake8 \
python3-pip \
python3-pytest-cov \
python3-rosdep \
python3-setuptools \
python3-vcstool \
wget
# Paquetes de Python 3 para pruebas
$ python3 -m pip install -U \
argcomplete \
flake8-blind-except \
flake8-builtins \
flake8-class-newline \
flake8-comprehensions \
flake8-deprecated \
flake8-docstrings \
flake8-import-order \
flake8-quotes \
pytest-repeat \
pytest-rerunfailures \
pytest
# Dependencias Fast-RTPS
$ sudo apt install --no-install-recommends -y \
libasio-dev \
libtinyxml2-dev
# Dependencias Cyclone DDS
$ sudo apt install --no-install-recommends -y \
libcunit1-dev
```

5. Clonar el código fuente de ROS 2

```
$ mkdir -p ~/ros2_foxy/src #crea el ambiente de trabajo
$ cd ~/ros2_foxy
$ wget https://raw.githubusercontent.com/ros2/ros2/foxy/
ros2.repos
$ vcs import src < ros2.repos
```

6. Instalar dependencias

```
$ sudo rosdep init  
$ rosdep update  
$ rosdep install --from-paths src --ignore-src -y --skip-keys "fastcdr rti-connext-dds-5.3.1 urdfdom_headers"
```

7. Compilar código fuente

```
$ cd ~/ros2_foxy/  
$ colcon build --symlink-install
```

8. Habilitar la API de ROS 2 en bash

```
$ source /opt/ros/foxy/setup.bash
```

9. Modificar el perfil de bash para que inicie ROS 2 con cada nueva terminal

```
$ echo "source /opt/ros/foxy/setup.bash" >> ~/.bashrc
```

Hecho lo anterior, el sistema debe de contar con una instalación completa de ROS 2. Para comprobar que la instalación se llevó a cabo de manera correcta, se pueden ejecutar los nodos demo que vienen incluidos en la instalación de escritorio.

Para ejecutar los nodos de demostración es necesario abrir dos terminales y ejecutar en cada una uno de los siguientes comandos:

1. **Talker**. Nodo publicador escrito en C++

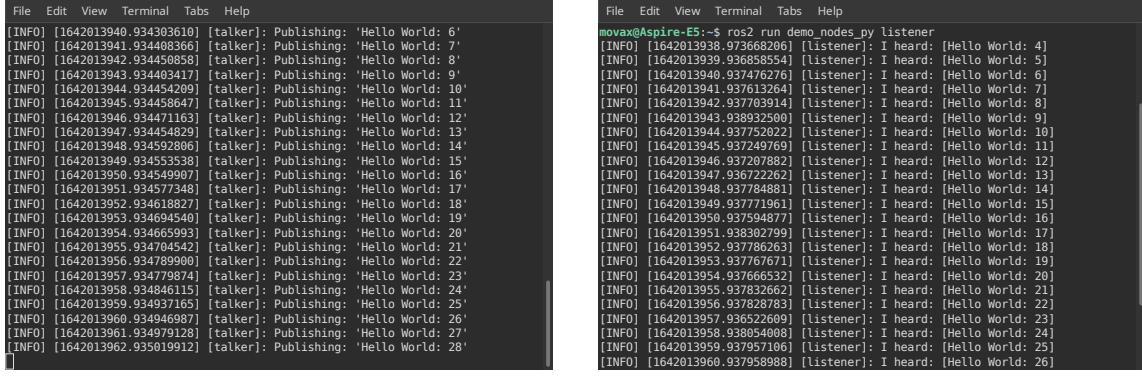
```
$ ros2 run demo_nodes_cpp talker
```

2. **Listener**. Nodo suscriptor escrito en Python

```
$ ros2 run demo_nodes_py listener
```

Las figuras 4.2a y 4.2b muestran la ejecución de los nodos anteriores, respectivamente. El nodo publicador envía un mensaje con un contador que va incrementando con cada mensaje enviado, mientras que el nodo suscriptor se encarga de recibir el mensaje enviado e imprimirlo en pantalla.

Capítulo 4. Resultados



```

File Edit View Terminal Tabs Help
[INFO] [1642013940, 934303610] [talker]: Publishing: 'Hello World: 6'
[INFO] [1642013941, 934408366] [talker]: Publishing: 'Hello World: 7'
[INFO] [1642013942, 934450858] [talker]: Publishing: 'Hello World: 8'
[INFO] [1642013943, 934403417] [talker]: Publishing: 'Hello World: 9'
[INFO] [1642013944, 934454209] [talker]: Publishing: 'Hello World: 10'
[INFO] [1642013945, 934458647] [talker]: Publishing: 'Hello World: 11'
[INFO] [1642013946, 934471163] [talker]: Publishing: 'Hello World: 12'
[INFO] [1642013947, 934454829] [talker]: Publishing: 'Hello World: 13'
[INFO] [1642013948, 934592806] [talker]: Publishing: 'Hello World: 14'
[INFO] [1642013949, 934553538] [talker]: Publishing: 'Hello World: 15'
[INFO] [1642013950, 934549907] [talker]: Publishing: 'Hello World: 16'
[INFO] [1642013951, 934577348] [talker]: Publishing: 'Hello World: 17'
[INFO] [1642013952, 934618827] [talker]: Publishing: 'Hello World: 18'
[INFO] [1642013953, 934694540] [talker]: Publishing: 'Hello World: 19'
[INFO] [1642013954, 934665993] [talker]: Publishing: 'Hello World: 20'
[INFO] [1642013955, 934704542] [talker]: Publishing: 'Hello World: 21'
[INFO] [1642013956, 934789900] [talker]: Publishing: 'Hello World: 22'
[INFO] [1642013957, 934779874] [talker]: Publishing: 'Hello World: 23'
[INFO] [1642013958, 934846115] [talker]: Publishing: 'Hello World: 24'
[INFO] [1642013959, 934937165] [talker]: Publishing: 'Hello World: 25'
[INFO] [1642013960, 934946987] [talker]: Publishing: 'Hello World: 26'
[INFO] [1642013961, 934979128] [talker]: Publishing: 'Hello World: 27'
[INFO] [1642013962, 935019912] [talker]: Publishing: 'Hello World: 28'

File Edit View Terminal Tabs Help
novalx@Aspire-E5:~$ ros2 run demo_nodes_py listener
[INFO] [1642013938, 9373668206] [listener]: I heard: [Hello World: 4]
[INFO] [1642013939, 936858554] [listener]: I heard: [Hello World: 5]
[INFO] [1642013940, 937476276] [listener]: I heard: [Hello World: 6]
[INFO] [1642013941, 937613264] [listener]: I heard: [Hello World: 7]
[INFO] [1642013942, 937763914] [listener]: I heard: [Hello World: 8]
[INFO] [1642013943, 938932500] [listener]: I heard: [Hello World: 9]
[INFO] [1642013944, 937752622] [listener]: I heard: [Hello World: 10]
[INFO] [1642013945, 937249769] [listener]: I heard: [Hello World: 11]
[INFO] [1642013946, 937207882] [listener]: I heard: [Hello World: 12]
[INFO] [1642013947, 936722262] [listener]: I heard: [Hello World: 13]
[INFO] [1642013948, 937784881] [listener]: I heard: [Hello World: 14]
[INFO] [1642013949, 937771961] [listener]: I heard: [Hello World: 15]
[INFO] [1642013950, 937594877] [listener]: I heard: [Hello World: 16]
[INFO] [1642013951, 938302799] [listener]: I heard: [Hello World: 17]
[INFO] [1642013952, 937786263] [listener]: I heard: [Hello World: 18]
[INFO] [1642013953, 937767671] [listener]: I heard: [Hello World: 19]
[INFO] [1642013954, 937666532] [listener]: I heard: [Hello World: 20]
[INFO] [1642013955, 937628662] [listener]: I heard: [Hello World: 21]
[INFO] [1642013956, 937628783] [listener]: I heard: [Hello World: 22]
[INFO] [1642013957, 936522689] [listener]: I heard: [Hello World: 23]
[INFO] [1642013958, 936054088] [listener]: I heard: [Hello World: 24]
[INFO] [1642013959, 93797106] [listener]: I heard: [Hello World: 25]
[INFO] [1642013960, 937958988] [listener]: I heard: [Hello World: 26]

```

(a) Nodo publicador

(b) Nodos suscriptor

Figura 4.2: Nodos de demostración incluidos en la instalación de ROS 2

4.1.2. Instalación de OpenCV

Como prerequisito para instalar la librería de OpenCv es indispensable contar con Python 3 y su gestor de paquetes *pip*.

La gran mayoría de distribuciones basadas en Ubuntu vienen con Python 3 instalado por defecto. Se puede verificar su instalación con el siguiente comando:

```
$ python3 --version
```

La expresión anterior debería de imprimir la versión de Python con la que cuenta el sistema, o en su defecto, si Python no se encuentra instalado, se muestra un mensaje que indica que el comando ingresado no existe. Si este último no es el caso, se puede proceder directamente a la instalación de pip.

Instalación de Python 3:

- Actualizar la lista de repositorios del sistema

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

- Instalar Python 3 desde los repositorios oficiales de Ubuntu

```
$ sudo apt install python 3
```

- Verificar la instalación de Python

```
$ python3 --version
```

4. Instalar pip

```
$ sudo apt install python3-pip
```

5. Verificar instalación de pip

```
$ pip3 --version
```

Una vez que se cumplió con el prerequisito anterior, se puede proceder con la instalación de OpenCV. Cabe mencionar que se aconseja usar ambientes virtuales de Python por cada proyecto, esto con el objetivo de evitar que la instalación de paquetes afecte a otros proyectos desarrollados en el mismo sistema. Sin embargo, debido a que el sistema que utilizado estuvo enfocado exclusivamente a la elaboración de este proyecto, en este trabajo se muestra la instalación global de la librería.

La instalación de la librería es sencilla y se puede realizar con un único comando

```
$ pip install opencv-contrib-python
```

Para verificar la instalación de la librería, se puede ejecutar un pequeño script de Python desde la terminal.

```
$ python3
>>> import cv2
>>> cv2.__version__
```

Si la instalación se realizó de forma correcta, se debe de mostrar un mensaje donde se indica la versión de OpenCV que se instaló, como se muestra en la figura 4.3

4.1.3. Instalación de Gazebo

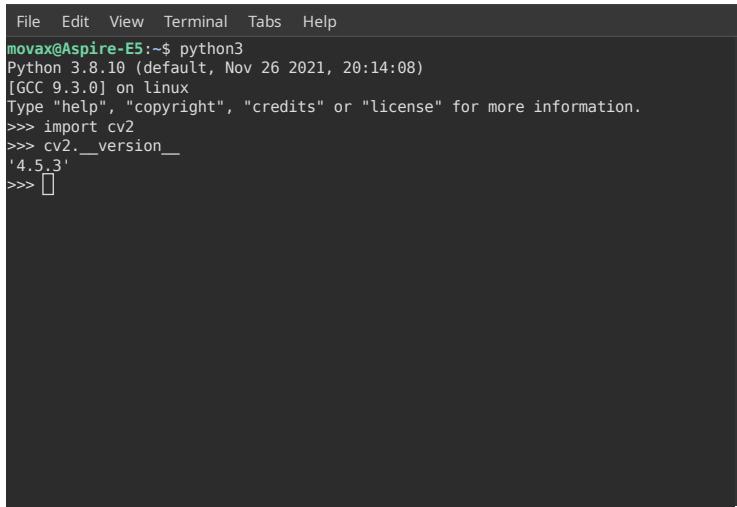
Como se mencionó en el marco teórico, Gazebo es un ambiente de simulación independiente; es decir, no necesita de ROS o ArduPilot para funcionar. Sin embargo, para habilitar la comunicación entre los nodos de ROS y Gazebo, es recomendable realizar la instalación de Gazebo utilizando los repositorios ofrecidos por ROS 2.

La instalación es sencilla y solo requiere ejecutar el siguiente comando:

```
$ sudo apt install ros-foxy-gazebo-ros-pkgs
```

Al ejecutar la instrucción anterior, se instala en conjunto Gazebo y el plugin para la comunicación entre ROS y Gazebo, *gazebo_ros_pkg*. Además, el repositorio también

Capítulo 4. Resultados



```
File Edit View Terminal Tabs Help
movax@Aspire-E5:~$ python3
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
>>> cv2.__version__
'4.5.3'
>>> 
```

Figura 4.3: Mensaje de validación para la instalación de OpenCV

incluye una serie de simulaciones de prueba para demostrar la manera en la que se lleva a cabo la comunicación entre una simulación en Gazebo y un nodo de ROS.

Por otro lado, cabe destacar que, de la misma forma en la cada versión de ROS es desarrollada para trabajar bajo una versión específica de Ubuntu, cada versión de ROS también tiene asociada una única versión compatible de Gazebo; para el caso de ROS 2 Foxy, se trabaja con la última versión disponible, Gazebo 11. La figura 4.4 muestra los datos técnicos sobre la versión de Gazebo con la que se trabajó.



Figura 4.4: Ficha técnica de la versión de Gazebo

Una vez que terminó la ejecución del comando anterior, se puede verificar que la instalación se realizó de manera correcta ejecutando Gazebo desde la terminal, tal que

```
$ gazebo
```

Capítulo 4. Resultados

La instrucción anterior ejecuta una instancia de Gazebo, en donde al no haber ingresado ningún parámetro para cargar un mundo o ambiente de simulación existente, se abre la pantalla inicial del simulador, con un mundo vacío, tal como se muestra en la figura 4.5.

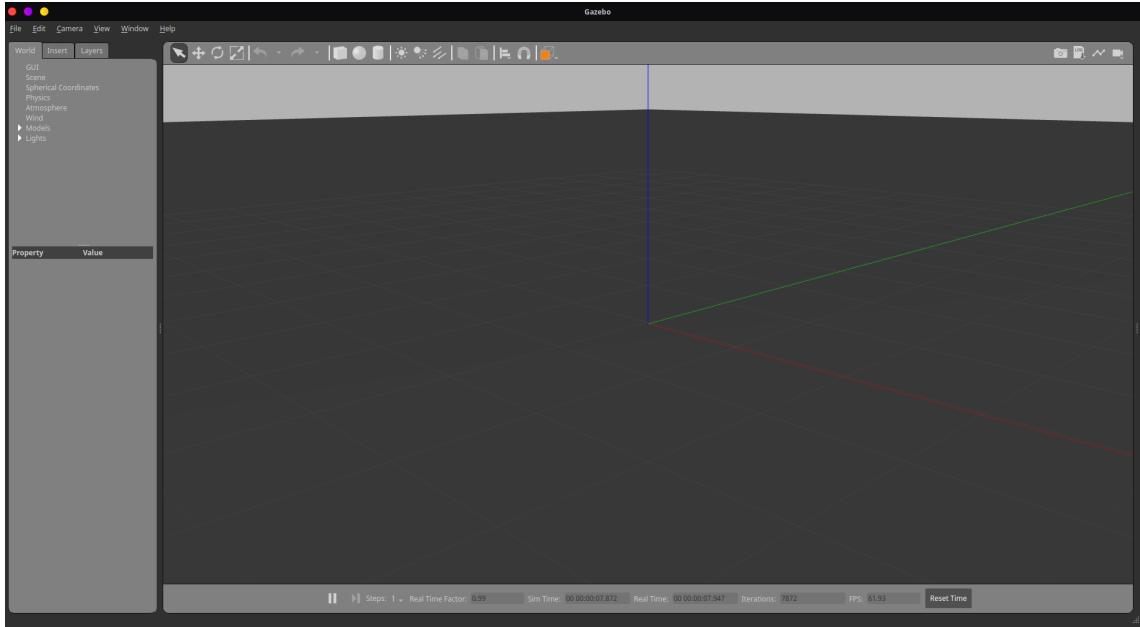


Figura 4.5: Proyecto vacío generado al inicializar Gazebo

Por otro lado, con el fin de comprobar la comunicación entre ROS y Gazebo, se puede ejecutar una de las simulaciones demos incluidas en la instalación. Para ello se selecciona una de las simulaciones más básicas, en donde se tiene un modelo sencillo de un robot y por medio de un topic de ROS se envían instrucciones al robot para su desplazamiento.

Para realizar lo anterior primero se debe de ejecutar una instancia de Gazebo con el mundo que se desea simular.

```
$ gazebo --verbose /opt/ros/foxy/share/gazebo_plugins/worlds/gazebo_ros_diff_drive_demo.world
```

Una vez iniciada la simulación, se pueden enviar instrucciones para el robot por medio de ROS, de la siguiente manera:

```
$ ros2 topic pub /demo/cmd_demo geometry_msgs/Twist '{ linear: {x: 1.0} }' -1
```

Capítulo 4. Resultados

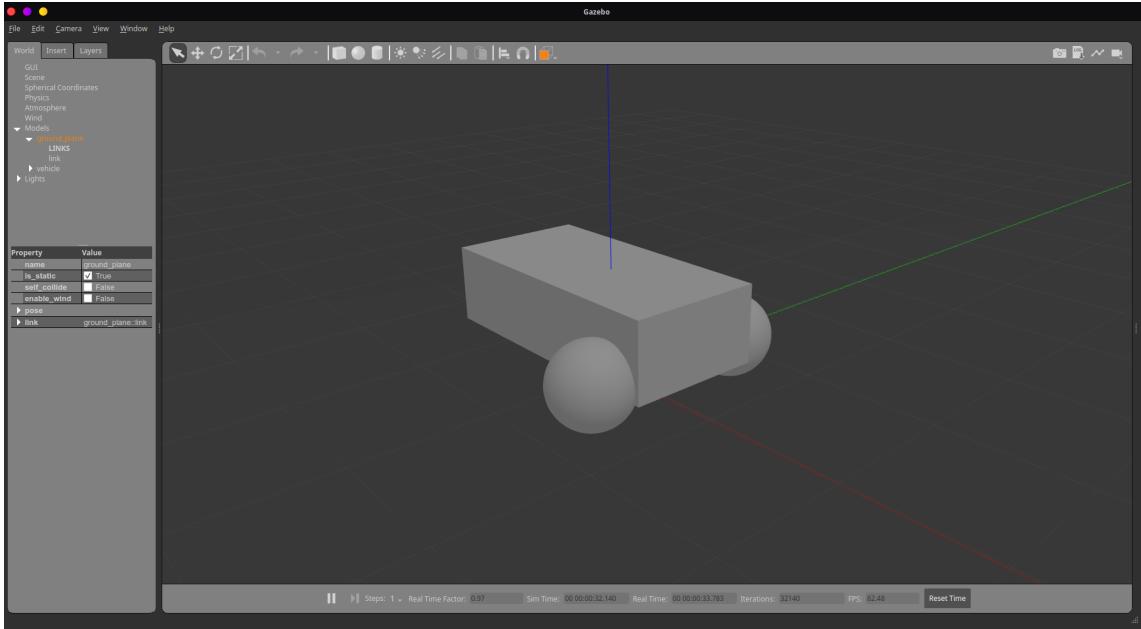


Figura 4.6: Proyecto de demostración en Gazebo que incluye el plug-in para comunicarse con ROS

La figura 4.7 muestra el movimiento observado en la simulación a partir de haber ingresado un comando por medio de la API de ROS.

Para consultar el resto de simulaciones de demostración incluida, se puede acceder al directorio donde se encuentra instalado ROS y listar los nombre de las simulaciones instaladas. Es posible abrir los archivos de simulación con un editor de texto y observar la documentación incluida en cada una, en donde se especifica el modo de uso de esta, la interfaz de mensajes que utilizar para la comunicación y la sintaxis necesaria para enviar mensajes utilizando ROS.

```
$ cd /opt/ros/foxy/share/gazebo-plugins/worlds  
$ ls
```

4.1.4. Instalación de ArduPilot SITL Simulator

Configurar y trabajar con el framework de simulación de ArduPilot es quizás la parte más compleja en cuanto al software utilizado para el sistema propuesto. Esto es debido a que la gran parte de la documentación oficial se encuentra desactualizada y los recursos

Capítulo 4. Resultados

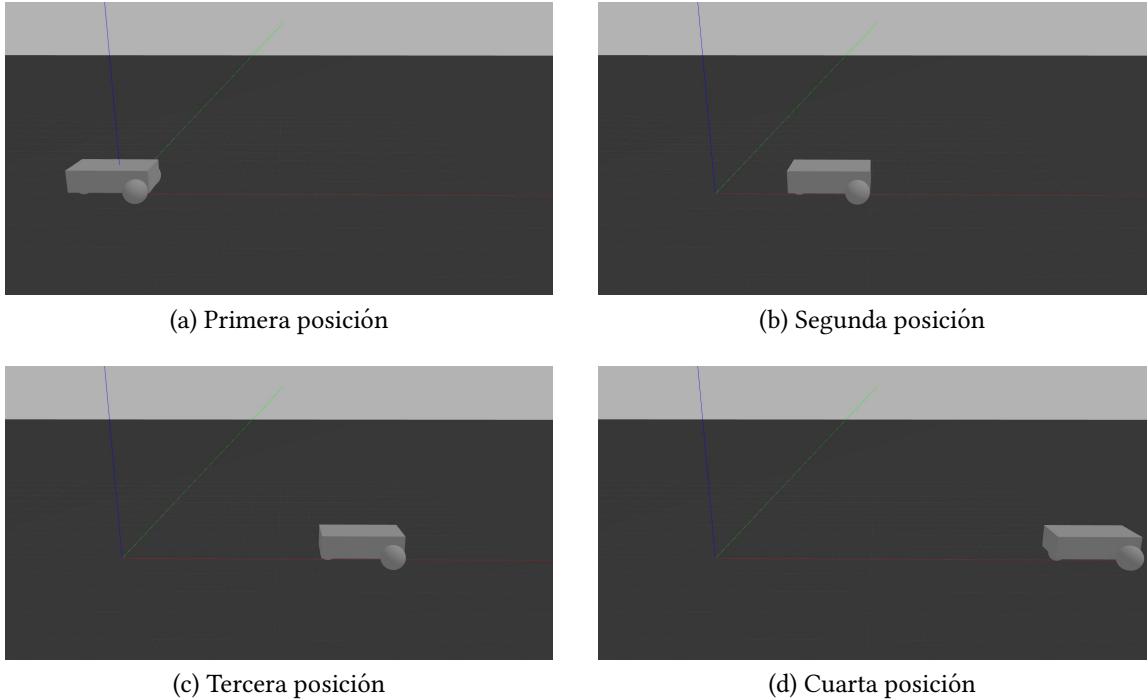


Figura 4.7: Movimiento de la simulación en Gazebo con plugin de comunicación de ROS

que proveen información al respecto se encuentran dispersos por foros y otros tipos de documentación no oficial.

A continuación se muestra una síntesis del proceso de instalación y Configuración para ArduPilot SITL Simulator.

1. Ubicarse en el directorio donde se desean almacenar los archivos del repositorio de ArduPilot y clonar el proyecto.

```
$ git clone --recursive https://github.com/ArduPilot/  
      arduPilot.git  
$ cd arduPilot
```

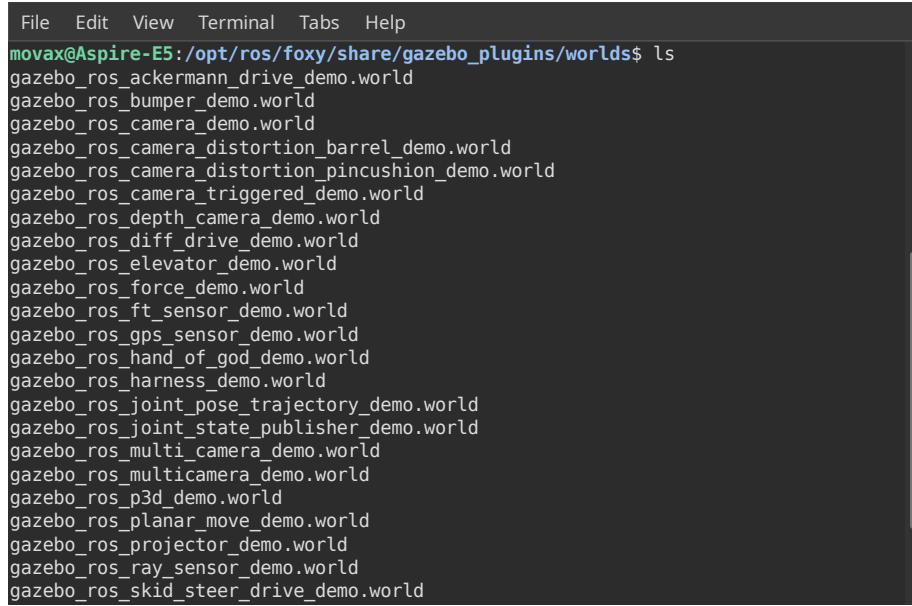
2. Instalar las herramientas necesarias para la compilación de ArduPilot

```
$ Tools/environment_install/install-prereqs-ubuntu.sh -y
```

3. Recargar la ruta de trabajo para hacer uso de las herramientas

```
$ . ~/.profile
```

Capítulo 4. Resultados



```
File Edit View Terminal Tabs Help
movax@Aspire-E5:/opt/ros/foxy/share/gazebo_plugins/worlds$ ls
gazebo_ros_ackermann_drive_demo.world
gazebo_ros_bumper_demo.world
gazebo_ros_camera_demo.world
gazebo_ros_camera_distortion_barrel_demo.world
gazebo_ros_camera_distortion_pincushion_demo.world
gazebo_ros_camera_triggered_demo.world
gazebo_ros_depth_camera_demo.world
gazebo_ros_diff_drive_demo.world
gazebo_ros_elevator_demo.world
gazebo_ros_force_demo.world
gazebo_ros_ft_sensor_demo.world
gazebo_ros_gps_sensor_demo.world
gazebo_ros_hand_of_god_demo.world
gazebo_ros_harness_demo.world
gazebo_ros_joint_pose_trajectory_demo.world
gazebo_ros_joint_state_publisher_demo.world
gazebo_ros_multi_camera_demo.world
gazebo_ros_multicamera_demo.world
gazebo_ros_p3d_demo.world
gazebo_ros_planar_move_demo.world
gazebo_ros_projector_demo.world
gazebo_ros_ray_sensor_demo.world
gazebo_ros_skid_steer_drive_demo.world
```

Figura 4.8: Lista de simulaciones de demostración para el uso de Gazebo con ROS 2

4. Compilar el paquete seleccionando el modelo de computadora de vuelo y el vehículo deseado.

```
$ ./waf configure --board CubeBlack
$ ./waf copter
```

Como comentario complementario, ArduPilot es compatible con varios modelos de computadoras de vuelo, en este caso se seleccionó una *Pixhawk2 Cube*. Para obtener el listado de todas las computadoras de vuelo compatibles, se puede ejecutar la siguiente instrucción; de tal forma que es posible seleccionar cualquier otro modelo cambiando el nombre del parámetro por cualquier de la lista.

```
$ ./waf list_boards
```

De igual manera, el parámetro de vehículo puede ser modificado por el nombre de otro de los vehículos con los que trabaja ArduPilot, acorde a las necesidades del usuario. Para enlistar los vehículos disponibles se puede utilizar el comando “*list*”.

```
$ ./waf list
```

5. Limpiar los archivos temporales generados tras la compilación.

```
$ ./waf clean
```

6. Añadir el API de ArduPilot al perfil de bash

```
$ echo "export PATH=$PATH:$HOME/ardupilot/Tools/autotest"  
      >> ~/.bashrc  
$ echo "export PATH=/usr/lib/ccache:$PATH" >> ~/.bashrc
```

7. Recargar el directorio de trabajo con el nuevo perfil de bash

```
$ . ~/.bashrc
```

Hecho lo anterior, se puede realizar la prueba del framework de SITL, para ello es necesario dirigirse al directorio del vehículo instalado, dentro del directorio donde se descargó ArduPilot.

```
$ cd ~/ardupilot/ArduCopter
```

Una vez dentro del directorio, se puede ejecutar la simulación del vehículo con el siguiente comando:

```
$ sim_vehicle.py --map --console
```

El comando anterior ejecuta una instancia del SITL de ArduPilot, de tal forma que se abren dos ventanas; un mapa una terminal.

La terminal que se abre al momento de ejecutar la simulación corresponde a la consola de vuelo, en esta se indican algunos parámetros de interés del dron, tal como el nivel de la batería, el modo de vuelo, la altura a la que se encuentra, un historial de eventos, entre otras cosas.

El mapa contiene un pequeño esquema de un cuadricóptero (vehículo compilado para este trabajo) y es donde se puede observar su desplazamiento con base en los comandos ingresados a partir de la terminal de ArduPilot.

Es posible controlar el dron utilizando comandos ingresados desde la terminal o directamente utilizando el mapa. Es posible asignar waypoints y rutas de vuelo de forma gráfica dando clic derecho sobre el mapa.

A continuación se adjuntan una serie de comandos ejemplo para realizar un desplazamiento básico, en donde el dron despega 10 m sobre el suelo y luego se mueve otros 20 m en el eje x.

Se debe de ingresar lo siguiente en la terminal desde donde se inició la sesión de ArduPilot:

```
> mode guided      #cambia el modo de vuelo  
> arm throttle    #arma los motores del dron  
> takeoff 10      #despegue  
> position 50 0 0 #desplazamiento en x,y,z
```

Lo anterior corresponde a una demostración del uso básico del simulador stand-alone de ArduPilot; sin embargo, en este trabajo se propone Gazebo como ambiente de simulación, por lo que es necesario conectar el SITL de Ardupilot con este simulador. Lo anterior es posible realizando la instalación de un plugin específicamente diseñado con este propósito.

Instalación de ArduPilot Gazebo plugin:

1. Dirigirse al directorio deseado para descargar al proyecto y clonar el repositorio de Github

```
$ git clone https://github.com/khancyr/ardupilot_gazebo  
$ cd ardupilot_gazebo
```

2. Compilar el proyecto

```
$ mkdir build  
$ cd build  
$ cmake ..  
$ make -j4  
$ sudo make install
```

3. Añadir la configuración de Gazebo al perfil de bash

```
$ echo 'source /usr/share/gazebo/setup.sh' >> ~/.bashrc  
$ echo 'export GAZEBO_MODEL_PATH=~/ardupilot_gazebo/models'  
      >> ~/.bashrc  
$ echo 'export GAZEBO_RESOURCE_PATH=~/ardupilot_gazebo/  
worlds:$GAZEBO_RESOURCE_PATH' >> ~/.bashrc
```

4. Recargar la ruta de trabajo con el nuevo perfil de bash

```
$ source ~/.bashrc
```

Capítulo 4. Resultados

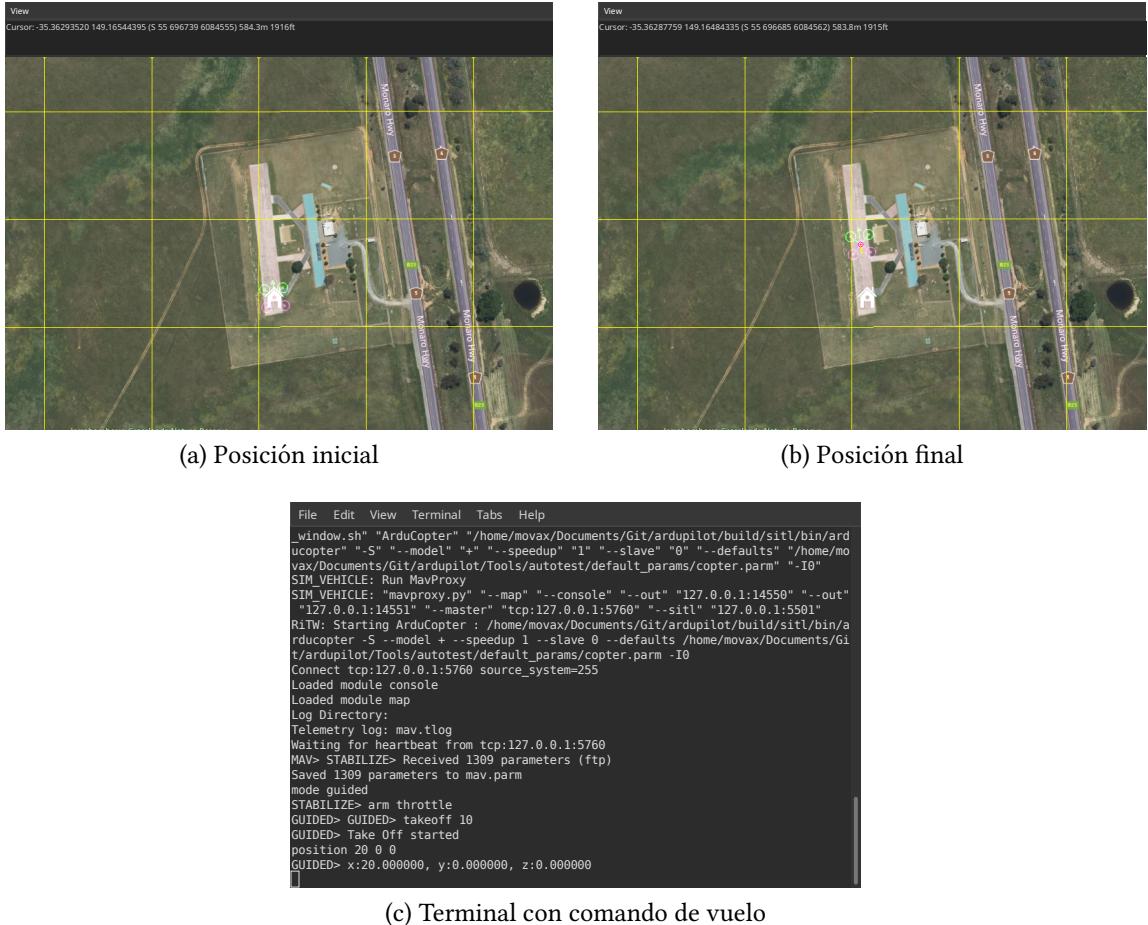
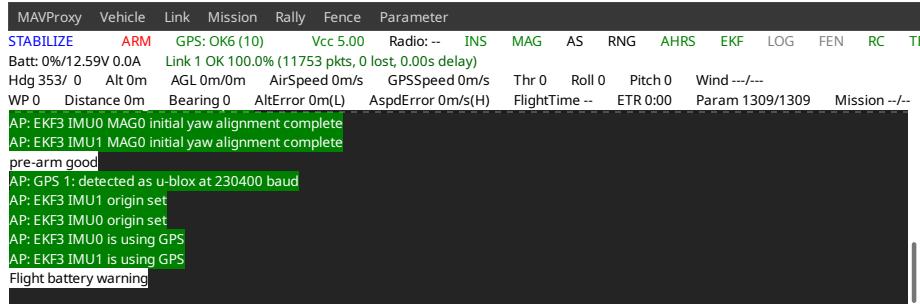


Figura 4.9: Prueba de ejecución del framework de SITL de ArduPilot

Con la configuración realizada hasta este punto, el sistema del usuario debe de ser capaz de iniciar una instancia de ArduPilot y conectarla con una simulación en Gazebo, de tal forma que los comando ingresados por medio de la terminal de ArduPilot tengan efecto dentro de la simulación de Gazebo.

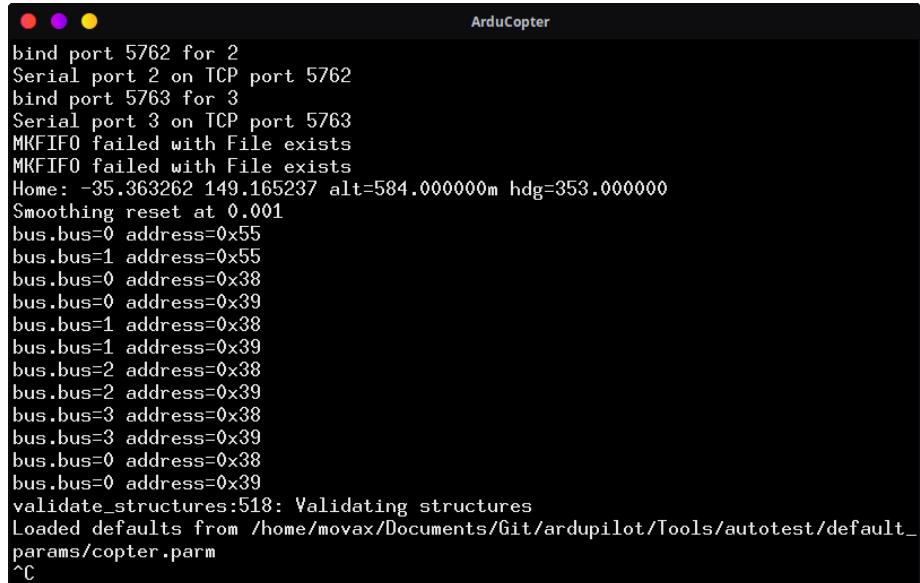
La instalación del plugin incluye una simulación de demostración para verificar la comunicación entre ambos programas; sin embargo, al momento de la escritura de este trabajo existe un bug al trabajar la simulación de prueba con Gazebo 11. La simulación de demostración integra un modelo 3D de un dron Iris, el cual viene configurado de tal manera que incluye la dinámica del dron y una serie de sensores simulados, entre ellos una cámara monocular; dicho lo anterior, el bug consiste en no permitir que se cargue el

Capítulo 4. Resultados



MAVProxy Vehicle Link Mission Rally Fence Parameter
STABILIZE ARM GPS: OK6 (10) Vcc 5.00 Radio: -- INS MAG AS RNG AHRS EKF LOG FEN RC TI
Batt: 0% / 12.59V 0.0A Link 1 OK 100.0% (11753 pkts, 0 lost, 0.00s delay)
Hdg 353/ 0 Alt 0m AGL 0m/0m AirSpeed 0m/s GPSSpeed 0m/s Thr 0 Roll 0 Pitch 0 Wind ---/---
WP 0 Distance 0m Bearing 0 AltError 0m(L) AspdError 0m/(H) FlightTime -- ETR 0:00 Param 1309/1309 Mission --/--
AP: EKF3 IMU0 MAG0 initial yaw alignment complete
AP: EKF3 IMU1 MAG0 initial yaw alignment complete
pre-arm good
AP: GPS 1: detected as u-blox at 230400 baud
AP: EKF3 IMU1 origin set
AP: EKF3 IMU0 origin set
AP: EKF3 IMU0 is using GPS
AP: EKF3 IMU1 is using GPS
Flight battery warning

Figura 4.10: Terminal de información del sistema de ArduPilot



bind port 5762 for 2
Serial port 2 on TCP port 5762
bind port 5763 for 3
Serial port 3 on TCP port 5763
MKIFO failed with File exists
MKIFO failed with File exists
Home: -35.363262 149.165237 alt=584.000000m hdg=353.000000
Smoothing reset at 0.001
bus.bus=0 address=0x55
bus.bus=1 address=0x55
bus.bus=0 address=0x38
bus.bus=0 address=0x39
bus.bus=1 address=0x38
bus.bus=1 address=0x39
bus.bus=2 address=0x38
bus.bus=2 address=0x39
bus.bus=3 address=0x38
bus.bus=3 address=0x39
bus.bus=0 address=0x38
bus.bus=0 address=0x39
validate_structures:518: Validating structures
Loaded defaults from /home/movax/Documents/Git/ardupilot/Tools/autotest/default_params/copter.parm
^C

Figura 4.11: Consola de comunicación de ArduPilot

modelo del dron dentro de la simulación, por lo que la comunicación entre los programas no se puede llevar a cabo.

Para corregir lo anterior, es necesario realizar una pequeña modificación dentro del archivo del modelo del dron:

1. Ir al directorio donde se encuentra el modelo del dron con el plugin de Ardupilot para Gazebo

```
$ cd /usr/share/gazebo-11/models/iris_with_ardupilot
```

2. Abrir el archivo *model.sdf* con nano con permisos de superusuario

```
$ sudo nano model.sdf
```

3. La segunda línea del archivo debe de contener lo siguiente:

```
<sdf version="1.7" xmlns:xacro='http://ros.org/wiki/xacro'>
```

4. Modificar la línea de código anterior de la siguiente manera

```
<sdf version="1.7">
```

5. Guardar los cambios y cerrar el archivo

Con la corrección anterior, el usuario debe de ser capaz de ejecutar la simulación de demostración en Gazebo que incluye el plugin de SITL de ArduPilot, de la siguiente manera:

1. Abrir 2 terminales

2. Ejecutar el SITL de ArduPilot en una de las terminales

```
cd ~/ardupilot/ArduCopter
```

```
sim_vehicle.py -f gazebo-iris --console
```

3. Ejecutar la simulación de Gazebo en la segunda terminal

```
gazebo --verbose worlds/iris_arducopter_runway.world
```

4. Esperar a que esté listo para recibir comandos de vuelo

5. Ejecutar los comandos de vuelo utilizados para validar la instalación del SITL de ArduPilot

Capítulo 4. Resultados

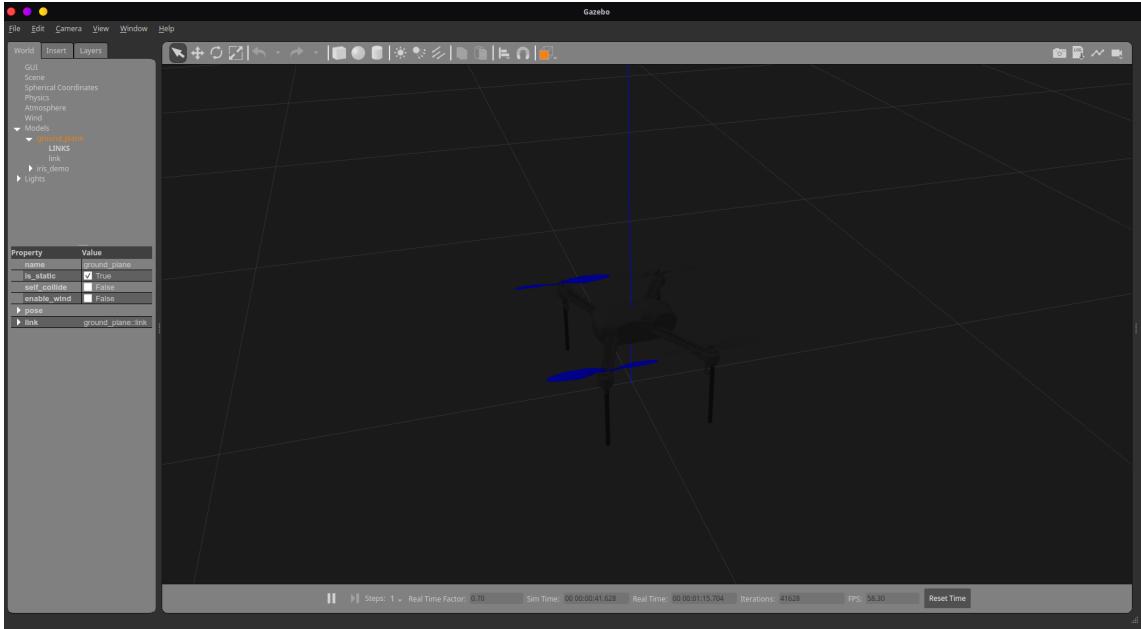


Figura 4.12: Modelo en Gazebo del dron Iris

4.1.5. Instalación de Pymavlink

Por último, en cuanto a la configuración del sistema, es necesario instalar Pymavlink para establecer la comunicación con el framework SITL de ArduPilot, de tal manera que se puedan enviar instrucciones de modos de vuelo, así como comandos que permitan definir la trayectoria de vuelo del dron, a través de un script en Python sin necesidad de ingresar estas instrucciones directamente en una terminal.

La librería Pymavlink está contenida en un módulo de Python, por lo que su instalación resulta un tanto trivial; sin embargo, cabe destacar que es necesario haber instalado el gestor de paquetes de Python 3 para ejecutar los siguientes comandos:

1. Actualizar los repositorios y paqueterías del sistema

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

2. Instala el módulo que contiene la librería

```
$ pip3 install mavproxy
```

Para verificar la instalación de la librería se puede abrir el intérprete de Python 3 en una terminal y transcribir la siguiente serie de comandos

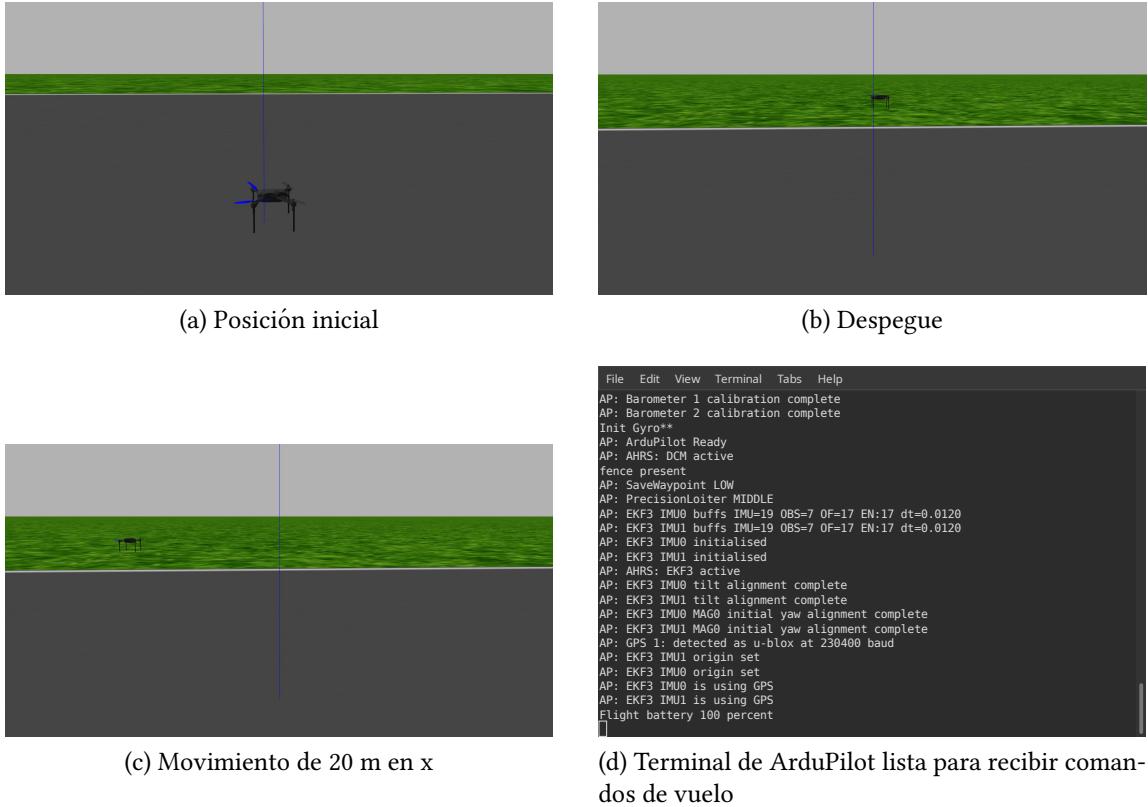


Figura 4.13: Prueba de integración entre Gazebo y el SITL de ArduPilot

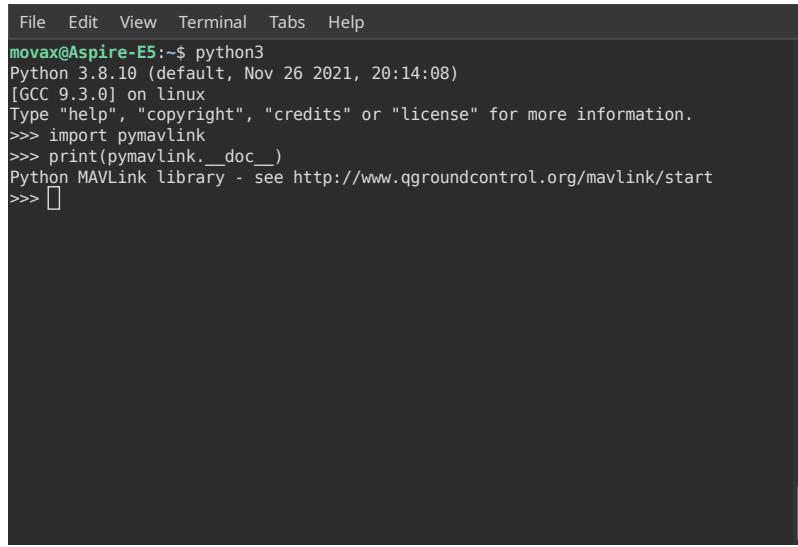
```
$ python3
>>> import pymavlink
>>> print(pymavlink.__doc__)
```

Al ejecutar lo anterior, se debe de generar una impresión en la terminal de la figura 4.14.

4.2. Circuito de Vuelo Virtual

La sección anterior corresponde a la configuración que se realizó para instalar y validar las herramientas que se utilizaron para la elaboración del proyecto. En los siguientes capítulos se detallan el uso específico que se le dio al software instalado, así como las pruebas y los resultados obtenidos; en esta sección se especifica el proceso de elaboración de la simulación con el circuito de vuelo para el dron simulado.

Capítulo 4. Resultados



```
File Edit View Terminal Tabs Help
movax@Aspire-E5:~$ python3
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pymavlink
>>> print(pymavlink.__doc__)
Python MAVLink library - see http://www.qgroundcontrol.org/mavlink/start
>>> 
```

Figura 4.14: Validación de instalación de Pymavlink

La figura 4.15 presenta un diagrama con en donde se especifica la estructura del circuito elaborado, en él se pueden apreciar las cotas con las distancias presentes entre cada una de las compuertas, así como el orden del recorrido realizado por el dron. Cabe destacar que el modelo de compuerta que se seleccionó para el circuito de vuelo, fue el utilizado en las distintas competencias del IROS.

A partir del esquema antes mencionado, la elaboración del circuito dentro de la simulación se llevó a cabo utilizando algunos modelos ya elaborados por la comunidad de Gazebo. Para el terreno y el modelo de la compuerta se utilizaron los modelos elaborados por (Rojas-Perez & Martinez-Carranza, 2020), los cuales fueron usados para entrenar su red neuronal profunda. Por otro lado, el modelo del dron Iris fue provisto por

Por otro lado, existe un bug al utilizar Gazebo, pues al crear mundo nuevo, no es posible guardar el proyecto con los cambios realizados, el menú de diálogo que aparece la opción de guarda proyecto simplemente no se muestra de forma correcta, por lo que crear un proyecto nuevo resulta imposible de esta forma. Debido o a lo anterior, en este trabajo se propone una solución para sobrellevar el problema anterior; para que el menú se muestre de forma correcta es necesario ejecutar Gazebo con permisos de administrador; sin embargo, al utilizar privilegios de administrador para crear el archivo del proyecto, este se encontrará protegido contra escritura, y resulta muy poco práctico necesitar permisos de administrador para realizar modificaciones sobre el proyecto.

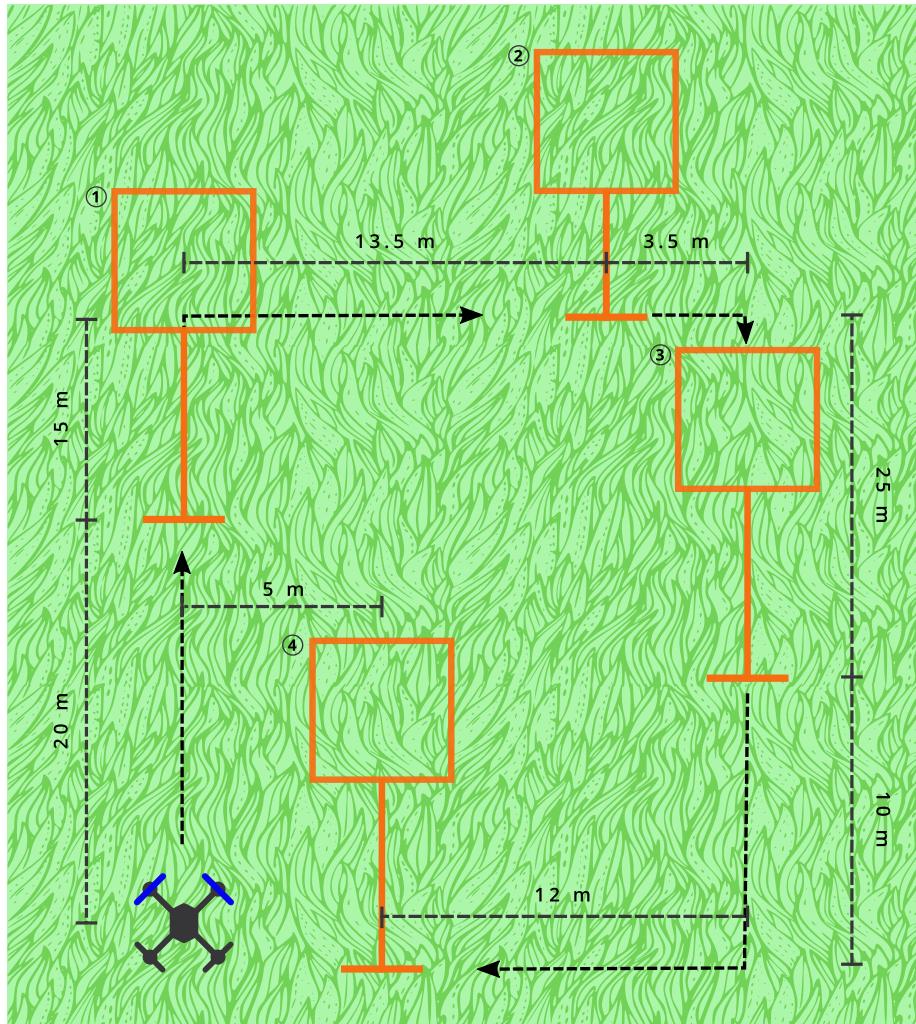


Figura 4.15: Esquema detallado del circuito de vuelo elaborado en simulación.

Entonces, para solucionar este segundo problema se debe de crear un archivo utilizando el comando *touch* con la extensión *.world*. Lo anterior genera un archivo un proyecto de Gazebo completamente vacío. Ahora, debido a que este tipo de archivos solamente contiene la descripción de los componentes utilizados en determinado proyecto, así como sus características físicas como posición, es posible abrir el archivo del proyecto con cualquier editor de texto y ver su contenido. Por lo tanto, se puede copiar el contenido del proyecto creado con privilegios de administrador y pegarlo dentro del nuevo proyecto vacío que se acaba de crear.

Hecho lo anterior, lo que queda es eliminar el proyecto protegido contra escritura.

Para ello es necesario abrir una terminal dentro del directorio donde se encuentra el proyecto y ejecutar el comando *rm* con permisos de administrador. A continuación se anexa un ejemplo:

```
$ sudo rm projectName.world
```

La figura 4.16 muestra los modelos 3D utilizados para la elaboración de la simulación; como se mencionó, se ocuparon dos tamaños de compuerta y por lo tanto dos modelos.

Adicionalmente, la figura 4.17 presenta el modelo 3D del dron iris utilizado. Cabe destacar que la principal diferencia entre el modelo provisto por (Johnson, 2018) y el modelo incluido en la simulación de demostración del plugin de Ardupilot para Gazebo, es la orientación de la cámara. En la simulación de demostración, la cámara se encuentra apuntando hacia el suelo, mientras que en el modelo mostrado en la figura, apunta hacia al frente de la dirección de vuelo del dron.

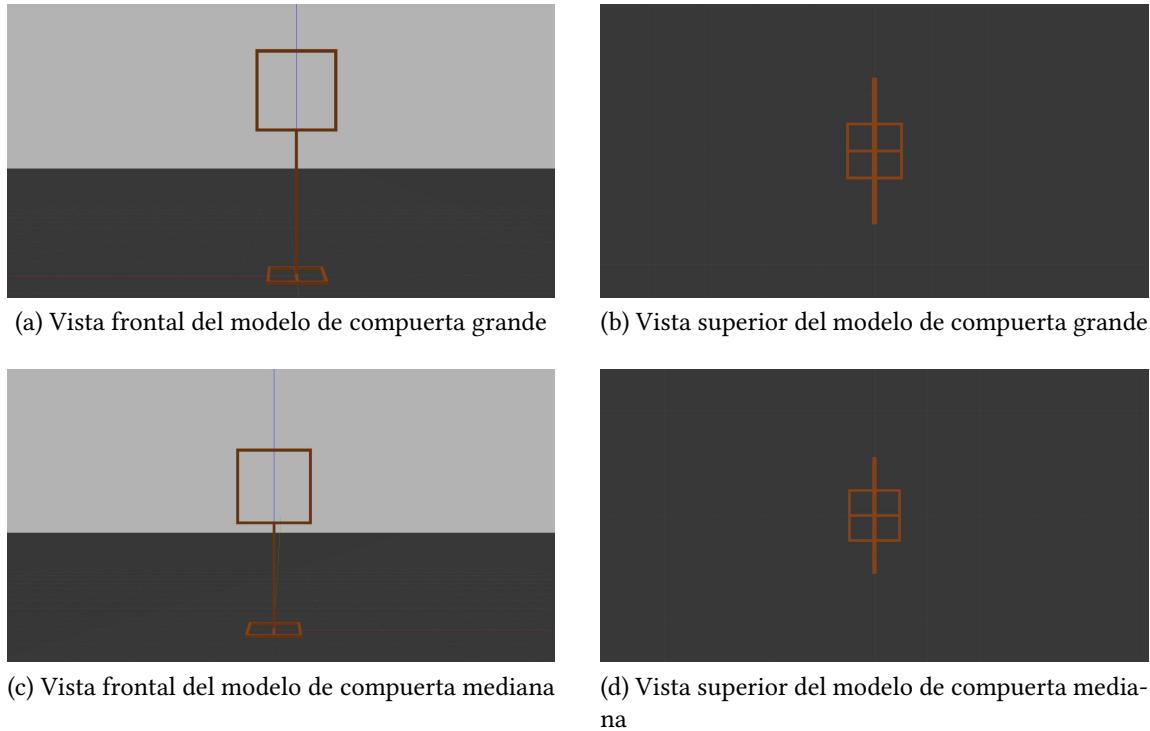


Figura 4.16: Modelos de compuerta proveídos por (Rojas-Perez & Martinez-Carranza, 2020)

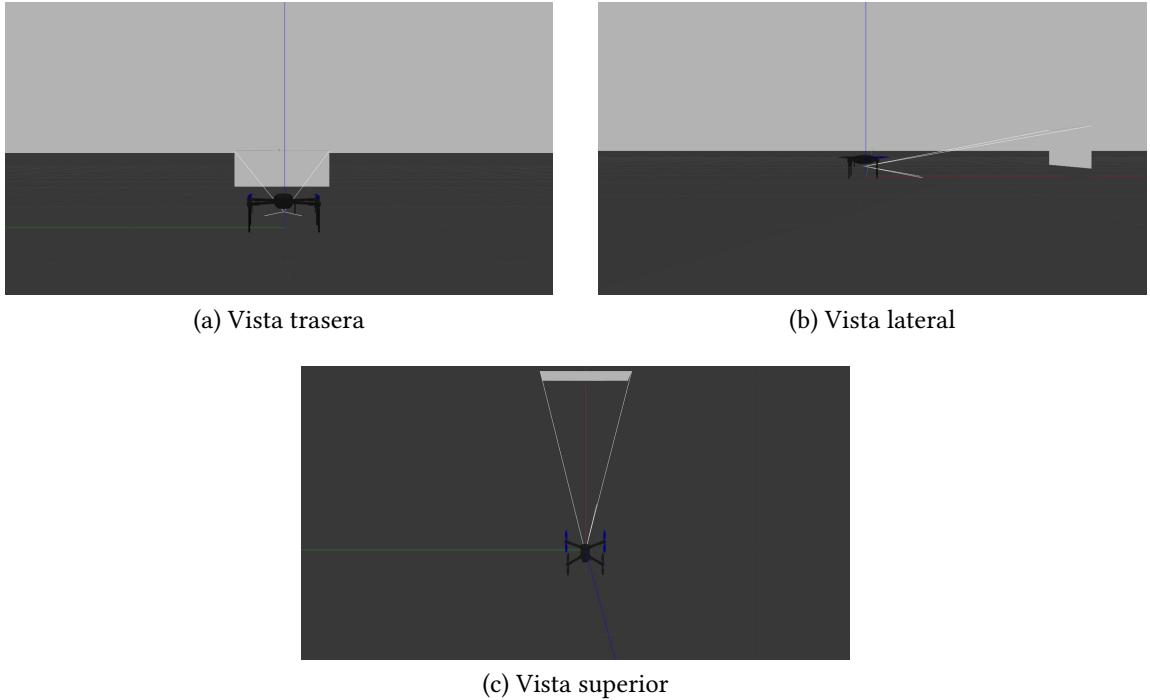


Figura 4.17: Modelo de dron iris con cámara frontal provisto por Johnson (2018)

Por último, la figura 4.18 muestra algunas capturas tomadas dentro del proyecto creado para implementar el circuito de vuelo implementado. La figura 4.18a presenta un panorama general del circuito de vuelo, en ella se observan las cuatro compuertas del circuito, el dron iris y el modelo de un edificio; este último se incluyó para fines del algoritmo de visión artificial. Por otro lado, las figura 4.18b y 4.18c muestran otras perspectivas del circuito y de los objetos que lo componen. Finalmente, la figura 4.18d muestra la perspectiva del dron Iris junto con la toma captada por la cámara simulada en este.

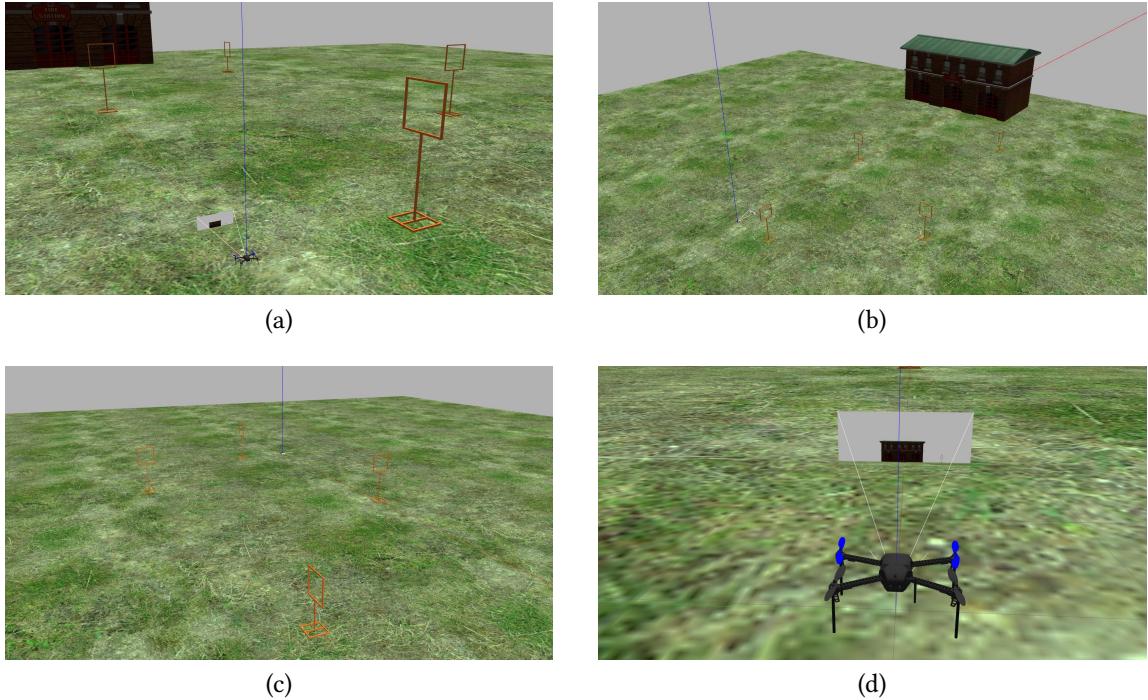


Figura 4.18: Capturas del ambiente de simulación implementado

4.3. Sistema de Visión Artificial

4.3.1. Descripción del sistema y forma de trabajo

En esta sección se describe el procedimiento realizado para llevar a cabo la adecuación del algoritmo de visión artificial para la detección de compuertas dentro del circuito de vuelo. Cabe mencionar que, como se mostró en la sección anterior, el tipo de compuertas seleccionadas para la formación del circuito de vuelo fue un modelo similar al utilizado en las competencias del IROS, con un característico color naranja. Ahora bien, debido a que el algoritmo de visión artificial está basado en la detección de color y no utiliza redes neuronales el aspecto más importante para adecuar el algoritmo es encontrar el rango de color en la escala HSV para nuestro objeto, o mejor dicho el color de nuestro objeto.

Entonces, para lograr sintonizar lo mejor posible la escala de color, se utilizó como base un rango de color adecuado para detectar el color naranja, pues este es el color principal del objeto de interés; sin embargo, el color de las compuertas es una de la infinidad

de tonalidades derivadas del naranja, por lo que fue necesario ajustar más el rango para que, de ser posible, el algoritmo fuera capaz de detectar exclusivamente las compuertas y no diera falsos positivos con objetos con una tonalidad derivada del naranja. Por otro lado, dentro del circuito de vuelo el único objeto con una tonalidad similar al color naranja es el edificio que se encuentra detrás de la primera compuerta, por lo que, se están asumiendo condiciones prácticamente ideales para la detección de las compuertas.

Dicho lo anterior, la figura 4.19 corresponde al mapa de color utilizado para definir el rango base para el algoritmo de detección. El diagrama está compuesto por 3 partes; el eje x corresponde al rango de valores para el matiz (hue), por otro lado, la primera parte del eje y , denotado con un (1), pertenece a los valores de matiz con una saturación que varía entre 0 y 255, así mismo, la segunda parte del eje y , identificada con un (2), denota los valores de matiz para los cuales los coeficientes de saturación y valor son iguales a 255. Entonces, para obtener la escala para la detección de un color, se debe escoger el valor de matiz y saturación correspondiente, y posteriormente establecer el coeficiente de valor entre 25 y 255.

Por lo tanto, a partir de la figura 4.19 la escala base seleccionada para la detección del color naranja fue:

$$H : 5 - 25; \quad S : 75 - 255; \quad V : 25 - 255$$

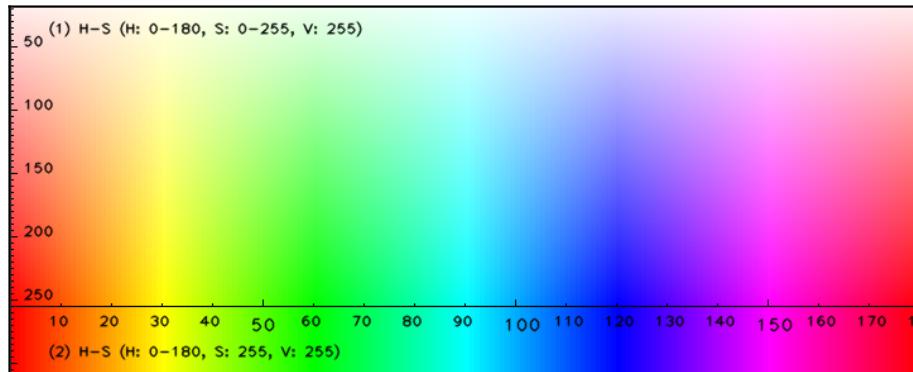


Figura 4.19: Mapa de color en escala HSV (Dey, 2020)

4.3.2. Sintonización de la escala de color

La sub-sección anterior representa el argumento para la selección de la escala inicial, ahora bien, como se mencionó al inicio de esta sección, fue necesario ajustar el rango de color para proveer al algoritmo de mayor precisión y robustez al momento de detectar las compuertas del circuito de vuelo. Lo anterior se logró utilizando imágenes del ambiente de simulación, de tal manera que se aplicó una máscara con el rango de color especificado y se fueron variando los valores de las componentes del modelo de color hasta observar que el algoritmo aislabía de manera satisfactoria las compuertas del resto de objetos en el circuito. La figura 4.20 muestra las imágenes utilizadas con este fin. La figura 4.20a corresponde a una fotografía obtenida a partir de la cámara simulada a bordo del dron, se trata de un primer plano hecho a una de las compuertas del circuito de vuelo, además, la figura 4.20b corresponde a una toma captada en tercera persona, en donde hay dos compuertas visibles y el dron intenta cruzar a través de una de ellas, incluso puede notarse el fotograma captado por la cámara del dron. Por otro lado, la figura 4.20c presenta otra toma en tercera persona, en donde es visible una compuerta y el dron se encuentra estático en el suelo, en esta fotografía se destaca el uso de un ambiente diferente al ambiente construido para el circuito de vuelo; por último, la figura 4.20d presenta una toma del circuito de vuelo en donde se alzan a apreciar dos compuertas y el único edificio incluido en el ambiente de simulación.

Cabe destacar que con las figuras 4.20c y 4.20d se buscó darle mayor robustez al algoritmo de detección de compuertas, utilizando otro tipo de ambiente y objetos de otros colores para realizar una calibración más adecuada del rango de color utilizado.

A partir de lo ya mencionado, para realizar la sintonización del rango de color se elaboró un script de Python, en donde se hizo uso de OpenCV para cargar las imágenes mostradas en la figura 4.20, convertir su modelo de color de BGR a HSV, y posteriormente ejecutar el algoritmo de detección con la escala de color base. Lo anterior se realizó de manera individual con cada una de las imágenes, de tal manera que con cada una se ajustó la escala base de forma gradual hasta observar una detección aceptable; es decir, hasta que la detección eliminara la mayor cantidad de falsos positivos en la imagen, dejando solamente las siluetas de las compuertas.

Es importante mencionar que en este punto del desarrollo del trabajo, el algoritmo no se encontraba ejecutándose de forma indefinida en algún proceso, sino que, se ejecu-

ta una sola vez sobre cada imagen, por lo que lo único que se necesitó para realizar la sintonización fueron las imágenes en cuestión para realizar el ajuste; es decir, el script elaborado no requiere de la ejecución del ambiente creado en Gazebo o de una instancia del SITL de ArduPilot para realizar el ajuste de la escala, este funciona de forma independiente. El proceso seguido para ejecutar el algoritmo desde un nodo en ROS se describe más adelante, en su respectiva sección.

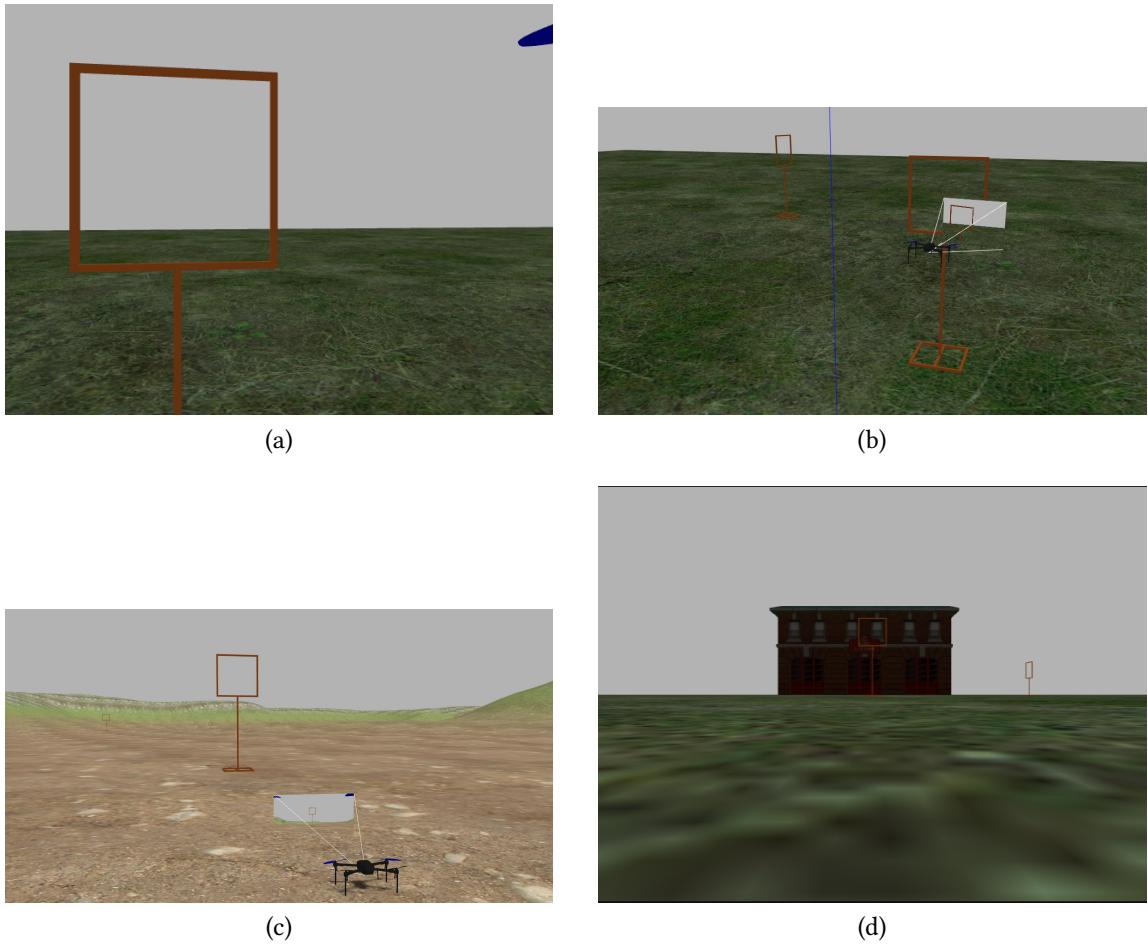


Figura 4.20: Imágenes utilizadas para la sintonización del rango de color

Entonces, a partir de la escala base se realizó la primera prueba del algoritmo de detección sobre la colección de imágenes. La figura 4.21 muestra los resultados obtenidos en esta prueba. Tomando en cuenta el contexto anterior, la figura 4.21a muestra la detección efectuada en la imagen en primer plano de una de las compuertas, como se puede

observar, la detección es prácticamente perfecta y se logra aislar con gran precisión la compuerta del fondo, el suelo y el fragmento visible de pala de uno de los rotores del dron. A manera de secuencia, la figura 4.21b presenta la fotografía en tercera persona de las dos compuertas, se aprecia que se logra aislar de gran manera el control de las compuertas, dejando a un lado el suelo y el dron; sin embargo, el algoritmo no es capaz de detectar el fragmento de compuerta que se observa en la previsualización de la toma de la cámara del dron. Después, la figura 4.21c presenta el comportamiento que se presentó sobre la imagen con cambio de ambiente, es posible observar que en este caso la detección es menos exacta, pues si bien es posible apreciar el aislamiento de gran parte del contorno de la compuerta, el suelo del ambiente también forma parte de la detección, pues su tonalidad entra en el rango de color de la escala base. Por último, la figura 4.21d, presenta el efecto de la detección sobre la imagen con el edificio del ambiente de simulación en el fondo, en esta primera prueba, está último caso presenta el peor desempeño por parte del algoritmo de detección, pues no fue capaz de captar en absoluto las compuertas de la imagen, sino que, más bien aisló la fachada rojiza del edificio; estos dos últimos casos son un claro ejemplo del porqué fue necesario realizar ajustes en el rango de color utilizado.

Dándole continuidad a lo observado, realizando el ajuste del rango base, se llegó una serie de valores que mejoraron notablemente el desempeño del algoritmo de detección. El proceso para la sintonización fue el ya expresado, de tal manera que se jugó con el límite inferior de cada parámetro en el modelo de color HSV, hasta que se observó una mejora en la detección de cada una de las imágenes, principalmente en las últimas dos, pues son las que presentaron los resultados más paupérrimos en las primeras pruebas. Entonces, el rango de color que se obtuvo con el reajuste es el siguiente:

$$H : 5 - 25; \quad S : 99 - 255; \quad V : 77 - 171$$

Se observa que el rango para el matiz permaneció igual, mientras que los parámetros de saturación y valor percibieron un tanto significativo. La figura 4.22 muestra los resultados obtenidos a partir de sintonización realizada, a grandes rasgos presenta el mismo número de imágenes que en la prueba anterior, y es posible observar que la detección para la figura 4.22a y 4.22b permaneció prácticamente igual, aun que no del todo. Realmente el desempeño logrado en la figura 4.22a fue bastante bueno, por lo que en este caso no se observa ninguna mejora en particular, por otro lado, en cuanto a la figura

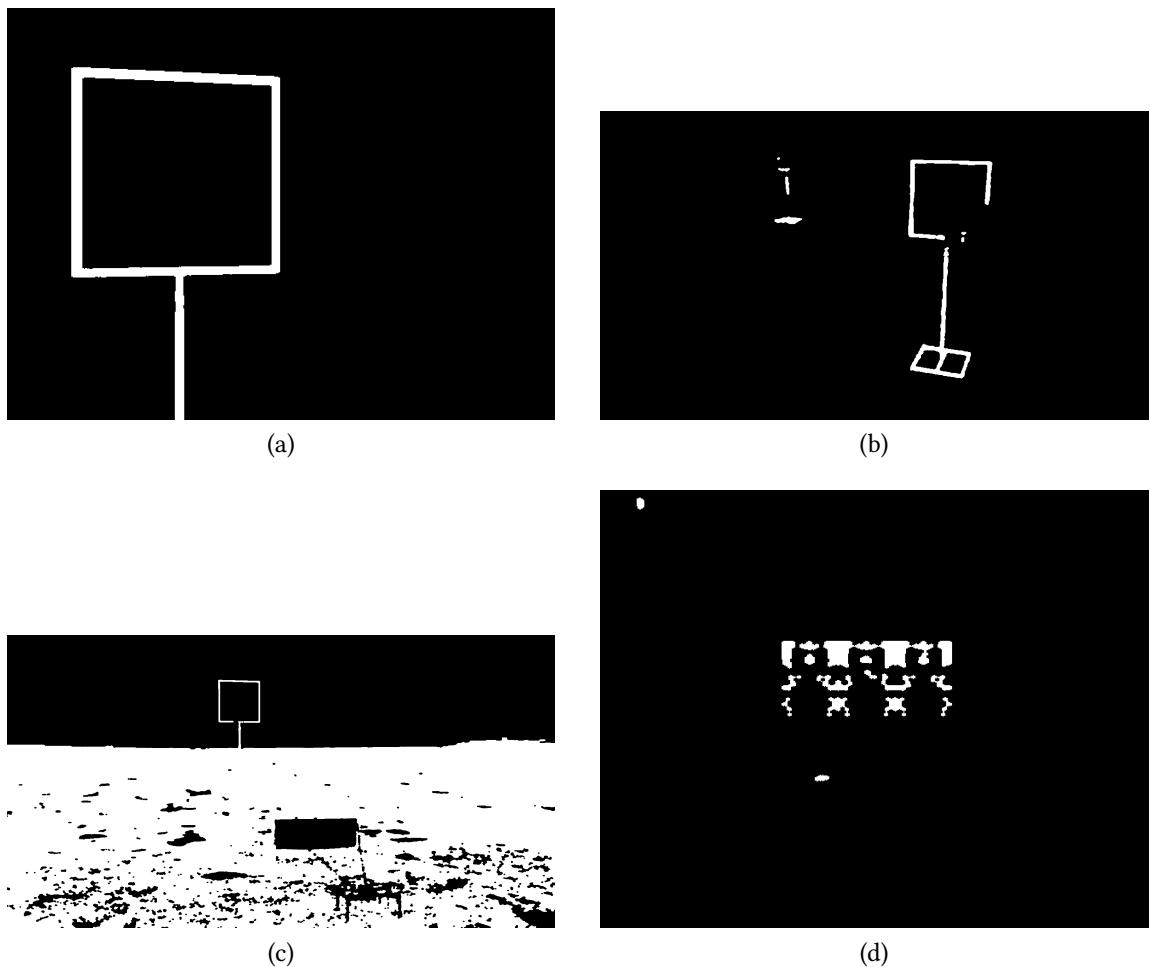


Figura 4.21: Detección de compuerta con el primer rango de color

4.22b, es posible observar que el reajuste provocó la eliminación de algunas secciones de la compuerta más cercana al dron, y en cuanto a segunda compuerta de la imagen, se observa una perdida bastante significativa en cuanto a la detección; sin embargo, esto no representa un comportamiento no deseado, del todo, pues tomando en cuenta el objetivo del algoritmo de detección y la ruta de vuelo seguida por el dron, se puede llegar a la conclusión de que realmente en ningún momento del vuelo el dron tendrá en su campo de visión a más de una compuerta a la vez, y si lo llega a hacer, lo que se buscó es que el dron fuera capaz de detectar la compuerta más cercana a él, por lo que las compuertas del fondo pierden cierto grado de relevancia para esta aplicación en particular.

En contraste con el caso anterior, el desempeño referente a la figura 4.22c presenta

una gran mejora con respecto a la primera prueba realizada, a tal grado que el porcentaje de falsos positivos obtenidos por la tonalidad del suelo resulta ser mucho menor, y en este caso ya es posible observar en mayor proporción el contorno y superficie de la compuerta. Por otro lado, la detección efectuada en la figura 4.22d sigue siendo bastante pobre, y al realizar el reajuste de la escala de color, ya no fue posible detectar ningún objeto dentro de esta imagen, por lo que la imagen resultante se encuentra completamente en negro.

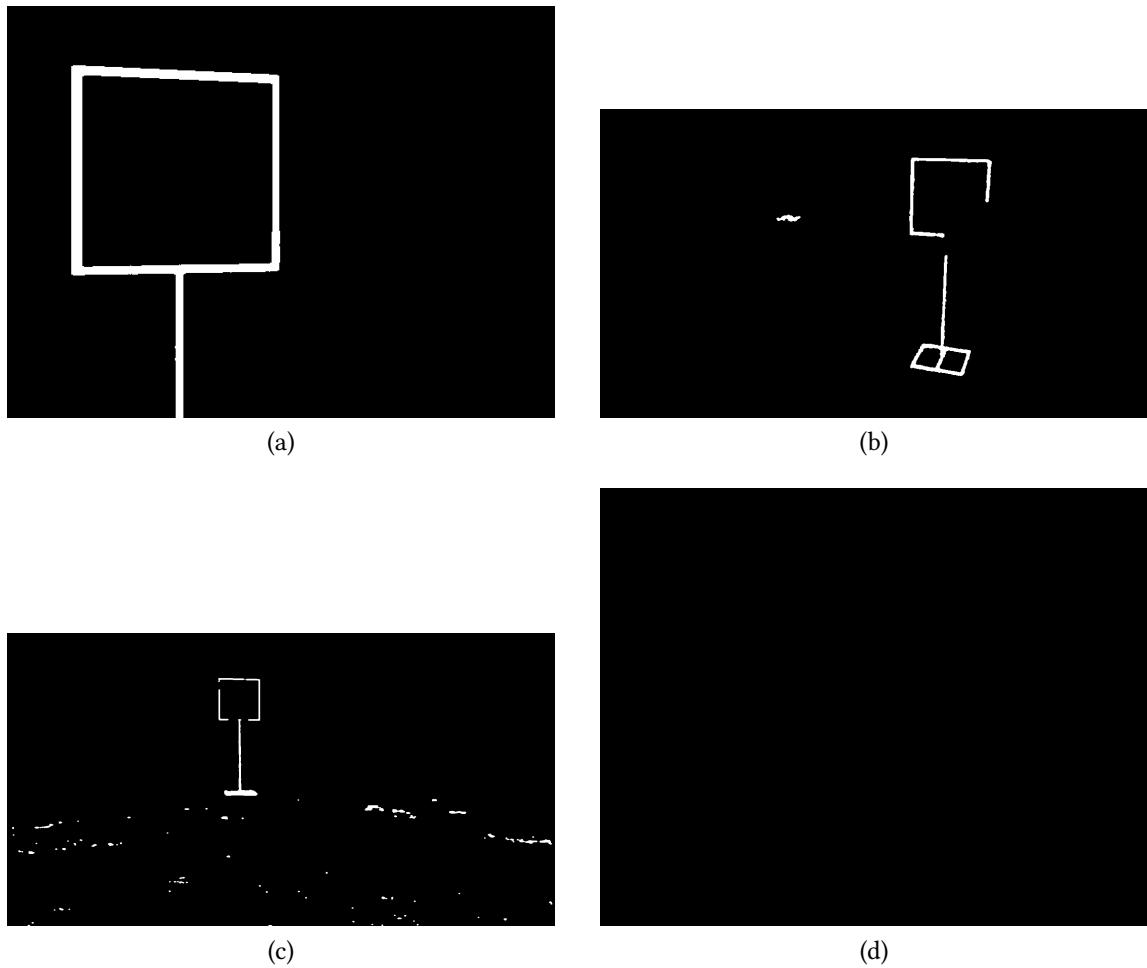


Figura 4.22: Sintonización del rango de color

Recapitulando un poco lo ya comentado, el reajuste en la escala de color permitió obtener una mejora bastante significativa para la reducción de falsos positivos en la detección de las compuertas; sin embargo, contrario o lo que se esperaba, el desempeño que se obtuvo en la última imagen resultó ser inferior al de la primera prueba. Entonces, se

buscó realizar una segunda sintonización con el objetivo de mejorar el desempeño en esta última imagen, por lo que ya no se realizaron más pruebas con las tres primeras imágenes, y el reajuste se realizó únicamente experimentando con esta última imagen. Sin embargo, pese a los esfuerzos realizados, no se logró encontrar un rango de valores que mejoraran la detección, simplemente el algoritmo no fue capaz de detectar la compuerta.

Lo anterior se atribuye a que, debido a la distancia con la que fue tomada la imagen, la proporción de superficie de las compuertas en la imagen es bastante reducida, por lo que el algoritmo no es capaz de aislar de manera satisfactoria el color que se busca. La figura 4.23 presenta los resultados obtenidos con este último intento de resintonización. En la figura 4.23b es posible observar el desempeño de la detección en la imagen objetivo, se puede apreciar que solamente fue posible mejorar el aislamiento de la fachada del edificio, dando como resultado el siguiente rango de parámetros:

$$H : 0 - 25; S : 45 - 217; V : 0 - 44$$

Entonces, se utilizó esta última escala de color para ejecutar la detección en el resto de fotografías, y lo que se obtuvo fue algo similar a lo observado en la figura 4.23a. Esta figura corresponde solamente a la detección efectuada sobre la toma en primer plano de la compuerta, sin embargo, el resultado obtenido en las otras imágenes fue el mismo, un recuadro completamente negro.

Finalmente, está claro que este último rango de parámetros no es para nada eficiente y no cumple con el objetivo de la detección de compuertas, por lo que al final se descartó y se utilizó la escala obtenida a partir del primer ajuste para su implementación en ROS.

4.3.3. Algoritmo de visión artificial

En la sub-sección anterior se describió el proceso implementado para adecuar el algoritmo de detección con base en los requerimientos mencionados. Ahora, esta última sub-sección tiene como objetivo definir la lógica secuencial utilizada, así como las funciones que componen el algoritmo implementado.

La estructura de la lógica implementada puede observarse en el algoritmo 1

Como se puede observar, es un algoritmo bastante sencillo. Ahora bien, como se mencionó en el marco teórico, OpenCV ofrece una gran variedad de funciones y herramientas



Figura 4.23: Intento de sintonización utilizando la cuarta imagen de referencia como base

Algorithm 1 Metodología para la detección de compuertas

1. **Adquirir** imagen
 2. **Convertir** espacio de color: RGB → BGR
 3. **Convertir** espacio de color: BGR → HSV
 4. **Aplicar** el método del valor umbral sobre la imagen
 5. **Aplicar** una operación de apertura sobre la imagen umbralizada
 6. **Aplicar** una operación de cerradura sobre la imagen umbralizada
 6. **Mostrar** la imagen original y la imagen procesada
-

para la ejecución de algoritmos de visión artificial, por lo que el programa resultante es bastante compacto en cuanto a líneas de código, pues se utilizaron funciones nativas de la librería. El código desarrollado para esta etapa puede ser consultado en el repositorio oficial del presente trabajo Ramírez Linarez (2022), dentro del directorio de recursos.

A continuación se describen las funciones utilizadas para la implementación del algoritmo.

imread: recibe como argumentos el directorio de la imagen que se desea procesar y, de forma opcional, una bandera que indica algunos perfiles de color con los que puede ser leída la imagen. En este caso, solo se indicó el directorio para cargar cada imagen de forma individual en cada prueba. Cabe mencionar que OpenCV almacena las imágenes en una serie de arreglos multidimensionales, que dependiendo del perfil y el espacio de color, pueden ser manipulados a nivel de píxeles o capas.

cvtColor: es el método utilizado para realizar la conversión entre modelos de color; OpenCV es capaz de manejar distintos espacios de color, en el caso de este trabajo, primero se realizó la conversión del espacio RGB a BGR y después del BGR al HSV.

inRange: ejecuta el método del valor umbral. Recibe como parámetros la imagen que se desea procesar, y el límite inferior y superior del arreglo de valores HSV; este método crea una máscara con base en el rango de color, la cual es utilizada para aplicar las operaciones morfológicas necesarias y realizar el aislamiento del objeto que se desea detectar.

erode: ejecuta la operación morfológica de erosión, recibe la variable donde se almacena la imagen procesada, la imagen a procesar y el núcleo para llevar a cabo la operación.

dilate: de forma similar a la función anterior, ejecuta la operación de dilación y recibe los mismos parámetros que la función de erosión.

imshow: despliega una ventana con una imagen indicada por el usuario. Recibe como parámetros el nombre de la ventana y la variable que almacena la imagen que se desea visualizar.

Por último, es importante mencionar que la operación de apertura, en el contexto del procesamiento de imágenes, consiste en ejecutar una operación de erosión seguida de una operación de dilación; por otro lado, para realizar una operación de cerradura, primero se ejecuta la erosión y después la dilación. La apertura sirve para remover los objetos pequeños del fondo, mientras que la cerradura ayuda a llenar los huecos creados a partir del método del valor umbral, en el objeto que se desea detectar.

Lo anterior concluye la sección dedicada a la descripción del algoritmo de visión artificial implementado.

4.4. Misión de Vuelo

En esta sección se describe el proceso de implementación de los comandos de vuelo y el algoritmo de seguimiento de trayectoria para el vuelo a través del circuito; cabe mencionar que se siguió un paradigma similar al de la sección anterior, primero se trabajó con scripts individuales de Python, y una vez que se validó su funcionamiento, se creó su respectivo nodo en ROS; los resultados correspondientes a este último punto, se tratan a detalle en el capítulo dedicado a ROS.

Dicho lo anterior, el algoritmo de seguimiento de trayectoria se trabajó en pequeñas etapas, en donde se probaba una funcionalidad distinta del piloto automático, de tal forma que cada etapa integra a la anterior, y así de forma sucesiva hasta que se obtuvo el script con el algoritmo de misión de vuelo. Los códigos fuente pertenecientes a cada etapa se encuentran en el respectivo repositorio de GitHub, perteneciente al presente trabajo. La tabla 4.2 muestra la relación que existe entre cada etapa y su script en el repositorio de GitHub.

	Etapa	Código fuente
1	Establecimiento de la conexión entre Pymavlink y ArduPilot	listen.py
2	Configuración del modo de vuelo del dron	mode.py
3	Armado de motores y despegue	takeoff.py
4	Seguimiento de trayectoria con waypoints	movement.py
5	Implementación de la misión de vuelo	mission.py

Tabla 4.2: Relación entre etapa y su código fuente

A continuación se muestran los resultados obtenidos en cada una de las etapas mencionadas.

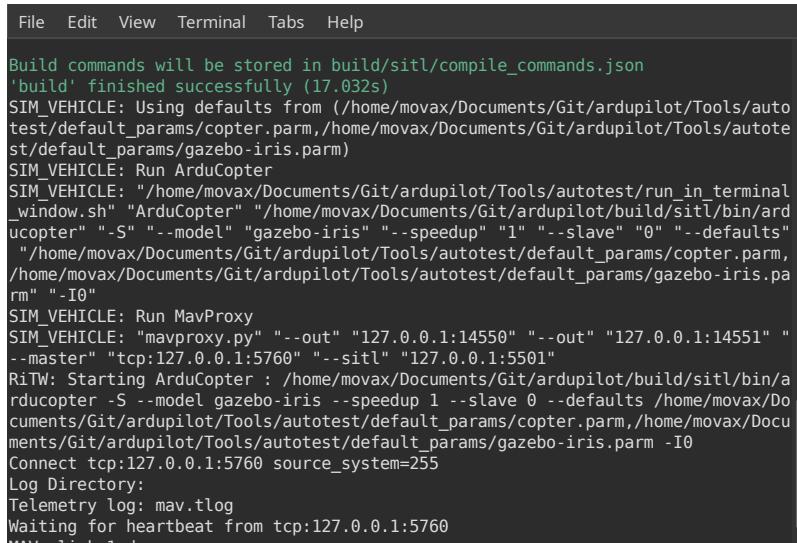
4.4.1. Conexión entre Pymavlink y ArduPilot

Inevitablemente, lo primero que se realizó fue comprobar la comunicación entre el script de Pymavlink y el SITL de ArduPilot. Por esta razón, el script perteneciente a esta etapa es bastante sencillo; sin embargo, representa la base de las subsecciones posteriores.

A grandes rasgos, lo más destacable es la inclusión de la librería Pymavlink y la creación de un objeto que sirve como intermediario entre el script y el piloto automático simulado. Dicho esto, para crear el objeto es necesario especificar la dirección que se utilizará para establecer la conexión; esta se especifica al momento de iniciar el SITL de ArduPilot en un terminal. Además, ArduPilot provee dos direcciones para realizar la conexión. En este caso, se escogió la 14551 de forma arbitraria, sin ninguna razón en específico. La figura 4.24 muestra la información generada por ArduPilot durante el inicio de una instancia del SITL, a simple vista es mucha información; sin embargo, los datos de interés para esta subsección se encuentran en la línea en donde se indican las direcciones disponibles para recibir los datos manejados por el piloto automático, al observar la figura de forma

Capítulo 4. Resultados

detenida es posible percibirse que hay dos direcciones denotadas por la palabra *out*, la 127.0.0.1:14550 y la 127.0.0.1:14551, en donde los últimos 5 dígitos indican la dirección que se tiene que especificar en el script de Pymavlink para establecer comunicación con ArduPilot.



The screenshot shows a terminal window with the following text output:

```
File Edit View Terminal Tabs Help
Build commands will be stored in build/sitl/compile_commands.json
'build' finished successfully (17.032s)
SIM_VEHICLE: Using defaults from (/home/movax/Documents/Git/ardupilot/Tools/autotest/default_params/copter.parm,/home/movax/Documents/Git/ardupilot/Tools/autotest/default_params/gazebo-iris.parm)
SIM_VEHICLE: Run ArduCopter
SIM_VEHICLE: "/home/movax/Documents/Git/ardupilot/Tools/autotest/run_in_terminal_window.sh" "ArduCopter" "/home/movax/Documents/Git/ardupilot/build/sitl/bin/arducopter" "-S" "--model" "gazebo-iris" "--speedup" "1" "--slave" "0" "--defaults" "/home/movax/Documents/Git/ardupilot/Tools/autotest/default_params/copter.parm,/home/movax/Documents/Git/ardupilot/Tools/autotest/default_params/gazebo-iris.parm" "-IO"
SIM_VEHICLE: Run MavProxy
SIM_VEHICLE: "mavproxy.py" "--out" "127.0.0.1:14550" "--out" "127.0.0.1:14551" "--master" "tcp:127.0.0.1:5760" "--sitl" "127.0.0.1:5501"
RiTW: Starting ArduCopter : /home/movax/Documents/Git/ardupilot/build/sitl/bin/arducopter -S --model gazebo-iris --speedup 1 --slave 0 --defaults /home/movax/Documents/Git/ardupilot/Tools/autotest/default_params/copter.parm,/home/movax/Documents/Git/ardupilot/Tools/autotest/default_params/gazebo-iris.parm -IO
Connect tcp:127.0.0.1:5760 source_system=255
Log Directory:
Telemetry log: mav.tlog
Waiting for heartbeat from tcp:127.0.0.1:5760
```

Figura 4.24: Datos de conexión de ArduPilot

Una vez que se estableció la conexión entre ambas partes es posible tener acceso a toda clase de datos de vuelo, entre ellos la actitud del dron. Cabe mencionar que, en los mensajes que recibe Pymavlink por parte de ArduPilot es una trama de datos de gran longitud, y debido a que se creó un objeto de Python para establecer la conexión, es posible acceder a un dato en específico como se accede a un atributo en la instancia de una clase. En este caso, se decidió imprimir en consola el dato del ángulo de roll del dron con el fin de comprobar que la comunicación se dio de forma exitosa.

Por último con respecto a esta subsección, la figura 4.25 muestra el resultado de haber establecido una comunicación exitosa entre Pymavlink y el piloto automático, en la figura 4.25a se logra apreciar la trama completa con los datos referentes a la actitud del dron al momento de llevar a cabo la comunicación, mientras que la figura 4.25b presenta esta trama de forma filtrada, de tal manera que solo se imprime el ángulo de roll del dron.

Capítulo 4. Resultados

```

File Edit View Terminal Tabs Help
ATTITUDE {time_boot_ms : 26572, roll : 0.001770165393813777, pitch : 0.001772125514879227, rollspeed : 0.00029608929576598, pitchspeed : 0.0002754572809977154, yawspeed : 0.001189235350182486}
ATTITUDE {time_boot_ms : 266023, roll : 0.0017786281295120716, pitch : 0.001773073356251636468, yaw : 0.0177307345173476277054, pitchspeed : 0.0002690583837518573, yawspeed : 0.001226210950612686}
ATTITUDE {time_boot_ms : 267075, roll : 0.00128446281295120716, pitch : 0.0010774225229397146, yaw : 0.0177266145467758, rollspeed : 0.00037381307013116476, pitchspeed : 0.00020951544856536843, yawspeed : 0.001173458527773187}
ATTITUDE {time_boot_ms : 267325, roll : 0.001280891930254449964, pitch : 0.001746699998704659, yaw : 0.01767144352197647, rollspeed : 0.00037101545132998, pitchspeed : 0.000295556931540038, yawspeed : 0.0011867942118644714}
ATTITUDE {time_boot_ms : 267446, roll : 0.001289063537832477, pitch : 0.000135274206193, yaw : 0.01762446800666809, rollspeed : 0.000308102654961443, pitchspeed : 0.0003027680950565657, yawspeed : 0.0011555940789923}
ATTITUDE {time_boot_ms : 267823, roll : 0.00128993537832477, pitch : 0.000108722887459072, yaw : 0.017610162622210995, rollspeed : 0.0003649964392659684, pitchspeed : 0.0003122246125712991, yawspeed : 0.0011866494980821569}
ATTITUDE {time_boot_ms : 268073, roll : 0.0012868151534348726, pitch : 0.001087072887459072, yaw : 0.017610162622210995, rollspeed : 0.0003649964392659684, pitchspeed : 0.0003122246125712991, yawspeed : 0.0011866494980821569}
ATTITUDE {time_boot_ms : 268133, roll : 0.0012868151534348726, pitch : 0.001087072887459072, yaw : 0.017610162622210995, rollspeed : 0.0003649964392659684, pitchspeed : 0.0003122246125712991, yawspeed : 0.0011866494980821569}
ATTITUDE {time_boot_ms : 268252, roll : 0.0012868151534348726, pitch : 0.001087072887459072, yaw : 0.017610162622210995, rollspeed : 0.0003649964392659684, pitchspeed : 0.0003122246125712991, yawspeed : 0.0011866494980821569}
ATTITUDE {time_boot_ms : 26825, roll : 0.0012868151534348726, pitch : 0.0010857646063894, yaw : 0.01753657493765683, rollspeed : 0.000262185570016581, pitchspeed : 0.0003656083913968457, yawspeed : 0.0011836469639062864}
ATTITUDE {time_boot_ms : 269075, roll : 0.0013000076869502664, pitch : 0.00010896584443455973, yaw : 0.017512233687400818, rollspeed : 0.0003408535769252033, pitchspeed : 0.0002408535769252033, yawspeed : 0.0011292471317574802}
ATTITUDE {time_boot_ms : 269075, roll : 0.001301456498054727, pitch : 0.000108953544267968, yaw : 0.017512233687400818, rollspeed : 0.0003408535769252033, pitchspeed : 0.0002408535769252033, yawspeed : 0.0011292471317574802}
ATTITUDE {time_boot_ms : 269573, roll : 0.001301456498054727, pitch : 0.000108953544267968, yaw : 0.017480126490772166, rollspeed : 0.00037809336780807294, pitchspeed : 0.000287417984738412, yawspeed : 0.0011307694949209691}
ATTITUDE {time_boot_ms : 269823, roll : 0.001301456498054727, pitch : 0.000108953544267968, yaw : 0.017445532669051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.0011294613500013947}
ATTITUDE {time_boot_ms : 270073, roll : 0.001301456498054727, pitch : 0.000109415708303262, yaw : 0.017424197895712763, rollspeed : 0.000315895369598977, pitchspeed : 0.0002568648398298, yawspeed : 0.0011294613500013947}
ATTITUDE {time_boot_ms : 270253, roll : 0.001301456498054727, pitch : 0.000109415708303262, yaw : 0.017424197895712763, rollspeed : 0.000315895369598977, pitchspeed : 0.0002568648398298, yawspeed : 0.0011294613500013947}
ATTITUDE {time_boot_ms : 270575, roll : 0.001301456498054727, pitch : 0.000109415708303262, yaw : 0.017398574263101394, rollspeed : 0.00037321291142632737, pitchspeed : 0.0002559937426301394, yawspeed : 0.00117462219559984}
ATTITUDE {time_boot_ms : 270825, roll : 0.0013117379276808192, pitch : 0.00108097011376840281, yaw : 0.0173797681932788997, rollspeed : 0.0002941774210659266, pitchspeed : 0.0002262561140832305, yawspeed : 0.0011472579702927}
ATTITUDE {time_boot_ms : 271232, roll : 0.0013117379276808192, pitch : 0.00108097011376840281, yaw : 0.0173797681932788997, rollspeed : 0.0002941774210659266, pitchspeed : 0.0002262561140832305, yawspeed : 0.0011472579702927}
ATTITUDE {time_boot_ms : 271573, roll : 0.001315693021759403, pitch : 0.001088126996747475, yaw : 0.0173296364564822, rollspeed : 0.0003200183535369887, pitchspeed : 0.000260607731641379, yawspeed : 0.00121105626254715}
ATTITUDE {time_boot_ms : 271823, roll : 0.001319620582595992, pitch : 0.001088126996747475, yaw : 0.0173296364564822, rollspeed : 0.0003200183535369887, pitchspeed : 0.000260607731641379, yawspeed : 0.0011591115811794996}
ATTITUDE {time_boot_ms : 272073, roll : 0.0013271461180879318, pitch : 0.001088994742898944, yaw : 0.01732285207812495134, rollspeed : 0.00033839215864354715, yawspeed : 0.00110844717845501}

```

(a) Trama de datos de la actitud del dron

```

File Edit View Terminal Tabs Help
mavproxy@e5:~/pymavlinks$ python3 listen.py
Heartbeat from system (system 1 component 0)
0.00034893141128122807
0.000327271643680426614
0.00039273887351477444
0.00026138446077806769897
0.0002030306446077806769897
0.00024369728775496513
0.00021968634973745793
0.0001981121313293532
0.00018420109669945807
0.00016397135914363362
0.00016397135914363362
0.00012302232313790548
0.00010289751177831424
0.670622628415903e-05
0.916467827977613e-05
5.1426794731199e-05
2.7764397609549273e-05
1.843062856208562e-05
-3.332536632388e-06
-1.665580111546442e-05
-3.477477002888918e-05
-4.752289861180745e-05

```

(b) Datos del roll

Figura 4.25: Datos obtenidos a partir de la conexión entre ArduPilot y Pymavlink

4.4.2. Configuración de modo de vuelo

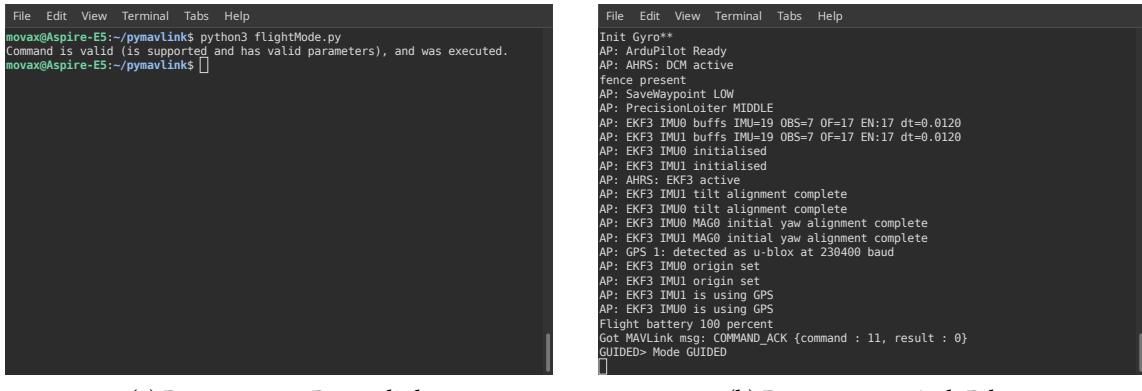
De acuerdo con la documentación oficial de ArduPilot (Johnson, 2022), el Arducopter cuenta con una gran variedad de modos de vuelo, no obstante, en este trabajo solo se utilizan únicamente dos modos, *guided* y *land*; el primero permite enviar comando de vuelo de forma directa al piloto automático a través de mensajes, mientras que el segundo ejecuta una rutina de aterrizaje para el dron. Este último modo de vuelo se aborda en la subsección marcada como extra.

Entonces, previo a la secuencia de despegue del dron, es necesario configurar el modo de vuelo en *guided*, pues al iniciar el SITL de ArduPilot, este inicializa con el modo de vuelo "stabilize", de tal forma que el piloto automático no es capaz de recibir comandos de vuelo ingresado directamente por el usuario. Esto representa un problema pues se busca que la trayectoria de vuelo del dron sea determinada a partir de un conjunto predefinido de waypoints.

Con base en lo anterior, el script perteneciente a esta subsección busca el identificador perteneciente al modo de vuelo indicado, dentro de una tabla de valores pre-configurados

dentro de la librería, y utiliza el objeto de la subsección anterior para enviar el comando de selección de modo de vuelo al piloto automático. En dado caso de que el modo de vuelo especificado tenga un registro válido dentro de la tabla de la librería, se imprime un mensaje en donde se confirma que el modo de vuelo ha sido cambiado.

Por último, el método utilizado para establecer el modo de vuelo fue *set_mode_send*, el cual recibe solamente 3 parámetros, el atributo del sistema del objeto de la conexión, el comando para establecer el modo de vuelo, y el código de identificación del modo de vuelo especificado. La figura 4.26 muestra el resultado que se obtuvo al ejecutar el código desarrollado para esta subsección, en la figura 4.26a se observa el mensaje de ejecución del script como tal, y por otro lado, la figura 4.26b presenta la aceptación del comando y el cambio de modo de vuelo reflejado en la terminal de ArduPilot.



```
File Edit View Terminal Tabs Help
movax@Aspire-E5:~/pymavlinks$ python3 flightMode.py
Command is valid (is supported and has valid parameters), and was executed.
movax@Aspire-E5:~/pymavlinks$
```

(a) Respuesta en Pymavlink

```
File Edit View Terminal Tabs Help
Init Gyro**
AP: ArduPilot Ready
AP: AHRS: DCM active
Fence present
AP: SaveWaypoint LOW
AP: PrecisionLoiter MIDDLE
AP: EKF3 IMU0 buffs IMU=19 OBS=7 OF=17 EN:17 dt=0.0120
AP: EKF3 IMU1 buffs IMU=19 OBS=7 OF=17 EN:17 dt=0.0120
AP: EKF3 IMU0 initialised
AP: EKF3 IMU1 initialised
AP: EKF3 IMU1 active
AP: AHRS: EKF3 active
AP: EKF3 IMU0 tilt alignment complete
AP: EKF3 IMU0 tilt alignment complete
AP: EKF3 IMU0 MAG0 initial yaw alignment complete
AP: EKF3 IMU1 MAG0 initial yaw alignment complete
AP: GPS 1: detected as u-blox at 230400 baud
AP: EKF3 IMU0 origin set
AP: EKF3 IMU1 origin set
AP: EKF3 IMU1 is using GPS
AP: EKF3 IMU0 is using GPS
Flight battery 100 percent
Got MAVLink msg: COMMAND_ACK {command : 11, result : 0}
GUIDED> Mode GUIDED
```

(b) Respuesta en ArduPilot

Figura 4.26: Ejecución del script para la configuración del modo de vuelo

4.4.3. Secuencia de despegue

Una vez que se logró establecer la comunicación con el piloto automático y se seleccionó en modo de vuelo adecuado, la secuencia de despegue resultó bastante sencilla de implementar, pues solo consta de dos pasos; enviar el comando para el armado de motores y enviar el comando de despegue especificando la altura deseada. Sin embargo, para que lo anterior se logre ejecutar de manera correcta, es necesario esperar a que los sistemas de la computadora de vuelo se inicialicen por completo, en especial aquellos sensores que sirven para la estimación de posición del dron, pues se requiere de estos para que el dron

pueda despegar y elevarse a la altura solicitada por el usuario.

Esto último representó un problema al momento de crear el nodo de la misión de vuelo en ROS, pues si los filtros de estimación no se encuentran listos al momento de ejecutar el comando de despegue, la instrucción no se ejecuta y el dron es incapaz de iniciar su rutina de vuelo. La solución a este problema se aborda en la respectiva sección de ROS.

Dicho lo anterior, el script perteneciente a esta subsección no cuenta con la etapa de la selección del modo de vuelo, pues es necesario ejecutar el programa en el momento en el que los filtros del sistema han sido inicializados, y si se corre el script al mismo tiempo que se inicia el SITL, el comando de despegue es rechazado. Entonces, para ejecutar el programa, el usuario tiene que cambiar el modo de vuelo de forma manual dentro de la terminal de ArduPilot, de esta forma se asegura que el sistema está listo para recibir la instrucción de despegue.

Además, en este caso el script envía los comandos al piloto automático mediante la instrucción `command.long_send`, la cual recibe 11 parámetros; en donde los primeros 2 corresponden al sistema y al componente de MAVLink, los cuales son atributos pertenecientes al objeto creado al momento de establecer la comunicación con el piloto automático. Por otro lado, el siguiente parámetro corresponde al comando de vuelo que se desea enviar y los siguientes 8 sus respectivos atributos. Recordando que un comando de MAVLink puede recibir hasta 7 atributos, existen muchos comando en donde no se ocupan por completo esta cantidad de atributos, en estos casos, es necesario revisar la documentación perteneciente al comando en cuestión y verificar la cantidad específica de atributos requeridos, el resto de parámetros no utilizados se debe de dejar en 0.

Entonces, en el caso de esta etapa, los comandos de vuelo utilizados fueron `MAV_CMD_COMPONENT_ARM_DISARM` para el armado de los motores y `MAV_CMD_NAV_TAKEOFF` para la orden de despegue. La figura 4.27 muestra el comportamiento observado en la simulación tras la ejecución del script con la secuencia de despegue, la figura 4.27a presenta el estado inicial de la simulación, antes de que se enviara el comando de vuelo, por otro lado, la figura 4.27b muestra el dron en vuelo, tras haber alcanzado la altura indicada en el comando de vuelo. Entonces, a simple vista se logra apreciar que el comando se ejecutó de manera correcta y el dron despegó sin ningún tipo de problema.

Si bien la figura 4.27 muestra de manera acertada la correcta ejecución de la secuencia

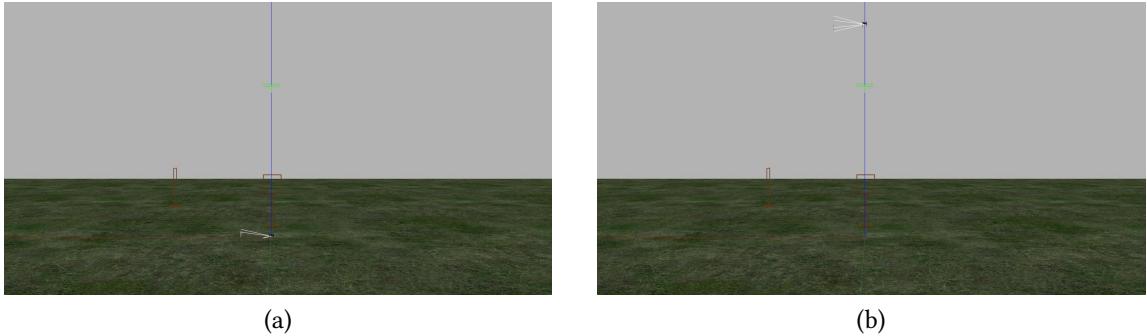


Figura 4.27: Comportamiento de la simulación ante el comando de despegue

de despegue, a simple vista resulta difícil decir con exactitud si el dron alcanzó, o no, la altura deseada. Debido a lo anterior, se recopilaron los datos correspondientes a la posición y velocidad del dron durante su ascenso, a partir de uno de los mensajes de MavLink que permite obtener acceso a esta información manejada por el piloto automático. En consecuencia, la figura 4.28 es una gráfica que muestra la altura del dron durante su despegue, de tal forma que es posible apreciar que, en efecto, el dron logra alcanzar la altitud deseada durante su despegue con un ligero sobre paso, aproximadamente a los 13 segundos después de comenzar a ejecutar el comando de vuelo. Por otro lado, cabe mencionar que, debido al marco de referencia con el que se está trabajando, un desplazamiento hacia arriba en el eje z es considerado negativo, mientras que un desplazamiento hacia abajo en ese eje se considera positivo; sin embargo, para fines de una interpretación de datos más intuitiva, se invirtió el signo de la lectura de datos en el eje z .

Adicionalmente, la figura 4.29 corresponde a una gráfica que describe el comportamiento de la velocidad de ascenso del dron, y de manera congruente con la gráfica mencionada anteriormente, se puede apreciar un descenso en la velocidad de ascenso en el intervalo de tiempo de 10 s y 15 s, lapso en el que el dron logra llegar a la referencia y por lo tanto ya no es necesario ascender más; además, se observa una velocidad máxima de ascenso de $2.21 \frac{m}{s}$.

4.4.4. Seguimiento de trayectoria

Hecho todo lo anterior, la última prueba que se tuvo que realizar, en cuanto a una funcionalidad en específico de Pymavlink, fue la del seguimiento de trayectoria por medio



Figura 4.28: Respuesta en la altura para el comando de despegue.

de comandos de vuelo directos, en forma de waypoints. Sin embargo, debido al problema mencionado en la etapa anterior, el script perteneciente a esta subsección no ejecuta las etapas anteriores, sino que para poder ejecutar el programa, el usuario tiene que esperar a que la terminal de ArduPilot permita el cambio de modo de vuelo de forma manual, y después ejecutar los respectivos comandos de vuelo para realizar la secuencia de despegue del dron.

Ahora bien, la prueba realizada para esta etapa fue sencilla, se enviaron dos waypoint utilizando el marco de referencia local, en donde el origen, o la coordenada (0,0,0), corresponde al punto inicial del mundo creado, denotado de forma visual por 3 barras ortogonales de color azul para el eje z, azul para el eje x y verde para el eje y. Además, como se mencionó, la trayectoria de vuelo está compuesta por dos waypoints, los cuales fueron definidos de tal manera que se buscó que el dron cruzara por las dos primeras compuertas del circuito de vuelo, dichos waypoint son [35,0,3] y [35, 17, 2]; Dicho esto, la secuencia pensada para la ejecución de esta prueba fue, primero hacer ascender el dron a una altura de 10 m y después indicarle su trayectoria de vuelo a partir de los waypoints especificados.

Asimismo, se buscó comprobar si el piloto automático espera a que el dron llegue al primer waypoint para dirigirse hacia el segundo, sin embargo, esto no sucede y el piloto automático ejecuta el segundo waypoint definido, pues no hay ningún tiempo de espera o bandera que le indique al piloto que debe de esperar a que se termine la ejecución del

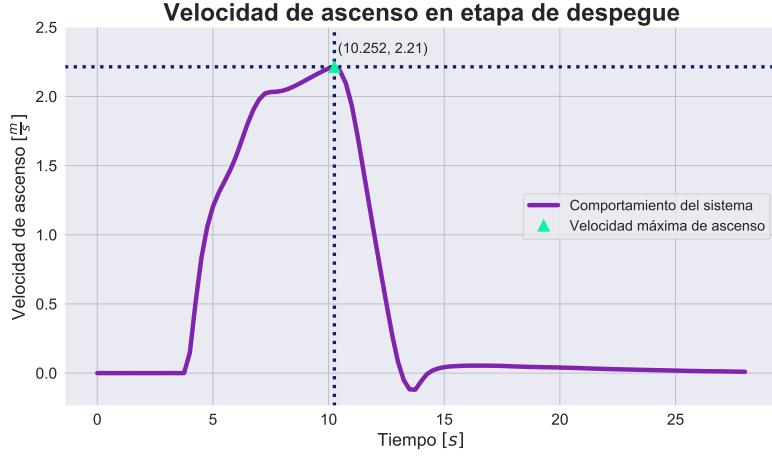


Figura 4.29: Gráfica de velocidad de despegue.

primer waypoint. Entonces, para evitar que el piloto automático ignorara el primer waypoint, se utilizó un mensaje de MAVLink definido como *NAV_CONTROLLER_OUTPUT* el cual publica información relacionada sobre el comando de vuelo enviado al piloto automático, entre la cual se encuentra la distancia faltante para que el dron llegue al waypoint especificado al momento de lectura del mensaje. Entonces, la instrucción para el seguimiento del segundo waypoint es enviada al piloto automático hasta que el mensaje especifica que la distancia restante para llegar al primer waypoint es igual a cero.

Continuando con los resultados de la prueba, en este caso el script solo ejecuta las instrucciones para guiar el vuelo del dron, por lo que es necesario que el usuario realice todo el pre-proceso necesario para el despegue del dron. En el caso de la prueba realizada la altitud de despegue indicada fue de 10 m, por lo que para el seguimiento del waypoint, el dron tuvo que descender a 3 m de altura al mismo tiempo que avanzó 10 m en el eje x, y después otros 20 m. Cabe mencionar que para el piloto automático, una altura negativa indica un punto sobre la referencia inicial, y análogamente, una altura positiva corresponde a un punto por debajo de la referencia inicial.

Además, los waypoints son enviados utilizando el comando de vuelo *MAVLink_set_position_target_local_ned_message*, el cual requiere 16 parámetros para realizar el seguimiento de un waypoint. La tabla 4.3 enumera los parámetros y muestra una breve descripción de los mismos; Por otro lado, el parámetro *type_mask* es algo complejo de explicar, por lo que, por motivos de practicidad y legibilidad, se describe a más detalle fuera de la

tabla.

Retomando lo anterior, el parámetro `type_mask` indica la cantidad de parámetros que el comando de vuelo tiene que tomar en cuenta de acuerdo a lo que el usuario deseé indicar, dígase posición, velocidad, aceleración o las tres magnitudes anteriores. Dicho esto, mencionando un ejemplo, si se desea enviar un comando de velocidad, es necesario especificar la velocidad de referencia para los 3 ejes cartesianos y el resto de parámetros serán ignorados independiente del valor indicado en el comando.

Adicionalmente, la tabla 4.4 muestra los distintos valores que se le pueden asignar al parámetro de máscara. En el caso de este trabajo se utilizó el primer tipo de máscara, debido a que solo se buscó que el dron siguiera una trayectoria, con la máxima velocidad que el vehículo pudiera proporcionar.

Parámetro	Descripción
<code>time_boot_ms</code>	Tiempo de arranque del sistema en ms
<code>target_system</code>	Identificador del sistema
<code>target_component</code>	Identificador del componente o controlador de vuelo
<code>coordinate_frame</code>	Marco de referencia
<code>type_mask</code>	Indica los campos que se deben de ignorar
<code>x</code>	Posición en el eje x en m
<code>y</code>	Posición en el eje y en m
<code>z</code>	Posición en el eje z en m
<code>vx</code>	Velocidad en el eje x en m/s
<code>vy</code>	Velocidad en el eje y en m/s
<code>vz</code>	Velocidad en el eje z en m/s
<code>afx</code>	Aceleración en el eje x en m/s^2
<code>afy</code>	Aceleración en el eje y en m/s^2
<code>afz</code>	Aceleración en el eje z en m/s^2
<code>yaw</code>	Ángulo de guiñada en radianes
<code>yaw rate</code>	Aceleración de la guiñada en rad/s

Tabla 4.3: Parámetros de la instrucción para seguimiento de waypoints

Ahora bien, retomando el contexto de los resultados obtenidos al momento de ejecutar el script desarrollado para esta subsección, la figura 4.30 presenta el comportamiento observado en la simulación tras la ejecución de la secuencia de comando especificada para esta subsección. Entonces, en la figura 4.30a se observa el estado inicial del dron, previo a la ejecución del protocolo; la figura 4.30b muestra el dron después de haber ascendido

Tipo	Binario	Hexadecimal	Decimal
Posición	0b110111111000	0x0DF8	3576
Velocidad	0b110111000111	0x0DC7	3527
Aceleración	0b110000111111	0x0C3F	3135
Pos + Vel	0b110111000000	0x0DC0	3520
Pos + Vel + Acel	0b110000000000	0x0C00	3072
Guiñada	0b100111111111	0x09FF	2559
Velocidad de guiñada	0b010111111111	0x05FF	1535

Tabla 4.4: Tipos de máscara para el comando de seguimiento de trayectoria

a la altura de 10 m (la cual fue comandada de forma manual a través de la terminal de ArduPilot); Por otro lado, pese a que se mencionó que los waypoints se definieron con el fin de que el dron fuera capaz de cruzar las primeras dos compuertas del circuito de vuelo, en la figura 4.30c se aprecia que el dron sobrevuela la primera compuerta del circuito y no para a través de ella, esto se debe a la altura de ascenso que se definió durante la prueba, sin embargo, cabe mencionar que si al dron se le comanda un despegue de 3 m (altura a la que se encuentra el centro de las compuertas grandes), este es capaz de cruzar por el centro de las dos primeras compuertas utilizando los dos waypoints definidos; asimismo, la figura 4.30d muestra una captura del dron después de que este llegó a su punto final tras seguir el segundo waypoint, se aprecia que en este caso el dron si fue capaz de cruzar a través de la segunda compuerta.

De manera similar a la subsección anterior, la figura 4.30 permite observar de manera rápida y práctica si el comportamiento del dron fue el adecuado; sin embargo, el análisis del desempeño del dron se logra a partir de los datos de vuelo recopilados. La figura 4.31 muestra una gráfica en donde se describe la trayectoria de vuelo seguida por el dron, en ella se especifica la ubicación de las compuertas con base en el recorrido descrito. Cabe mencionar que, contrario a lo que se considera intuitivo, el desplazamiento a lo largo del eje *x* se observa en el eje vertical, mientras que el desplazamiento en el eje *y* corresponde al eje horizontal, esto es debido a que de esta forma la gráfica es congruente con la perspectiva de la cámara dentro de la simulación. En consecuencia, se puede decir que, a partir de la figura 4.31, el dron siguió los waypoints de manera satisfactoria, logrando completar una ruta de vuelo simple.

De forma subsecuente, la figura 4.32 presenta el comportamiento del desplazamiento

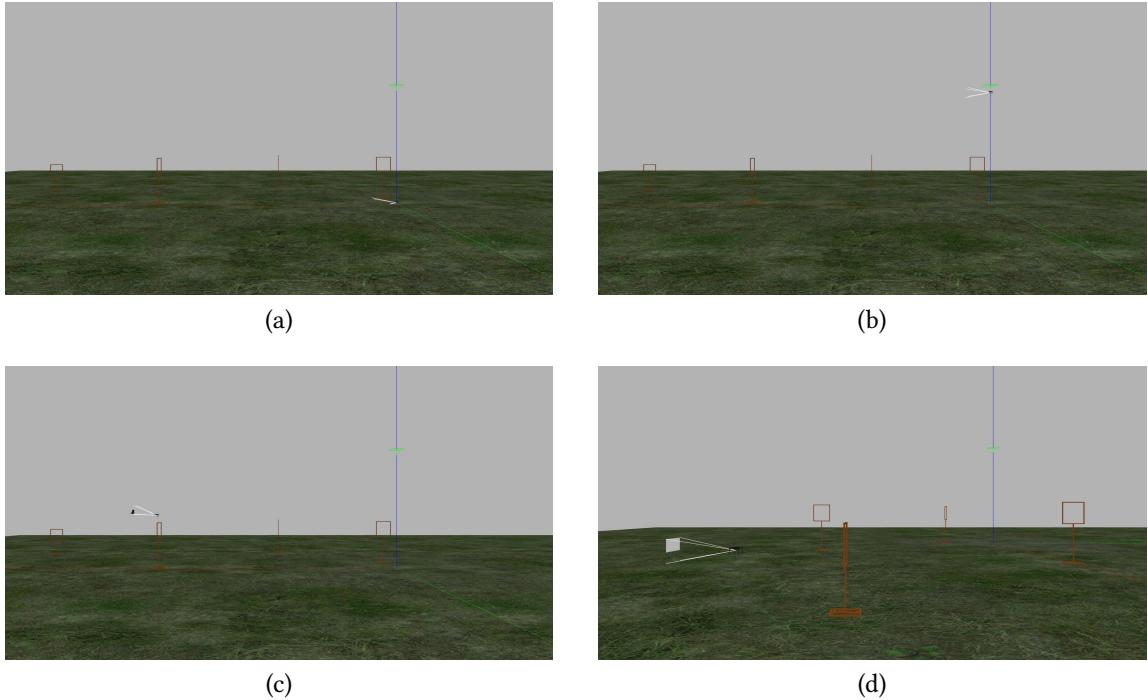


Figura 4.30: Etapas de la prueba de seguimiento de trayectoria

lineal del dron a lo largo de los tres ejes espaciales. La figura 4.32a corresponde al desplazamiento en el eje x , en ella se observa como es que el dron llega a la referencia de 35 m sin ningún sobre paso, en un tiempo de aproximadamente 10 s; una vez que el dron completa la trayectoria definida por el primer waypoint, este gira a la izquierda para dirigirse al segundo waypoint, de tal forma que ya no se observa un desplazamiento en el eje x , sino, en el eje y , en consecuencia, el desplazamiento observado en la gráfica de la figura 4.32b ocurre hasta el segundo 10, que es cuando se da la transición entre waypoints. Por último, en la gráfica de la figura 4.32c se logra apreciar el cambio de altura durante el vuelo del dron, resulta evidente la forma en la que el dron desciende de los 10 m indicados en su despegue a una primera referencia de 3 m, y posteriormente el lapso de transición entre waypoints desciende 1 m más, pues la segunda compuerta es más pequeña y está más cerca del suelo; en ambos casos logra el dron lograr llegar a la altura indicada.

Asimismo, el conjunto de gráficas en la figura 4.33, esta esta relacionada con las velocidades lineales asociadas a cada uno de los 3 ejes de desplazamiento del dron. Dicho lo anterior, en la figura 4.33a, se observa como en el eje x se alcanza la velocidad máxima



Figura 4.31: Recorrido realizado por el dron

de $8.34 \frac{m}{s}$, en el segundo 3.75, antes del tiempo de transición de waypoints, de tal manera que para el segundo 10, la velocidad a lo largo de este eje ha decrecido de forma significativa y de forma intuitiva, se aprecia en la figura 4.33b que como es que la velocidad a lo largo del eje y comienza su ascenso a partir del tiempo de transición de waypoints, llegando un máximo de $5.59 \frac{m}{s}$ en el segundo 11.25 del vuelo del dron.

Por otra parte, la figura 4.33c muestra el comportamiento de la velocidad a lo largo del eje z ; es importante no perder de vista que, como se mencionó en la subsección de la etapa de despegue, los signos de las velocidades fueron invertidos para que el análisis de resultados correspondiera de forma visual con el comportamiento observado durante la ejecución de la simulación. Dicho esto, en la figura 4.33c se destaca una pendiente negativa en los primeros 2 segundos de la simulación, esto se debe a que el dron inició a una altura de 10 m y tuvo que descender a 3 m, por lo que el sentido del vector de velocidad fue hacia abajo, llegando a una velocidad de $-1.56 \frac{m}{s}$ en el segundo 2, después de este lapso, el dron tuvo que disminuir su velocidad pues se fue aproximando cada vez más a la altura deseada de 3 m, lo cual ocurrió poco después del segundo 5, con base en la figura 4.32c; en la gráfica se puede observar un pequeño sobrepaso con una velocidad positiva de $0.0634 \frac{m}{s}$. Luego, un poco antes del segundo 10, ocurre la transición al siguiente waypoint, por lo que el dron tuvo que descender a los 2 m de altura, una vez más, presentando una velocidad negativa para después detenerse de forma gradual hasta

Capítulo 4. Resultados

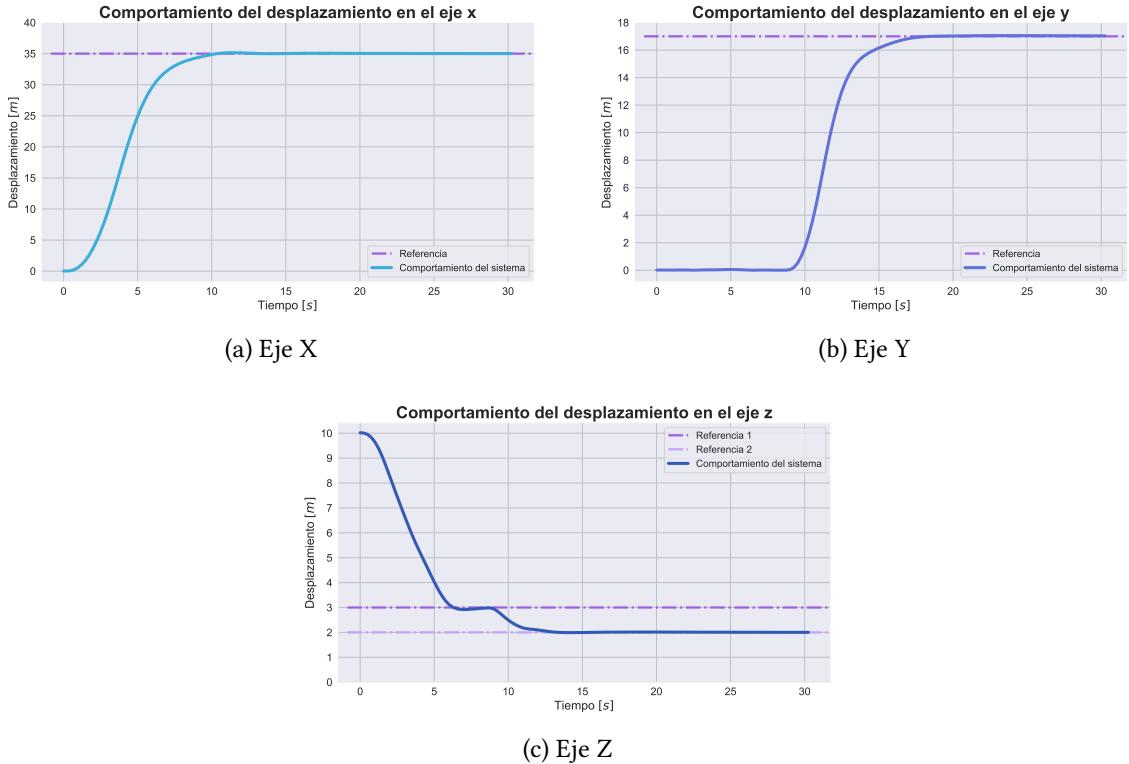


Figura 4.32: Gráficas de desplazamiento para los tres ejes

llegar a la altura de referencia. Por último, se observa que el dron finaliza con su recorrido aproximadamente en el segundo 15, pues la velocidad se mantiene en cero después de ese punto.

Por último con respecto a esta prueba, la figura 4.34 presenta una proyección en 3D de la trayectoria de vuelo ejecutada por el dron. Esta gráfica presenta de manera más visual la ruta observada durante la ejecución de la simulación y es posible observar los 3 instantes más significativos de esta prueba. Primero, el dron inicia con una altura de 10 m; después se le comanda el primer waypoint [35,0,3], lo que corresponde al descenso pronunciado observado en la gráfica; cuando el dron llega a su primera referencia, se le comanda el segundo waypoint [35,17,2], donde se observa otro descenso, menos pronunciado y un movimiento a lo largo del eje y.

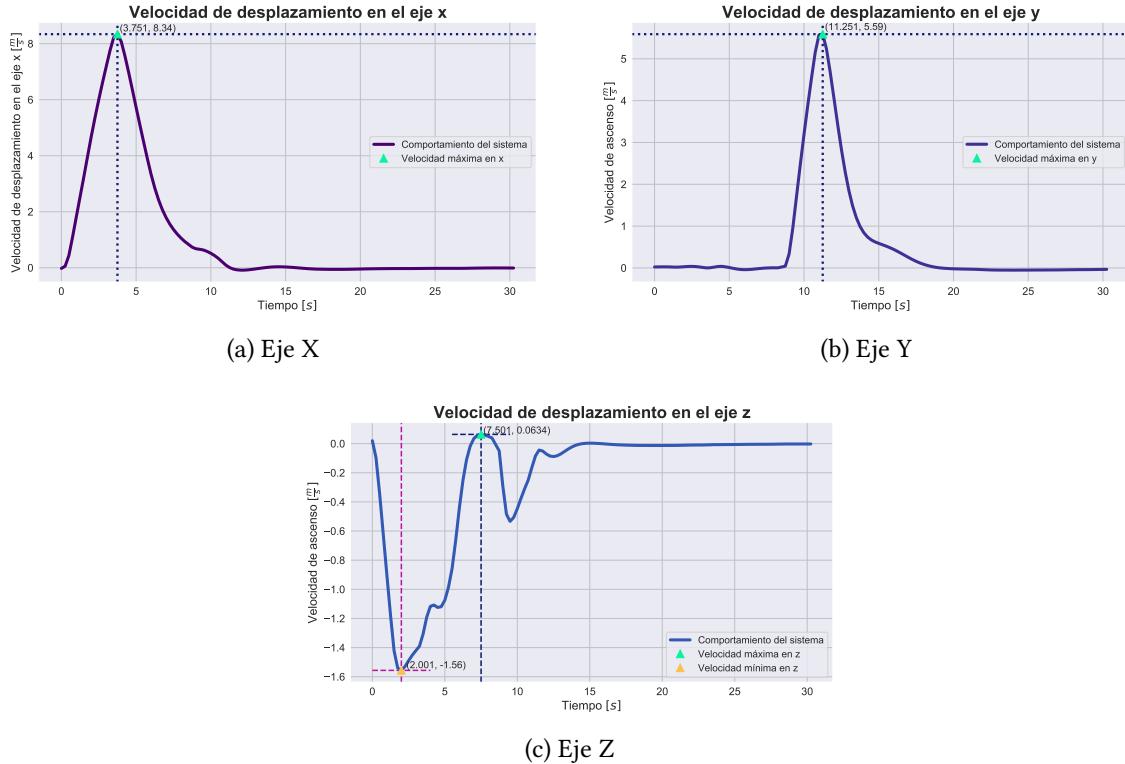


Figura 4.33: Comportamiento de la velocidad lineal de vuelo en los tres ejes

4.4.5. Misión de vuelo

Con la etapa anterior se culminaron las pruebas por módulos; en esta subsección se describe el proceso realizado para integrar todas las etapas descritas hasta el momento.

Debido a que se trabajó con un paradigma de módulos, en donde cada etapa descrita corresponde a un módulo y/o funcionalidad específica, la unificación de todo lo anteriormente descrito resultó ser un proceso bastante lineal y práctico, en la mayor parte de las etapas se utilizó el mismo código desarrollado para su respectivo script; Por otro lado, en esta etapa se buscó implementar de forma completa la misión de vuelo para el dron, de tal manera que fuera capaz de cruzar a través de las 4 compuertas que componen el circuito de vuelo, por lo tanto, la misión de vuelo cuenta con 4 waypoints, los cuales se muestran en la tabla 4.5.

Dicho lo anterior, a manera de recapitulación, se describen los módulos en el orden en el que fueron implementados en el script de esta subsección:

Trayectoria de vuelo

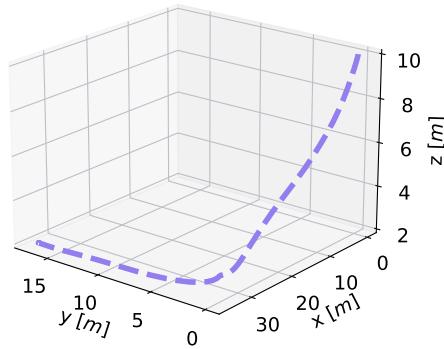


Figura 4.34: Gráfica en 3D de la trayectoria seguida por el dron.

x	y	z
35	0	3
35	17	2
0	17	3
0	0	3

Tabla 4.5: Waypoints utilizados en la misión de vuelo

1. Establecimiento de comunicación entre Pymavlink y ArduPilot
2. Cambio de modo de vuelo de stabilize a guided
3. Armado de motores
4. Envío de orden de despegue
5. Envío de comandos de vuelo para seguimiento de waypoints

Adicionalmente, el script de misión de vuelo busca eliminar la necesidad de que el usuario tenga que introducir comando de vuelo de forma manual en la terminal de ArduPilot; es decir, solo es necesario ejecutar el script y el dron será capaz de ejecutar todas las etapas descritas.

Ahora bien, la figura 4.35 muestra el comportamiento observado en simulación tras haber ejecutado el script de la misión de vuelo, de tal manera que en la figura 4.35a se

observa como el dron es capaz de cruzar por el centro de la primera compuerta; la figura 4.35b presenta el cruce por la segunda compuerta; la figura 4.35c corresponde al waypoint para la tercera compuerta y por último en la figura 4.35d se observa el estado final del dron tras haber atravesado la cuarta compuerta. Entonces, a partir de la figura 4.35 es posible apreciar, a simple vista, como el dron logró completar el circuito de vuelo de forma exitosa.

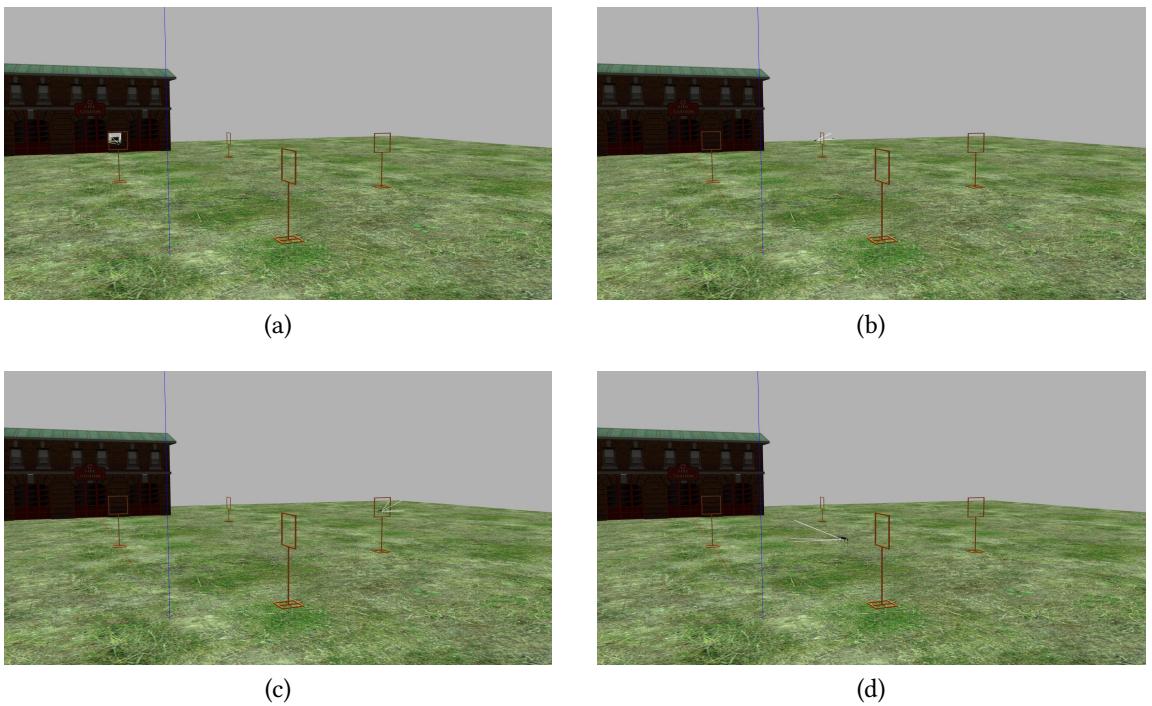


Figura 4.35: Seguimiento de misión de vuelo completa

De forma similar a la subsección correspondiente al seguimiento de trayectoria, la figura 4.36 presenta la ruta de vuelo realizada por el dron. Se vuelve a destacar que, contraintuitivamente, el desplazamiento entre los ejes se encuentra acomodado de tal manera que correspondan a la perspectiva con la que se observó el vuelo del dron; es decir, como si se estuviera viendo el desplazamiento del dron como se observa en la figura 4.35. Además, la figura 4.36 cuenta con la ubicación de las cuatro compuertas del circuito de vuelo, así como el sentido de vuelo que siguió el dron durante su recorrido. A partir de esto, se logra apreciar que los waypoints fueron definidos de tal manera que el dron describiera una trayectoria rectangular uniforme, por lo tanto, la trayectoria recorrida por

cada waypoint corresponde a cada una de las artistas del rectángulo y la transición entre cada uno está representado por los vértices de este.

Por otro lado, cabe mencionar que MAVLink acepta misiones de vuelo de forma nativa, en donde solo es necesario cargar un archivo de texto en donde se defina la serie de waypoints que se desean seguir, por medio de coordenadas según el marco de referencia; sin embargo, al tiempo de desarrollo de este trabajo, no se encontró ningún comando de vuelo aceptado por PymavLink que permitiera cargar de forma sencilla una misión de vuelo, por lo que, la trayectoria de vuelo implementada se realizó a partir de una función desarrollada desde cero, en donde los waypoints son enviados de forma explícita y de uno en uno, debido a esto fue necesario implementar la solución mencionada en la subsección anterior, en donde por medio de un mensaje es posible conocer la distancia restante para que el dron complete la trayectoria de cada waypoint.



Figura 4.36: Recorrido realizado por el dron

Siguiendo con los resultados obtenidos en esta subsección, la figura 4.37 muestra el desplazamiento lineal capturado durante el vuelo del dron. Es posible observar un comportamiento bastante peculiar en todos los ejes, por un lado en la figura 4.37a se puede apreciar como para el primer waypoint el desplazamiento en el eje x incrementa de forma gradual hasta llegar a 35 m, en aproximadamente 10 segundos, después, ocurre la transición para el segundo waypoint, sin embargo la posición en el eje x se mantiene intacta, por eso el dron permanece en 35, hasta el segundo 16, en donde ocurre la transición

al tercer waypoint y es necesario disminuir el desplazamiento hasta llegar a 0 m en el eje x , en donde el dron se mantiene hasta finalizar su recorrido. Por otro lado, en la figura 4.37b se aprecia un comportamiento en el eje y de cierta forma complementario al del eje x , en los primeros 10 segundos del recorrido, el dron no presenta ningún desplazamiento en y por lo que su posición con respecto a este eje se mantiene en 0, posteriormente, ocurre la transición al segundo waypoint y es cuando ocurre el primer desplazamiento en y , de tal forma que el dron avanza 17 m, después, en el segundo 25, se da la transición al tercer waypoint y la posición en y comienza a decrementar, de tal forma que para el segundo 30 el dron ya se encuentra en el origen del circuito de vuelo. De manera similar, a partir de la figura 4.37c es posible intuir el comportamiento de la altura del dron durante su vuelo, el cual resulta ser más sencillo que los dos anteriores pues en este caso, la altitud solamente osciló entre dos valores 3 y 2 m, donde a los 10 segundos de vuelo ocurre un descenso a los 2 m de altura, pues ocurre la transición al segundo waypoint y la compuerta correspondiente a este waypoint es de menor tamaño, por lo que es necesario que el dron descienda y luego vuelva a ascender a los 3 m cuando ocurre la siguiente transición de waypoints.

Así mismo, la figura 4.38 presenta el comportamiento de la velocidad lineal en los tres ejes de desplazamiento. La gráfica de la figura 4.38a presenta el comportamiento en el eje x , mientras que la figura 4.38b hace lo respectivo con el eje y ; como se observa, ambas gráficas presentan comportamientos bastante similares. El primer waypoint [35,0,3] involucra un desplazamiento positivo en el eje x , sin desplazarse en el eje y , por lo que en los primeros 10 segundos de la simulación se observa el máximo en v_x de $8.42 \frac{m}{s}$, mientras que v_y permanece en 0; el decrecimiento de v_x se debe a que el dron comienza a detenerse conforme se acerca a la referencia de posición, y poco antes del segundo 10 se observa un incremento en v_y , debido a la transición al segundo waypoint [35,17,2], donde v_y alcanza su máximo de $5.53 \frac{m}{s}$ en el segundo 11.25 para después disminuir su velocidad.

Luego, se observa una pendiente negativa en v_x pues el tercer waypoint [0,17,3] implica un retroceso en el desplazamiento en x , de tal forma que después del segundo 18.5 también se observa una reducción en la magnitud de v_x hasta llegar a cero. Este comportamiento también se observa para el cuarto waypoint [0,0,0] en donde v_y presenta la pendiente negativa y posteriormente un detenimiento completo.

Por último, en la figura 4.38c es posible observar el comportamiento de v_z . Se aprecia

Capítulo 4. Resultados

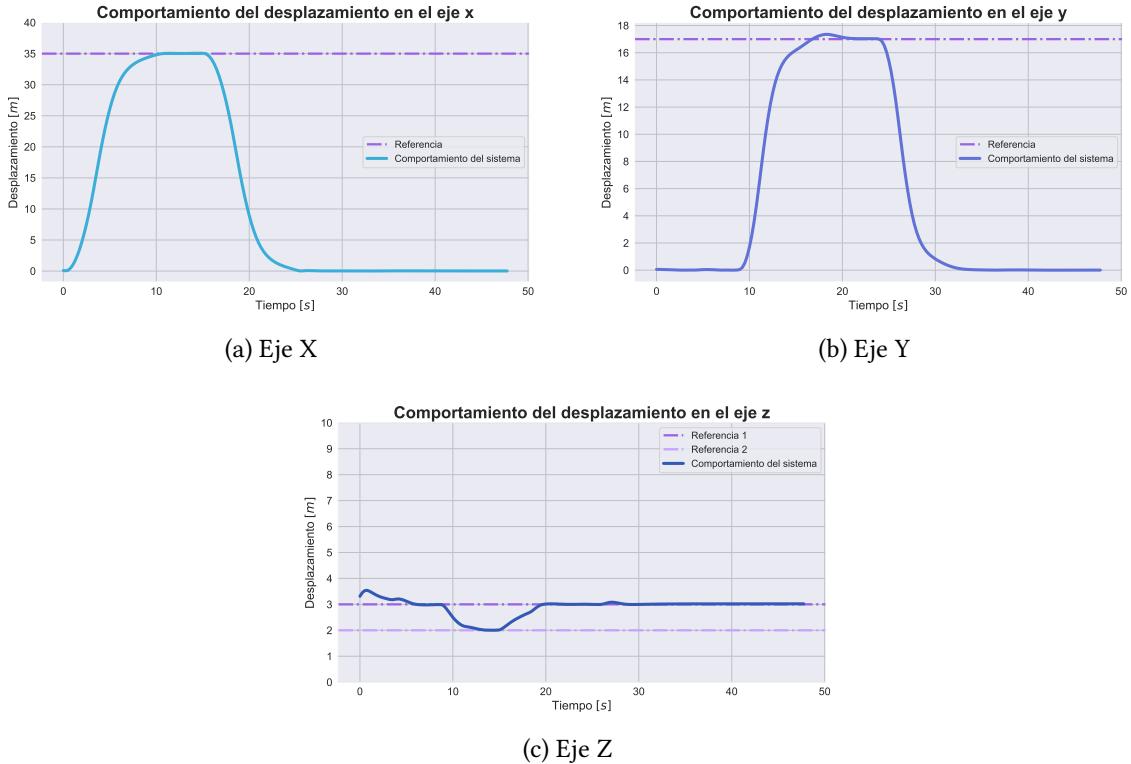


Figura 4.37: Gráficas de desplazamiento para los tres ejes

que al inicio de la gráfica se presenta el máximo de velocidad ($0.67 \frac{m}{s}$), esto se debe a que la adquisición de datos comenzó justo después de que la etapa de despegue hubiera concluido, por lo que el dron ya contaba con cierta velocidad, después de eso, se observan decrementos graduales en v_z con el fin de detener su movimiento, por la llegada a la altura de referencia; sin embargo, en el segundo 10 ocurre el comando del segundo waypoint, el cual cambia la referencia a 2 m, por lo que el dron tuvo que descender para alcanzar esa altura, y posteriormente, el tercer waypoint provocó que el dron volviera a ascender, debido a esto se observa picos positivos en el segundo 15 aproximadamente.

4.4.6. Aterrizaje (extra)

Por último, con respecto al algoritmo de seguimiento de trayectoria, esta subsección se maneja como extra debido a que en la implementación final, el dron no realiza la secuencia de aterrizaje, pues se decidió que completara el circuito de forma indefinida. Sin

Capítulo 4. Resultados

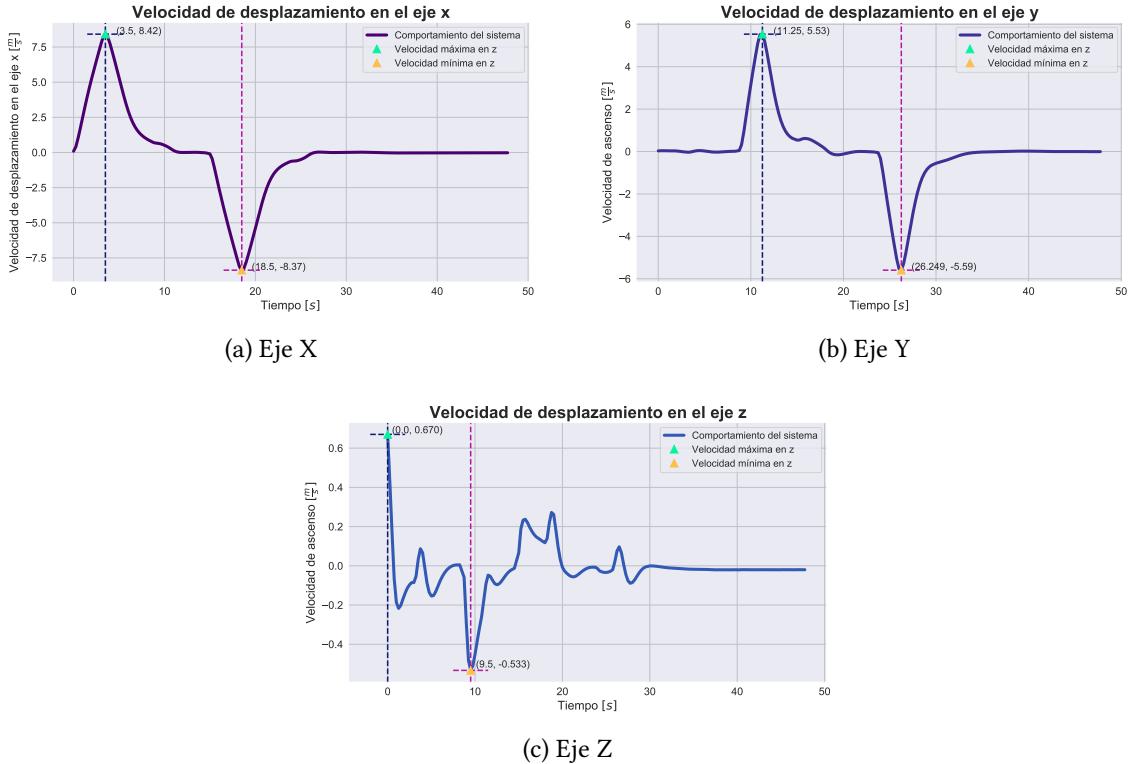


Figura 4.38: Comportamiento de la velocidad lineal de vuelo en los tres ejes

embargo, se consideró que es importante mencionar que durante las pruebas realizadas sí se implementó una lógica que permitiera aterrizar el dron una vez que completo el circuito de vuelo.

Se consideraron dos posibles implementaciones para el aterrizaje, la primera, utilizar 2 waypoints más para guiar al dron hacia el origen y, por otro lado, el uso del modo de vuelo integrado en el piloto automático. La segunda opción resultó ser más factible, pues al utilizar waypoints para el descenso del dron, este lo hace de forma brusca y al entrar en contacto con el suelo se observa un rebote derivado de la velocidad de descenso.

Entonces, para implementar el modo de vuelo de aterrizaje, se realizó el mismo proceso implementado en la subsección de la secuencia de despegue, la diferencia se encuentra en que ahora el modo de vuelo especificado es *LAND*, y por supuesto, en este caso no se requiere enviar el comando de vuelo para el despegue.

4.5. ROS

En esta última sección de resultados se desglosa el proceso llevado a cabo para la implementación de los nodos del algoritmo de visión artificial y el seguimiento de trayectoria. Cabe mencionar que en esta última parte no es tan extensa como las anteriores, pues básicamente la creación de los nodos puede ser vista como un proceso de integración. De hecho, se parte de lo últimos códigos desarrollados en cada sección, de tal forma que el contenido de cada nodo es en esencia lo desarrollado en su respectiva subsección.

4.5.1. Visión Artificial

La implementación del algoritmo de detección de compuertas fue quizás la más sencilla, pues al utilizar funciones básicas de OpenCV, no fue necesario incluir librerías adicionales, quizás el cambio más representativo con respecto al programa mencionado en la sección del sistema de visión artificial es la inclusión de la librería *cv_bridge.h*, la cual permite realizar la conversión entre una imagen recibida por un topic de ROS a un formato que OpenCV sea capaz de manejar. Entonces, la implementación de la aplicación resultó ser bastante intuitiva y práctica.

A partir de lo anterior, el paradigma para la conversión entre un programa de programación estructura a un nodo de ROS es sencillo en mucho de los casos, pues la consideración más importante que se tiene que realizar es que, al ser un nodo de ROS, el programa se estará ejecutando como un proceso dentro del sistema operativo, de tal manera que su ejecución se hará de forma indefinida a manera de ciclo. Entonces, el algoritmo 2 ejemplifica a grandes rasgos la lógica implementada. Se puede observar que la secuencia a seguir es sencilla; sin embargo, quizás lo que puede llegar a consumir más tiempo al realizar este tipo de porteo son las consideraciones que se tiene que realizar dentro de la programación orientada a objetos (POO), que si bien un nodo de ROS puede ser implementado con lógica secuencial, el uso de POO simplifica bastante la cantidad de código.

Ahora, las figuras 4.39, 4.40, 4.41, 4.42 presentan el desempeño del algoritmo dentro de la simulación, en donde cada conjunto de imágenes corresponde al momento en que la cámara del dron tiene en su campo de visión a cada una de las compuertas que componen el circuito de vuelo. Hablando de manera general, cada grupo de imágenes contiene un total de 8 sub-imágenes, en donde la primera fila de imágenes muestra el fotograma

Algorithm 2 Nodo de visión artificial para la detección de compuertas

```

while True do
    Leer topic de transmisión de imagen de cámara
    Convertir tipo de imagen de ROS a OpenCV
    Ejecutar algoritmo de visión artificial
end while

```

original captado por la cámara del dron, y la segunda fila contiene el respectivo resultado asociado al procesamiento de cada uno de estos fotogramas.

Hablando de forma más detallada sobre cada conjunto de imágenes, la figura 4.39 presenta la secuencia de acercamiento que el dron realizó con respecto a la primera compuerta. Es posible resaltar el hecho de que, a pesar de tener un edificio con una fachada roja, el algoritmo fue capaz de aislar la compuerta con éxito; la detección se hace de forma gradual y mejora conforme el dron acorta la distancia con la compuerta.

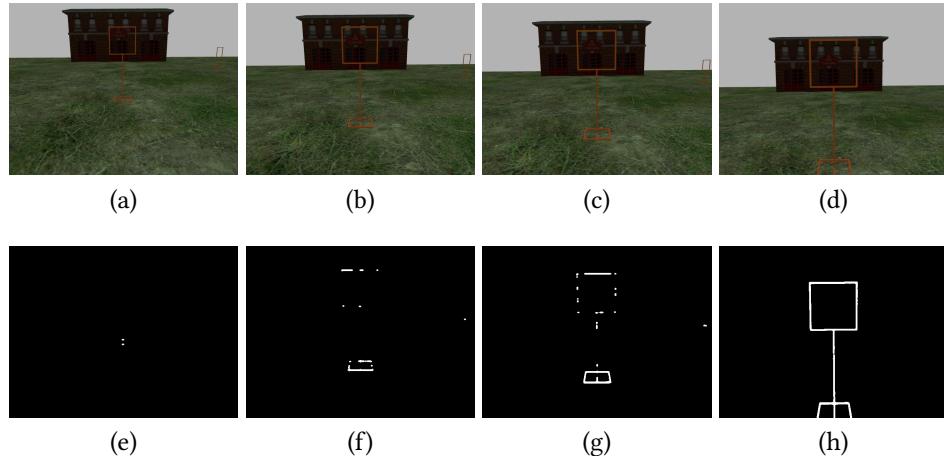


Figura 4.39: Comportamiento de la velocidad lineal de vuelo en los tres ejes

Después, la figura 4.40 contiene la secuencia de fotogramas asociada a la detección de la segunda compuerta, en este caso en particular no existe otro objeto de gran tamaño que puede afectar la detección de forma significativa; sin embargo, es importante mencionar que esta compuerta en específico es de una altura menor en comparación con el resto de compuertas, además, también es posible hacer hincapié en que en algunos de los fotogramas se observa las palas de algunos de los motores del dron. En este caso la detección fue más limpia en comparación al anterior.

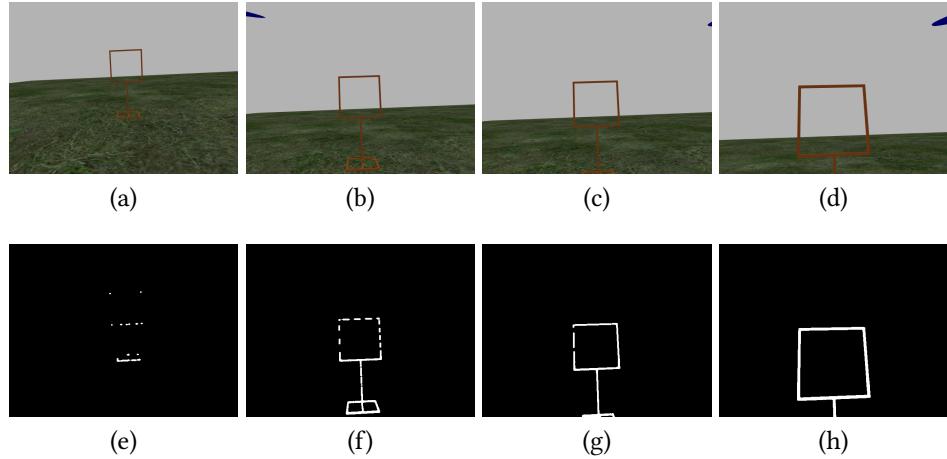


Figura 4.40: Comportamiento de la velocidad lineal de vuelo en los tres ejes

Continuando con el análisis, la figura 4.41 presenta el desempeño obtenido para la detección de la tercera compuerta. En este caso es posible apreciar de mejor manera el efecto que tiene la distancia con respecto al objeto de interés al momento de realizar la detección. Se observa que, a pesar de que no existe una gran cantidad de objetos que puedan afectar el funcionamiento del algoritmo, este no es capaz de detectar las compuertas a grandes distancias, a pesar de que se encuentren dentro de su campo de visión, es más, en este caso se aprecia que no se detectó la compuerta completa, pues cuando el algoritmo fue capaz aislar por completo el objeto, el dron ya se encontraba a una distancia en donde el campo de visión de la cámara no enfoca toda la compuerta.

Por último, la figura 4.42 presenta la respectiva detección para la compuerta 4. En este caso se incluyen fotogramas en donde el dron se encuentra realizando un viraje durante la transición del waypoint. Al igual que en los otros casos, se aprecia que el dron requiere acortar la distancia para realizar el aislamiento por completo; sin embargo, a diferencia del caso anterior, en esta secuencia de fotogramas el dron sí fue capaz de detectar toda la estructura de la compuerta.

A partir de los resultados observados, y a manera de síntesis, se puede decir que el algoritmo propuesto tiene un funcionamiento de cierta forma adecuado, pues logra detectar el objeto para el que fue sintonizado; sin embargo, el desempeño obtenido no fue el mejor, pues el dron necesita estar lo suficientemente cerca de la compuerta para realizar la detección, esto representa un comportamiento poco deseable, pues si en algún

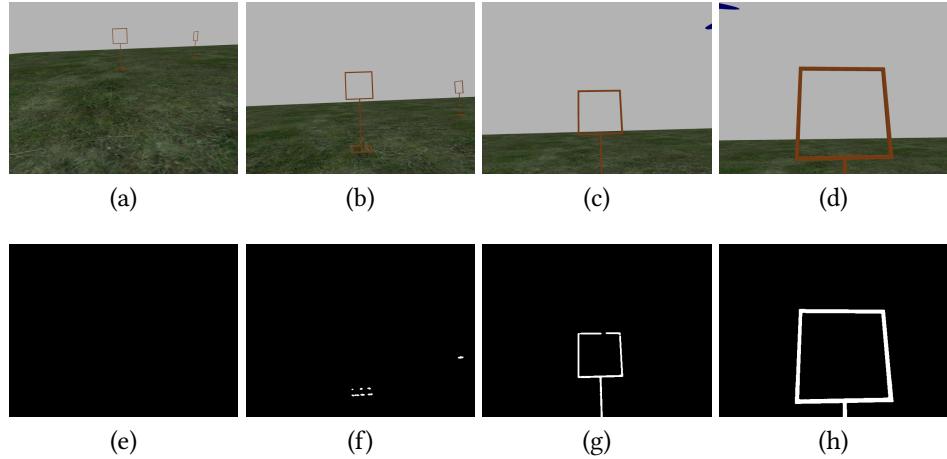


Figura 4.41: Comportamiento de la velocidad lineal de vuelo en los tres ejes

futuro se desea activar la transición de waypoints a partir de la detección de compuerta, no habría mucho tiempo de margen para maniobrar y dirigirse al siguiente waypoint. Además, con base en lo observado, el algoritmo no parece presentar la robustez necesaria para implementarlo en ambientes reales; sin embargo, esto último se encuentra fuera del alcance del presente trabajo y queda como propuesta para futuras implementaciones y análisis.

4.5.2. Seguimiento de trayectoria

A diferencia del algoritmo de visión artificial, la implementación del nodo para el seguimiento de trayectoria si representó un reto, principalmente por la problemática mencionada sobre la asincronía del firmware del piloto automático y la aplicación de seguimiento de trayectoria. Y es que la lógica seguida sigue siendo la misma, sin embargo, fue difícil encontrar una forma de crear un tiempo muerto o de espera, que permitiera que el piloto automático inicializara por completo, de tal manera que estuviera listo para recibir comando de vuelo y no anulara la ejecución de la secuencia de vuelo.

Lo anterior representó un problema de gran relevancia pues el fin máximo de hacer el de los programas a nodos de ROS, fue el aprovechar la posibilidad de crear un lanzador que permitiera ejecutar todo el sistema utilizando un único comando y no 4 por separado. De tal forma que si la secuencia de vuelo es cancelada durante la ejecución del lanzador

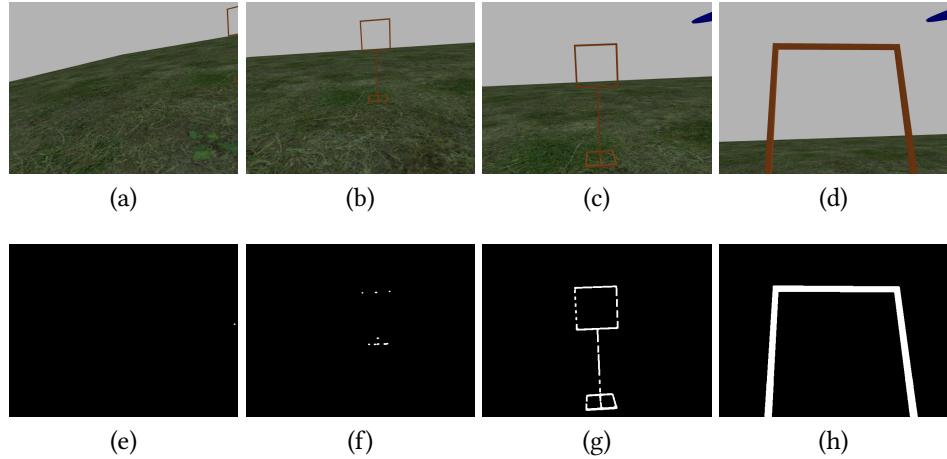


Figura 4.42: Comportamiento de la velocidad lineal de vuelo en los tres ejes

de procesos de ROS, se tendría que volver a ejecutar el programa de seguimiento de trayectoria de forma manual.

Lo anterior se pudo solventar implementando dos acciones en específico:

1. Leer el estado del sistema y espera a que todos los sensores hayan sido inicializados
2. Esperar el tiempo necesario para que el sistema arme los motores y se pueda proceder a la fase del despegue

El primer punto se logró leyendo uno de los mensajes publicados por el piloto automático; *SYS_STATUS* entrega una serie de datos acerca del estado del sistema, entre ellos los estados de los sensores dentro de la simulación, de tal forma que la lectura de este parámetro devuelve un entero. A partir de las pruebas realizadas, se observó que cuando todos los sensores del sistema han sido inicializados el parámetro devuelve un valor de 1382128815, por lo que se implementó un ciclo que realizará la lectura del parámetro hasta que este tuviera el valor deseado. Una vez que los sensores han sido inicializados, se puede proceder con el resto de la secuencia de vuelo.

Ahora, previo al despegue es necesario indicarle al piloto automático que准备 los motores del dron para la secuencia del dron; sin embargo, esta acción conlleva un poco de tiempo, y si se envían comandos de vuelo de forma apresurada, el piloto automático rechaza las instrucciones, es por ello que se utilizó el método *motors_armed_wait()* pa-

ra pausar la secuencia de vuelo hasta que el sistema indique que los motores han sido armados de forma satisfactoria.

Con lo anterior se concluye la descripción de la solución para los problemas enfrentados durante el porteo del Nodo de seguimiento de trayectoria. Ahora, la figura 4.43 presenta una proyección en tres dimensiones de la trayectoria seguida por el dron tras la implementación del algoritmo de misión de vuelo. Realmente no se presenta ningún tipo de novedad, pues con base en la figura, se puede observar que la trayectoria de vuelo fue ejecutada de forma satisfactoria y corresponde a la misma trayectoria definida en la sección destinada al desarrollo del algoritmo por medio de scripts.

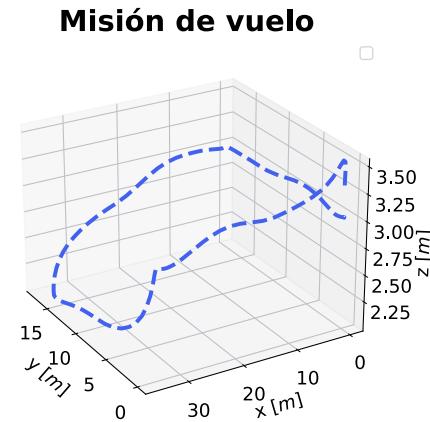


Figura 4.43: Recorrido realizado por el dron

Adicionalmente, retomando las figuras 4.39a, 4.39b, 4.39c y 4.39d es posible observar parte de la trayectoria de vuelo desde la perspectiva de la cámara del dron. Por lo que, se puede decir con certeza que el seguimiento de trayectoria se implementó de manera correcta, logrando que el dron atravesara las 4 compuertas y completando el circuito de forma cíclica.

4.5.3. Integración

Por último, la integración del ambiente no solo conllevó el conjuntar los nodos creados en las secciones anterior, sino que, también se buscó la forma de implementar el SIL de ArduPilot y la simulación de Gazebo a partir de un archivo de lanzamiento en ROS.

Dicho esto, el repositorio del proyecto cuenta con los archivos desarrollados para cada una de las etapas descritas en el capítulo de resultados; es decir, cuenta con todo lo necesario para recrear los resultados reportados en el presente documento y para utilizar el trabajo desarrollado, basta con clonar el repositorio y seguir las instrucciones de configuración del sistema descritas al inicio del capítulo.

A partir de todo lo que se ha discutido en esta sección, la figura 4.44 muestra el diagrama de nodos que se genera al momento realizar el lanzamiento del sistema con ROS. Se puede observar que, en su conjunto, se desarrolló un sistema sencillo con dos nodos; el nodo designado como *waypoints_node* es el nodo encargado de ejecutar la secuencia de vuelo, mientras que el *computer_vision_node* corresponde al nodo en donde se implementa el algoritmos de detección de compuerta. Adicionalmente, es posible observar otro nodo designado con el nombre de *camera_controller*, el cual es creado por el plugin que se encarga de establecer la comunicación entre ROS y gazebo, de hecho, se puede observar el topic por donde se envían los fotogramas generados por la simulación hacia el nodo de visión artificial, el cual está designado como *cam_image_raw*.

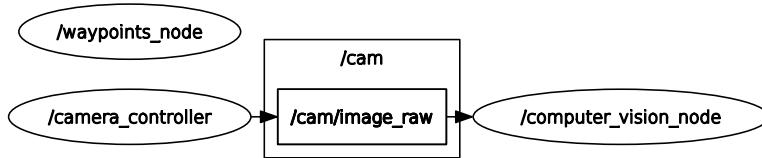


Figura 4.44: Recorrido realizado por el dron

Además, cabe mencionar que a pesar de consultar distintas fuentes, la implementación de la ejecución del ambiente de simulación de Gazebo con un archivo de lanzamiento de ROS no fue posible, pues cuando la simulación se ejecuta de esta forma, no se genera de forma adecuada el ambiente; sin embargo, no existió ningún problema al momento de implementar el SIL de Ardupilot en el lanzador.

Por último, si el usuario siguió las instrucciones de configuración de forma adecuada y cuenta con los archivos del repositorio asociado a este trabajo, debería de ser capaz de ejecutar el sistema propuesto con dos simples comandos

```
$ ros2 launch adf_sil sil.launch.py
```

```
$ gazebo --verbose .../src/gazebo_sim/worlds/adr_circuit.world
```

Capítulo 4. Resultados

El primero comando se encarga de ejecutar el nodo de visión artificial, misión de vuelo y el SIL de ardupilot, mientras que el segundo incializa el ambiente de simulación de Gazebo. El directorio especificado en este segundo comando puede variar dependiendo de donde se haya clonado el repositorio del proyecto.

CAPÍTULO 5

CONCLUSIÓN

El desarrollo de tecnología enfocada a sistemas de navegación autónoma de cuadrotómetros ha tenido un gran auge en los últimos años, pues debido a que estos sistemas combinan un tamaño reducido con una agilidad y autonomía de vuelo bastante buena, son ideales para tareas de reconocimiento, búsqueda y rescate, entre otras, en donde es necesario recorrer zonas angostas a gran velocidad. Sin embargo, el fin último del desarrollo de este campo, es desarrollar sistemas autónomos que sean capaces de hacer frente a los mejores pilotos humanos, de tal forma que se le saque el mayor provecho posible a los recursos con los que cuenta el sistema.

Si bien es cierto que hoy en día existe un gran conjunto de esfuerzos desarrollando soluciones para este tipo de sistemas, el estado del arte del campo de estudio todavía no se encuentra a la altura del desempeño de los pilotos profesionales, o bueno, no sin utilizar una gran cantidad de sensores y cámaras para conocer la odometría del vehículo.

Por otro lado, retomando lo mencionado sobre las competencias en el marco teórico, prácticamente todos los sistemas desarrollados por los equipos pasan por una etapa de validación mediante simulación, e incluso una de las competencias mencionadas se encuentra basada exclusivamente en ambientes simulados. De ahí que exista la necesidad de tener a disposición una variedad de simuladores especializados en este tipo de vehículos.

Entonces, con base en lo ya mencionado y en los objetivos y la metodología definida, se puede concluir que se desarrolló de forma satisfactoria un sistema de misión de

vuelo de un cuadrotor autónomo basado en una ambiente virtual, con un bajo coste de recursos computacionales y completamente gratuito y de código abierto. Los resultados muestran que los algoritmos desarrollados funcionan de manera adecuada o acorde a los requerimientos establecidos para estos.

Hablando sobre el desempeño del algoritmo de visión artificial se puede decir que la detección se realizó adecuadamente, pues la sintonización del rango de color permitió filtrar de mejor manera el fondo de los fotogramas captados por la cámara del dron; sin embargo, el algoritmo no es perfecto, de hecho es bastante sencillo, pues utiliza una combinación de métodos de apertura y cerradura para realizar la detección tras la conversión del modelo de color del fotograma. Además, a partir de lo observado durante la segunda sintonización del rango de color y de los resultados obtenidos con ROS, se pudo apreciar que para que el algoritmo de detección de mejores resultado, es necesario que se encuentre relativamente cerca de la compuerta, pues no es capaz filtrar el fondo a la distancia. Por otro lado, las pruebas hechas dentro de simulación fueron realizadas tomando en cuenta condiciones ideales en el ambiente, como iluminación, viento, ruido, etc., por lo que, es muy probable que en un ambiente real el rendimiento del algoritmo de detección no sea tan eficiente, y más aún si es que existen cambios en la iluminación del ambiente; Sin embargo, tomando en cuenta la simpleza y el bajo consumo de recursos que conlleva la ejecución del algoritmo, los resultados obtenidos fueron más que adecuado para realizar la prueba del framework de simulación.

En cuanto al desempeño del algoritmo de seguimiento de trayectoria, los resultados observados fueron bastante bueno, pues el dron fue capaz de completar el circuito de vuelo pasando a través de todas y cada una de las compuertas. En este caso no hay observaciones más detalladas, pues el algoritmo se comportó de acuerdo a lo esperado; sin embargo, en cuanto a la librería utilizada para comandar el vuelo si existen algunas observaciones, pues uno de los principales problemas que surgió durante el desarrollo del trabajo fue la naturaleza asíncrona de ArduPilot y el script con el algoritmo de seguimiento de trayectoria, pues a pesar de leer de forma detallada la documentación, no se encontró alguna bandera o mensaje que indicara que el piloto automático estaba listo para recibir comandos de vuelo, y a pesar de que la solución implementada funciones, resulta un poco rudimentaria.

Por último, en cuanto a los problemas enfrentados durante el desarrollo del trabajo,

el principal obstáculo fue la curva de aprendizaje de las tecnologías utilizadas, principalmente ROS 2, pues la documentación asociada al framework todavía es algo escasa, y como se mencionó, hoy en día ROS 1 sigue teniendo mucha presencia en las aplicaciones desarrolladas con el framework, de tal forma que no es posible aplicar todo lo documentado por la comunidad para aplicaciones de ROS 2. Además, la documentación de ArduPilot se encuentra bastante desactualizada y también es limitada, por lo que gran parte del trabajo desarrollado para el proyecto ha conllevado una investigación intensiva en foros y otras fuentes poco concurridas, además de experimentación. Entonces, se espera que la documentación proveída en el trabajo sea de gran utilidad para la comunidad, pues representa un compilado de muchos fragmentos esparcidos en distintas fuentes.

5.1. Desarrollos futuros

Está claro que los resultados obtenidos en este proyecto abren las puertas a una serie de mejoras que permitan explotar de mejor manera lo aquí presentado, pues al final de cuentas, al inicio del documento se mencionó que este proyecto sienta las bases de una ambiente de simulación que puede ser explotado todavía más allá del alcance de este trabajo.

Entre las posibles mejoras se tiene:

- El diseño de un circuito de vuelo con una trayectoria más compleja
- La implementación de un algoritmo de visión artificial basado en redes neuronales convolucionales
- La integración de los algoritmos de segmentación y seguimiento de trayectoria a partir del uso de servicios en ROS, de tal forma que la aplicación de visión por computadora participe de forma activa para la definición la trayectoria de vuelo del dron
- Uso de la librería de MAVROS en lugar de PymavLink para el envío de comandos de vuelo y recepción de información sobre el vehículo, pues es de más bajo nivel y se integra de forma nativa al concepto de nodos en ROS.

Capítulo 5. Conclusión

- Implementación de la arquitectura en hardware físico. Debido a que el algoritmo de visión artificial propuesto utiliza funciones básicas de morfología matemática, no requiere el uso de una tarjeta gráfica para ser ejecutado, por lo que el sistema propuesto puede ser implementado en una computadora de placa única, como una Raspberry Pi.

BIBLIOGRAFÍA

ArduPilot-Dev-Team. (2022). Ardupilot. <https://ardupilot.org>. Accessed: 2022-02-17.

Bradski, G. & Kaehler, A. (2008). *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc..

Cabrera-Ponce, A. A., Rojas-Perez, L. O., Carrasco-Ochoa, J. A., Martinez-Trinidad, J. F., & Martinez-Carranza, J. (2019). Gate detection for micro aerial vehicles using a single shot detector. *IEEE Latin America Transactions*, 17(12), 2045–2052.

Dey, S. (2020). *Python Image Processing Cookbook*. Packt Publishing.

Ermakov, V. (2021). mavros. <http://wiki.ros.org/mavros>. Accessed: 2022-01-04.

Ferrari, P. (2018). Ssd: Single-shot multibox detector implementation in keras. https://github.com/pierluigiferrari/ssd_keras.

Foehn, P., Brescianini, D., Kaufmann, E., Cieslewski, T., Gehrig, M., Muglikar, M., & Scaramuzza, D. (2020). Alphapilot: Autonomous drone racing. *arXiv preprint arXiv:2005.12813*.

Foehn, P., Romero, A., & Scaramuzza, D. (2021). Time-optimal planning for quadrotor waypoint flight. *Science Robotics*, 6(56).

BIBLIOGRAFÍA

- García, G. B., Suarez, O. D., Aranda, J. L. E., Tercero, J. S., Gracia, I. S., & Enano, N. V. (2015). *Learning image processing with OpenCV*. Packt Publishing Birmingham.
- Guerra, W., Tal, E., Murali, V., Ryou, G., & Karaman, S. (2019). FlightGoggles: Photo-realistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*: IEEE.
- Johnson, E. (2018). Iq simulations. https://github.com/Intelligent-Quads/iq_sim.
- Johnson, E. (2022). Flight modes. <https://ardupilot.org/copter/docs/flight-modes.html>.
- Kaufmann, E., Loquercio, A., Ranftl, R., Dosovitskiy, A., Koltun, V., & Scaramuzza, D. (2018). Deep drone racing: Learning agile flight in dynamic environments. In *Conference on Robot Learning* (pp. 133–145).: PMLR.
- Loquercio, A., Maqueda, A. I., Del-Blanco, C. R., & Scaramuzza, D. (2018). Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2), 1088–1095.
- Lorenz Meier, Andreas Antener, t. (2021). Mavlink developer guide. <https://mavlink.io/en/>. Accessed: 2022-01-04.
- Madaan, R., Gyde, N., Vemprala, S., Brown, M., Nagami, K., Taubner, T., Cristofalo, E., Scaramuzza, D., Schwager, M., & Kapoor, A. (2020). Airsim drone racing lab. In *NeurIPS 2019 Competition and Demonstration Track* (pp. 177–191).: PMLR.
- Mellinger, D. & Kumar, V. (2011). Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE international conference on robotics and automation* (pp. 2520–2525).: IEEE.
- Moon, H., Martinez-Carranza, J., Cieslewski, T., Faessler, M., Falanga, D., Simovic, A., Scaramuzza, D., Li, S., Ozo, M., De Wagter, C., et al. (2019). Challenges and implemented technologies used in autonomous drone racing. *Intelligent Service Robotics*, 12(2), 137–148.
- Moon, H., Sun, Y., Baltes, J., & Kim, S. J. (2017). The iros 2016 competitions. *IEEE Robotics and Automation Magazine*, 24(1), 20–29.

BIBLIOGRAFÍA

- Mueller, M. W., Hehn, M., & D'Andrea, R. (2013). A computationally efficient algorithm for state-to-state quadrocopter trajectory generation and feasibility verification. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 3480–3486).: IEEE.
- OPAL-RT-TECHNOLOGIES (2022a). Hardware in the loop. <https://www.opal-rt.com/hardware-in-the-loop/>. Accessed: 2022-02-28.
- OPAL-RT-TECHNOLOGIES (2022b). Rapid control prototyping. <https://www.opal-rt.com/rapid-control-prototyping/>. Accessed: 2022-02-28.
- OPAL-RT-TECHNOLOGIES (2022c). Software in the loop. <https://www.opal-rt.com/software-in-the-loop/>. Accessed: 2022-02-28.
- Open-Robotics (2014). <http://gazebosim.org/>.
- Open-Robotics (2021a). Installing ros 2 on ubuntu linux. <https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Binary.html>. Accessed: 2022-01-12.
- Open-Robotics (2021b). Open robotics. <https://www.openrobotics.org/>.
- Open-Robotics (2021c). Ros 2 documentation: foxy documentation. <https://docs.ros.org/en/foxy/index.html>.
- OpenCV-Team (2021). Opencv: Introduction. <https://opencv.org>. Accessed: 2022-02-17.
- Ramírez Linarez, A. (2022). Axolotsil. <https://github.com/MOVAX19/AxolotSIL>.
- Robotics, O. (2021). Building ros 2 on ubuntu linux – ros 2 documentation: foxy documentation. <https://docs.ros.org/en/foxy/Installation/Ubuntu-Development-Setup.html>.
- Rojas-Perez, L. O. & Martinez-Carranza, J. (2020). Deepilot: A cnn for autonomous drone racing. *Sensors*, 20(16), 4524.
- Rojas-Perez, L. O. & Martinez-Carranza, J. (2021). On-board processing for autonomous drone racing: An overview. *Integration*.

BIBLIOGRAFÍA

- Saravanakumar, S., Vadivel, A., & Ahmed, C. S. (2011). Multiple object tracking using hsv color space. In *Proceedings of the 2011 International Conference on Communication, Computing & Security* (pp. 247–252).
- Shah, S., Dey, D., Lovett, C., & Kapoor, A. (2017). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*.
- Song, Y., Naji, S., Kaufmann, E., Loquercio, A., & Scaramuzza, D. (2020). Flightmare: A flexible quadrotor simulator. In *Conference on Robot Learning*.
- Stevens, J.-L. (2021). Autonomous visual navigation of a quadrotor vtol in complex and dense environments. *Jean-Luc Stevens*.
- Xia, X., Xu, C., & Nan, B. (2017). Inception-v3 for flower classification. In *2017 2nd International Conference on Image, Vision and Computing (ICIVC)* (pp. 783–787).: IEEE.