

UNIVERSIDAD AERONÁUTICA EN QUERÉTARO

DISEÑO Y SIMULACIÓN DE UN SISTEMA DE VISIÓN ARTIFICIAL PARA EL RECONOCIMIENTO DE OBSTÁCULOS Y GENERACIÓN DE TRAYECTORIAS DE VUELO DE 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, Marzo 2021



RESUMEN



GLOSARIO

ÍNDICE GENERAL

1. Introducción	9
1.1. Antecedente históricos	9
1.2. Objetivos	10
1.2.1. Objetivo general	10
1.2.2. Objetivo específicos	10
1.3. Justificación	11
1.4. Metodología	13
1.5. Límites y alcances	14
1.5.1. Alcances	14
1.5.2. Límites	14
1.6. Estructura de la tesis	15
2. Estado del Arte	16
3. Marco Teórico	26
3.1. Competencias de Drones Autónomos	26
3.1.1. Autonomous Drone Racing	27
3.1.2. AlphaPilot Challenge	28
3.1.3. Game of Drones	30
3.2. Robot Operating System (ROS)	31

ÍNDICE GENERAL

3.2.1.	Concepto	31
3.2.2.	Conceptos básicos de ROS	32
3.2.3.	Las limitaciones de ROS 1	33
3.2.4.	¿Por qué ROS 2?	34
3.2.5.	Diferencias entre ROS 1 y ROS 2	36
3.3.	Software in The Loop	37
3.4.	Gazebo	37
3.4.1.	Concepto	37
3.4.2.	Historia	38
3.4.3.	¿Por qué Gazebo?	39
4.	Resultados	41
4.1.	Configuración del Sistema	41
4.1.1.	Instalación de ROS 2	42
4.1.2.	Instalación de OpenCV	46
4.1.3.	Instalación de Gazebo	47
4.1.4.	Instalación de ArduPilot SITL Simulator	50
4.1.5.	Instalación de Pymavlink	58
4.2.	Gazebo	59
4.3.	Sistema de Visión Artificial	63
4.4.	Pymavlink	64
4.4.1.	Conexión entre Pymavlink y ArduPilot	65
4.4.2.	Configuración de modo de vuelo	68
4.4.3.	Secuencia de despegue	70
4.4.4.	Seguimiento de trayectoria	72
4.4.5.	Misión de vuelo	75
4.4.6.	Aterrizaje (extra)	76
4.5.	ROS	77
Bibliografía		83

ÍNDICE DE FIGURAS

2.1. Validación del algoritmo de detección de compuertas basado en SSD7 [1]	17
2.2. Imagen compuesta del vuelo del dron [11]	18
2.3. Dron con raqueta incorporada [14]	19
2.4. Imagen compuesta del vuelo del dron [12]	19
2.5. Resultados de la implementación del equipo TU Delft [12]	20
2.6. Propuesta desarrollada por el equipo UNIST, [12]	22
2.7. Funcionamiento de la red DeepPilot[20]	23
2.8. Funcionamiento de DeepPilot[5]	24
2.9. Funcionamiento de DeepPilot[23]	25
4.1. Características del sistema en donde se desarrolló el proyecto	42
4.2. Nodos de demostración incluidos en la instalación de ROS 2	46
4.3. Mensaje de validación para la instalación de OpenCV	48
4.4. Ficha técnica de la versión de Gazebo	48
4.5. Proyecto vacío generado al inicializar Gazebo	49
4.6. Proyecto de demostración en Gazebo que incluye el plug-in para comunicarse con ROS	50
4.7. Movimiento de la simulación en Gazebo con plugin de comunicación de ROS	51
4.8. Lista de simulaciones de demostración para el uso de Gazebo con ROS 2 .	52

Índice de figuras

4.9.	Prueba de ejecución del framework de SITL de ArduPilot	55
4.10.	Terminal de información del sistema de ArduPilot	56
4.11.	Consola de comunicación de ArduPilot	56
4.12.	Modelo en Gazebo del dron Iris	58
4.13.	Prueba de integración entre Gazebo y el SITL de ArduPilot	59
4.14.	Validación de instalación de Pymavlink	60
4.15.	Esquema detallado del circuito de vuelo elaborado en simulación.	61
4.16.	Modelos de compuerta proveídos por [20]	62
4.17.	Modelo de dron iris con cámara frontal provisto por	63
4.18.	Capturas del ambiente de simulación implementado	64
4.19.	Imágenes utilizadas para la sintonización del rango de color	65
4.20.	Detección de compuerta con el primer rango de color	66
4.21.	Resintonización del rango de color	67
4.22.	Intento de sintonización utilizando la cuarta imagen de referencia como base	68
4.23.	Datos de conexión de ArduPilot	69
4.24.	Datos obtenidos a partir de la conexión entre ArduPilot y Pymavlink . .	70
4.25.	Ejecución del script para la configuración del modo de vuelo	71
4.26.	Comportamiento de la simulación ante el comando de despegue	72
4.27.	Respuesta en la altura para el comando de despegue.	72
4.28.	Gráfica de velocidad de despegue.	73
4.29.	Etapas de la prueba de seguimiento de trayectoria	76
4.30.	Recorrido realizado por el dron	77
4.31.	Gráficas de desplazamiento para los tres ejes	78
4.32.	Comportamiento de la velocidad lineal de vuelo en los tres ejes	79
4.33.	Seguimiento de misión de vuelo completa	80
4.34.	Recorrido realizado por el dron	80
4.35.	Gráficas de desplazamiento para los tres ejes	81
4.36.	Comportamiento de la velocidad lineal de vuelo en los tres ejes	82



ÍNDICE DE TABLAS

3.1.	Especificaciones de sensores utilizados en el APC [4]	30
3.2.	Lista de distribuciones de ROS 2	36
4.1.	Características técnicas de la laptop Aspire E5-575	41
4.2.	Relación entre etapa y su código fuente	68
4.3.	Parámetros de la instrucción para seguimiento de waypoints	74
4.4.	Tipos de máscara para el comando de seguimiento de trayectoria	75



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 rotores 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 Ivan Jerome desarrollaron una aeronave con una estructura en forma de equis y rotores 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 quadrotores grandes para uso civil y militar. Este prototipo contaba con dos motores, que controlan el giro de dos rotores, cada uno, a partir de lo anterior, el control de la aeronave se lograba al variar el empuje proporcionado por los rotores.

1.2. Objetivos

1.2.1. Objetivo general

Proponer e implementar en simulación un algoritmo de detección de compuertas rectangulares mediante visión artificial para la definición y control de trayectoria de 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. Lo anterior 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 im-

plementen 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, para definir la trayectoria de vuelo de un dron autónomo con el fin de que sea capaz de completar un circuito definido. El entrenamiento e implementación se realizan 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. 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 empleados y 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, 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 manuales con el modelo de dron definido dentro de un circuito de prueba compuesto por compuertas. 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; se utilizan las imágenes recopiladas para el entrenamiento del algoritmo de visión artificial.

Cuando el algoritmo de visión artificial proporciona una identificación adecuada del tipo de compuerta utilizada, se implementa el algoritmo con base en la arquitectura definida. Se realiza la validación del algoritmo en otro circuito de vuelo; a lo largo de la simulación, existe un intercambio de información constante entre la simulación y el algoritmo de visión artificial, 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 el centro de la compuerta más cercana y devuelve comandos de vuelo a la simulación para definir una ruta de vuelo que permita que el dron sea capaz de volar a través de la compuerta identificada y finalizar el circuito de forma autónoma.

Se reportan los resultados obtenidos y las posibles mejoras para el proyecto en su conjunto

1.5. Límites y alcances

1.5.1. Alcances

Se implementa la arquitectura de red neuronal convolucional (RNC) DeepPilot, la cual toma capturas de la única cámara a bordo del dron y predice cuatro comandos de vuelo ($\phi, \theta\psi, h$) como salida. La RNC es entrenada a partir de un dataset proveído por los autores de la arquitectura y que contiene un gran conjunto de imágenes obtenidas a partir de simulación, las cuales están asociadas a ciertos comandos de vuelo. La arquitectura es evaluada dentro de un entorno de simulación realizado en simulador 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 propio de ROS2 que contiene el algoritmo y arquitectura de DeepPilot. Además, se documenta de forma detallada la configuración realizada para la creación del ambiente de simulación, especificando la integración entre Gazebo, ROS y Python 3 para la evaluación de la arquitectura de DeepPilot. Por último, el proyecto en su conjunto se distribuye bajo el paradigma de código abierto.

1.5.2. Límites

A diferencia de las contribuciones y proyectos más populares dentro de la comunidad de las carreras de drones autónomos, en donde se utiliza ROS1 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, al momento de escritura del trabajo, la versión 11. Por lo que es muy posible que algunos plugins tanto de ROS como de Gazebo, 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 y se asume que las compuertas se encuentran de forma paralela a la cámara del dron, y es necesario que siempre exista una compuerta visible después de haber cruzado por otra.

1.6. Estructura de la tesis

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, [1] 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 con una arquitectura base de Single Shot

Detector de 7 capas (SSD7[3]). 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 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 [1], 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.

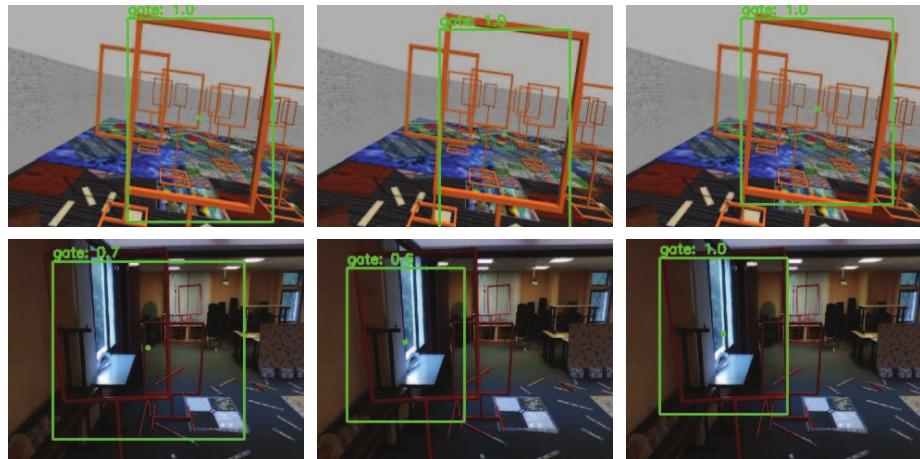


Figura 2.1: Validación del algoritmo de detección de compuertas basado en SSD7 [1]

Por otro lado, [11] 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 [11]

Adicionalmente, Mueller et al.(2013)[14] 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.

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.



Figura 2.3: Dron con raqueta incorporada [14]

El trabajo realizado por Moon et al.(2019) [12] 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 [12]. Iniciando por el equipo ganador de la competencia, el equipo 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.

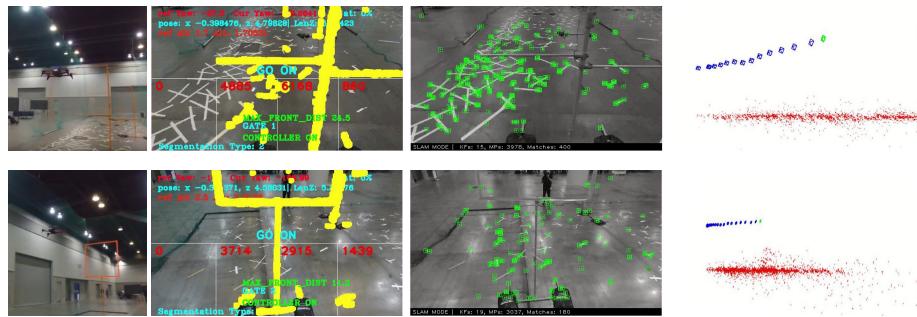


Figura 2.4: Imagen compuesta del vuelo del dron [12]

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 el dron del equipo de UZH navevara 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. 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. 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. 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.



Figura 2.5: Resultados de la implementación del equipo TU Delft [12]

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 (LOS 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 [24], 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 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.

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) [7]; desarrollaron 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

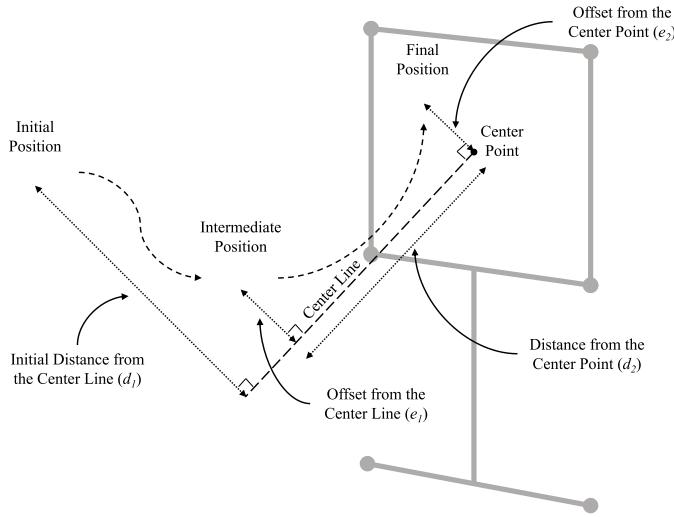


Figura 2.6: Propuesta desarrollada por el equipo UNIST, [12]

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)[8], 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)[20] 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, se 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 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, se utiliza ROS para la gestión de los procesos en la computadora de vuelo y se hace la comparación con la arquitectura de otras redes neuronales profundas. La figura 2.7 muestra un diagrama general de la implementación de [20] 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.

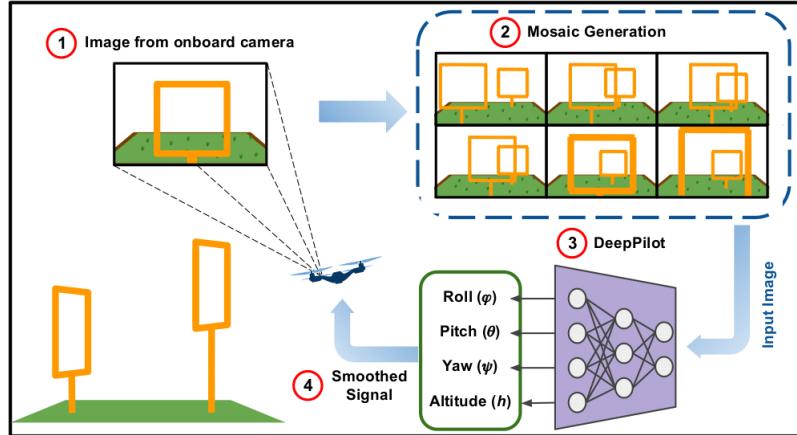


Figura 2.7: Funcionamiento de la red DeepPilot[20]

En cuanto a soluciones particulares para algoritmos de seguimiento de trayectoria, [5] 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.



Figura 2.8: Funcionamiento de DeepPilot[5]

Por último, mencionando un ejemplo que va más allá de las competencias de drones autónomos, [23] 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 efectividad 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 sentir 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.

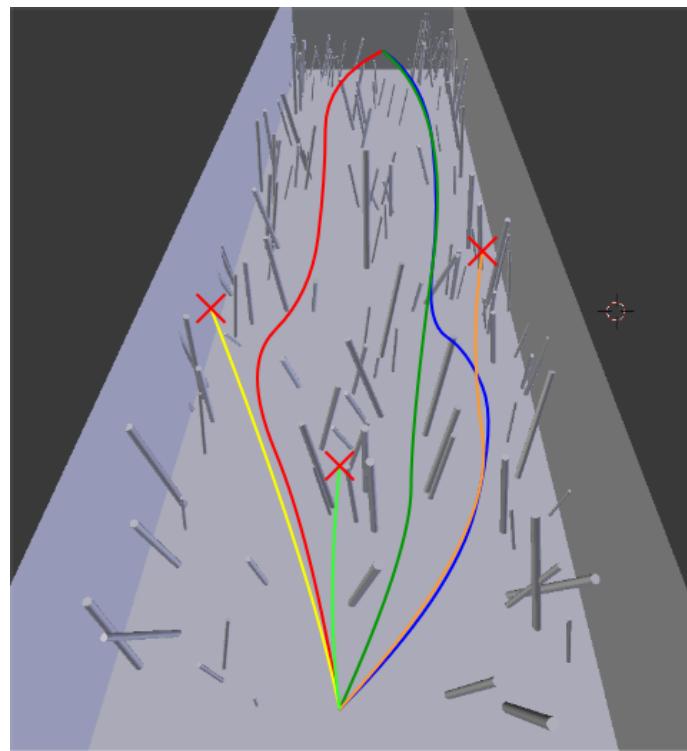


Figura 2.9: Funcionamiento de DeepPilot[23]

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)*[13], llevado a cabo cada año en la Conferencia Internacional de Sistemas y Robots Inteligentes

(IROS, por sus siglas en inglés), el *AlphaPilot Challenge (APC)*[4], organizado por Lockheed Martin en colaboración con Nvidia y la Liga de Carrera de Drones (DRL); y *Game of Drones (GOD)*[10], 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 [21].

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[4].

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 [4]

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[4] en conjunto con la Universidad de Stanford y la Universidad de Zúrich buscaron fomentar el avance de las tecnologías utilizadas en 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[22], 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[17], ROS (del inglés, Robot Operating System) es un conjunto de herramientas y librerías de software para robótica desarrolladas por Open Robotics[19] 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, el paradigma de diseño de ROS 1 tenía varias limitantes para las necesidades emergentes de la industria y nuevas aplicaciones.

Entre los nuevos casos de uso que dirigen el desarrollo de ROS 2 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 conservaran, pero ofreciendo una gran mejora y experiencia al usarla. Esto significa que la API de ROS 1 no será 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 encuentran activas dos distribuciones, Melodic Morenia y Noetic Ninjemys. Siendo esta última la versión final de ROS 1, cuyo soporte ter-

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

minará 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 soporte por parte de Open Robotics.

3.2.5. Diferencias entre ROS 1 y ROS 2

A continuación se resumen los cambios más significativos entre las versiones de ROS.

Plataformas: ROS 1 solo trabaja de forma nativa en Ubuntu; ROS 2 soporta Ubuntu, Mac OS X y Windows 10.

Lenguajes de programación: ROS 1 trabaja con C++03 y Python 2; ROS 2 usa C++11 de forma extensiva y usa algunas partes de C++14, en un futuro también trará con C++17. Además, Python a partir de la versión 3.5.

Lógica de comunicación: ROS 1 utiliza una capa de comunicación diseñada prácticamente desde cero (XML-RPC); ROS 2 adopta un protocolo ya definido (DDS) para el manejo de sus comunicaciones.

Sistema de compilación:

3.3. Software in The Loop

De acuerdo con su documentación oficial [9], 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] 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.

3.4. Gazebo

3.4.1. 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 [15], 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.

3.4.2. 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 [15]. 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.

3.4.3. ¿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 Sistema

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.

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.

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

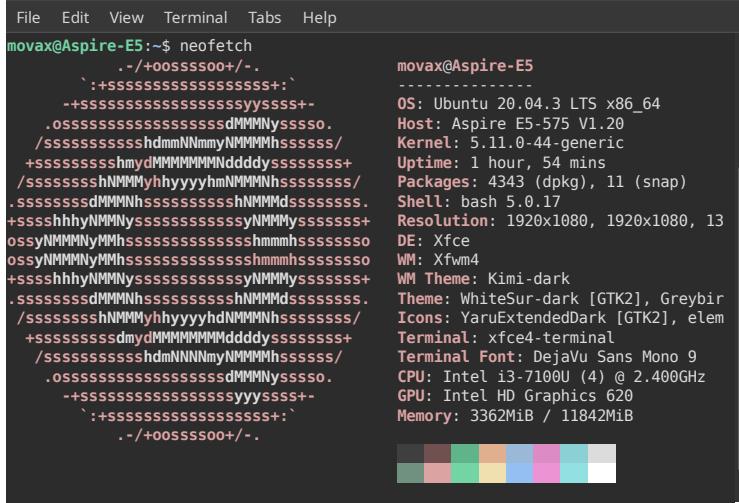


Figura 4.1: Características del sistema en donde se desarrolló el proyecto

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[18] 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 [16].

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 x, 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-setuptools \  
python3-wheel
```

```
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 \
libcunit-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++

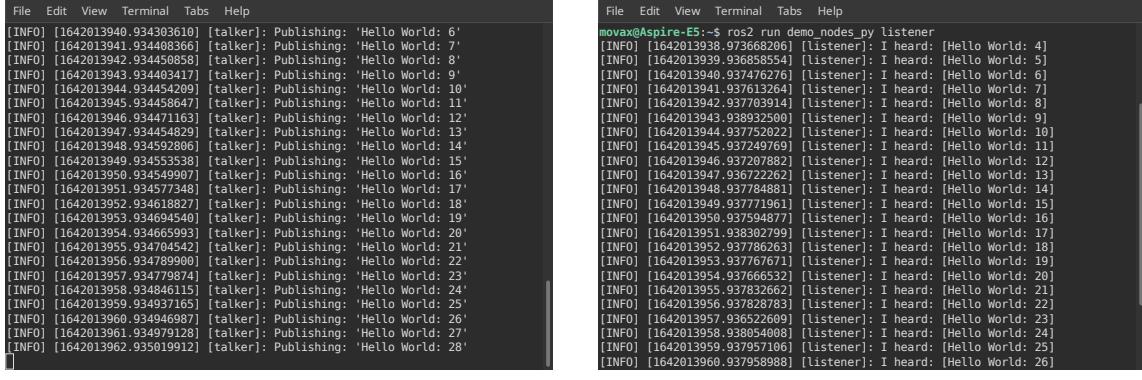
```
$ ros2 run demo_nodes_cpp talker
```

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, 935009912] [talker]: Publishing: 'Hello World: 28'

File Edit View Terminal Tabs Help
novalx@Aspire-E5:~$ ros2 run demo_nodes_py listener
[INFO] [1642013938, 937668206] [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, 937703914] [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, 938054088] [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

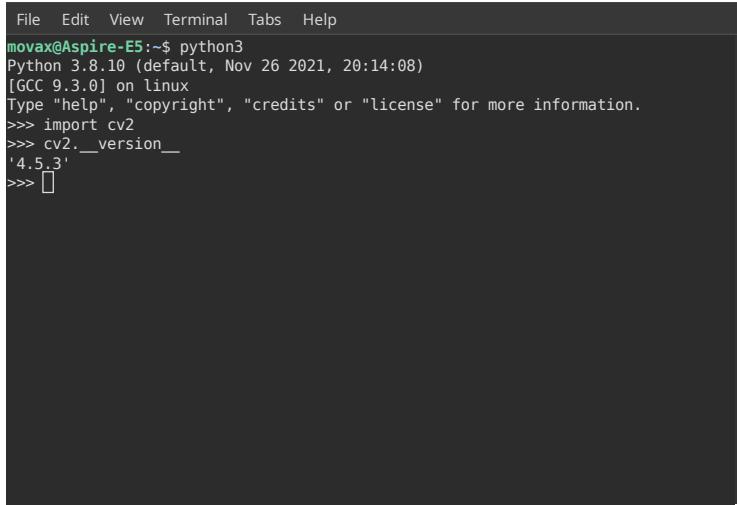
Como se mencionó en el marco teórico, Gazebo es un ambiente de simulación independiente; es decir, no necesita de ROS o ArduPilot para funcionar. Sin embargo, para habilitar la comunicación entre los nodos de ROS y Gazebo, es recomendable realizar la instalación de Gazebo utilizando los repositorios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

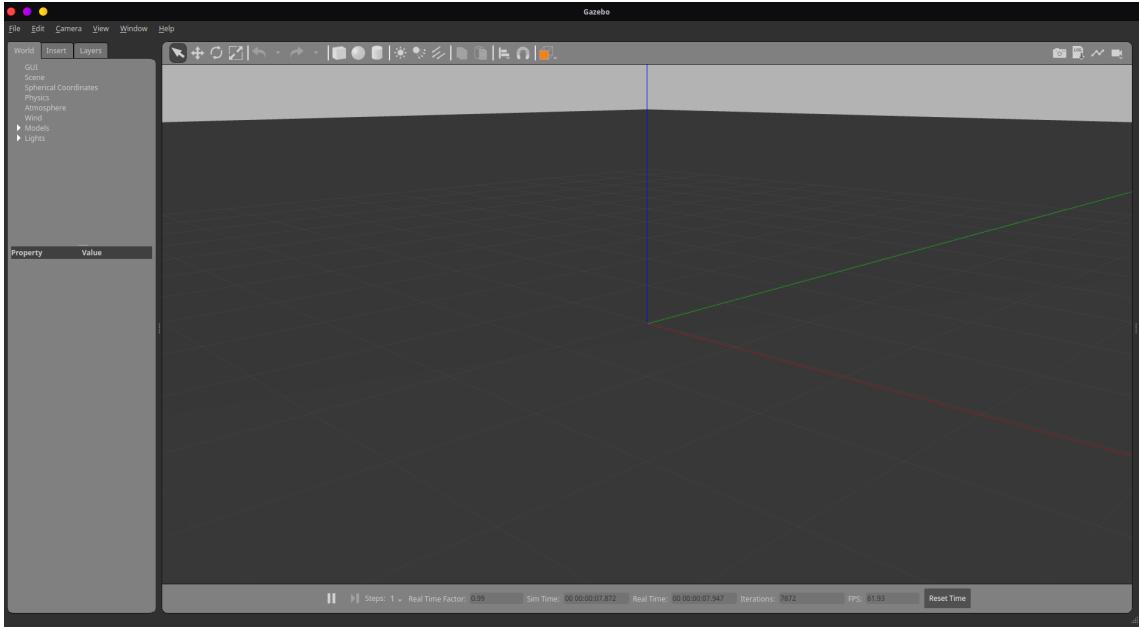


Figura 4.5: Proyecto vacío generado al inicializar Gazebo

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

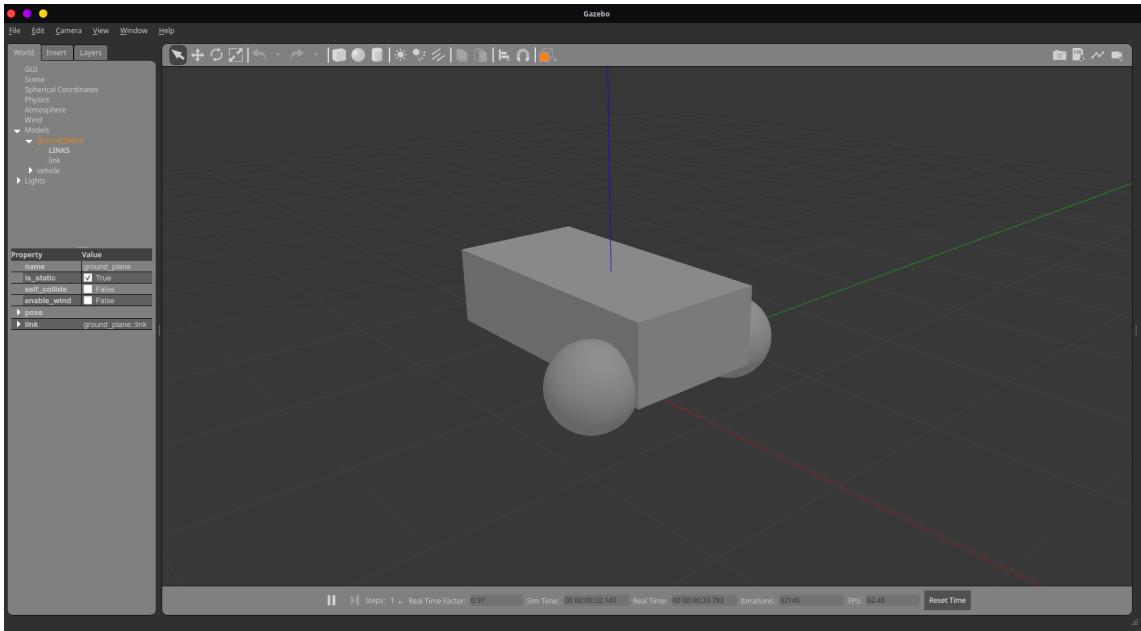


Figura 4.6: Proyecto de demostración en Gazebo que incluye el plug-in para comunicarse con ROS

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

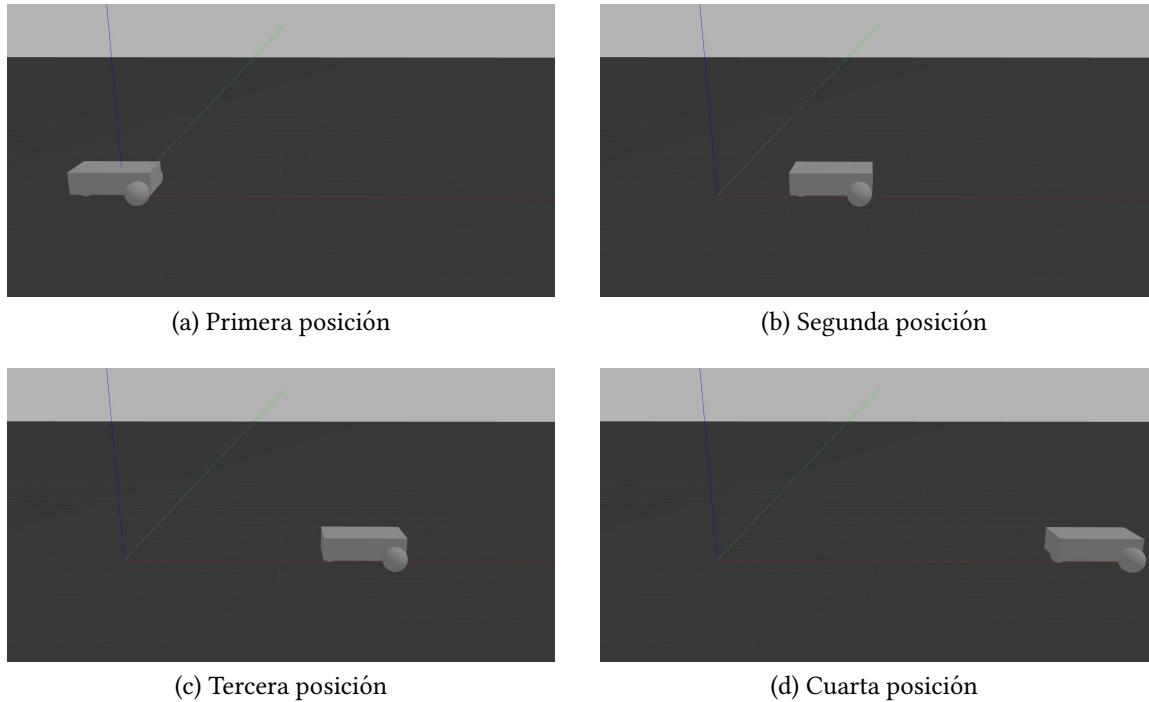


Figura 4.7: Movimiento de la simulación en Gazebo con plugin de comunicación de ROS

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
```

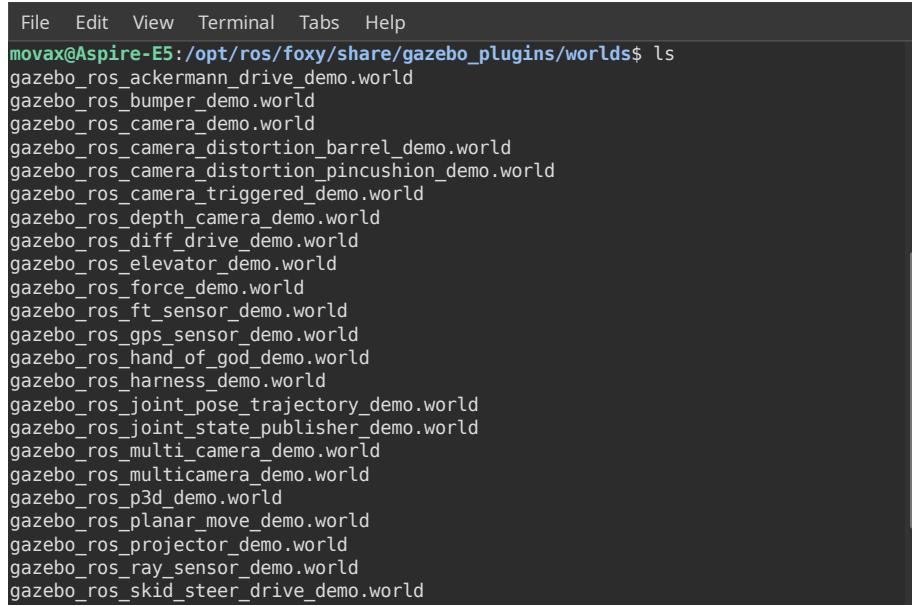
2. Instalar las herramientas necesarias para la compilación de ArduPilot

```
$ 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

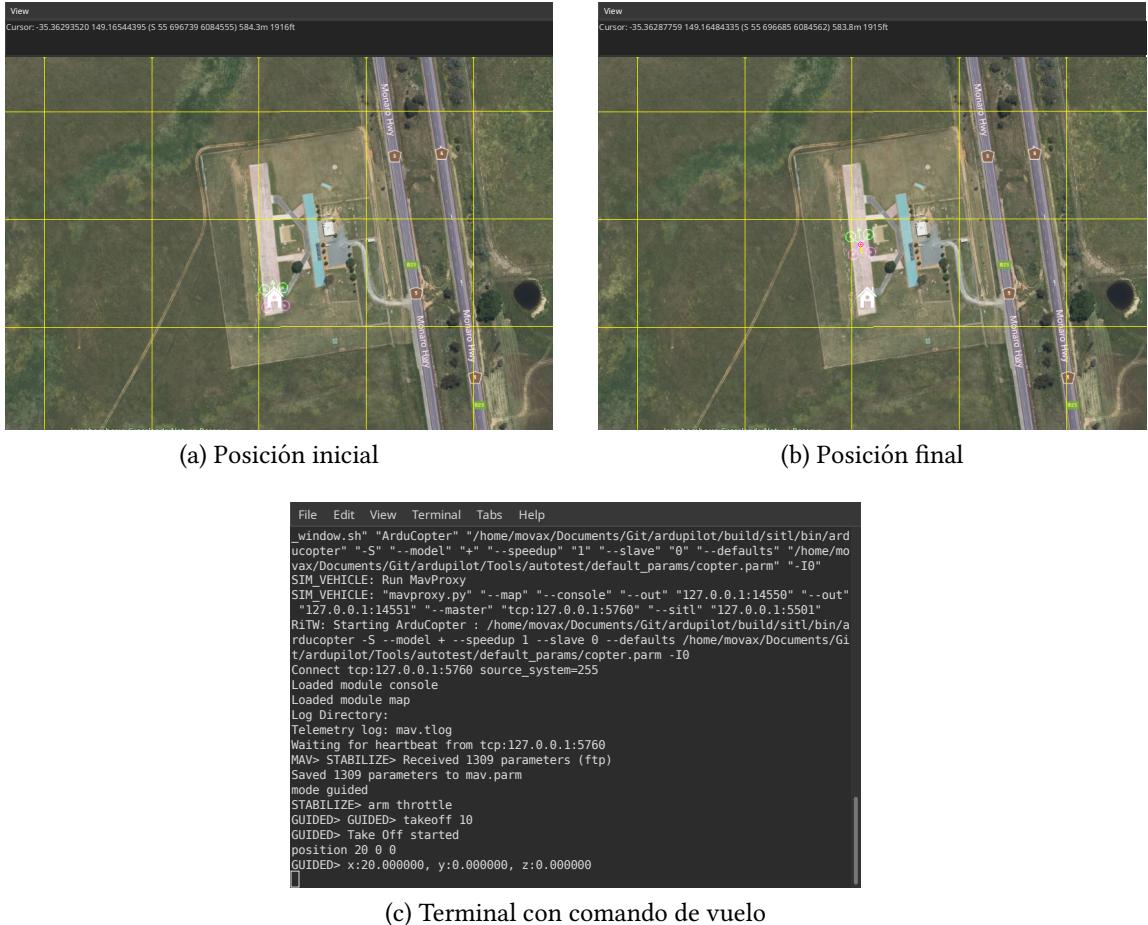


Figura 4.9: Prueba de ejecución del framework de SITL de ArduPilot

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

Figura 4.10: Terminal de información del sistema de ArduPilot

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

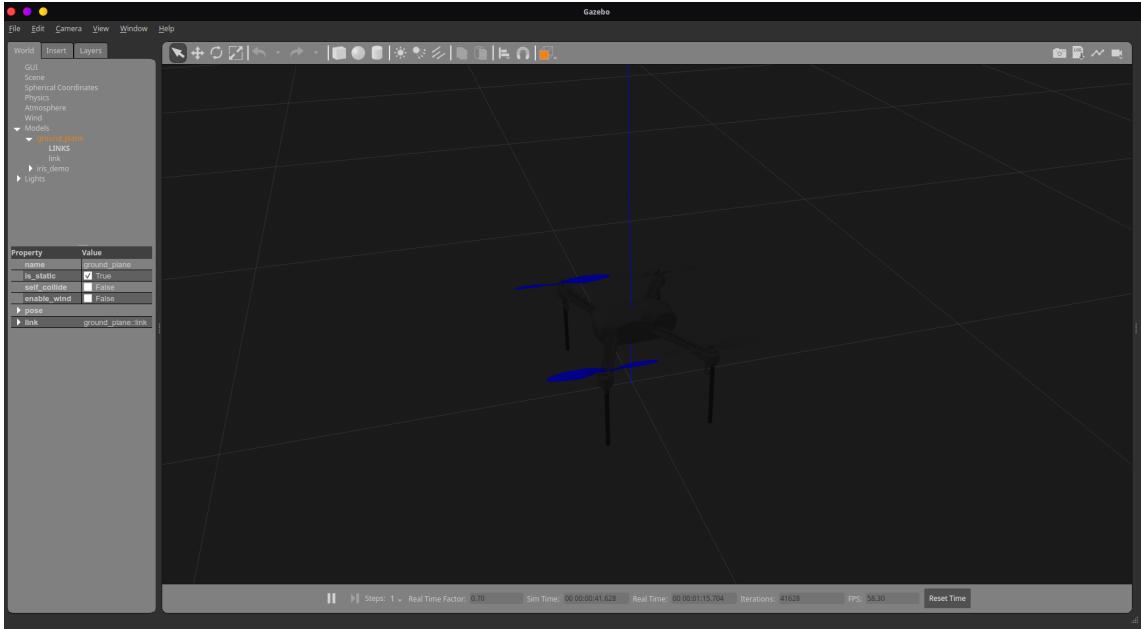


Figura 4.12: Modelo en Gazebo del dron Iris

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

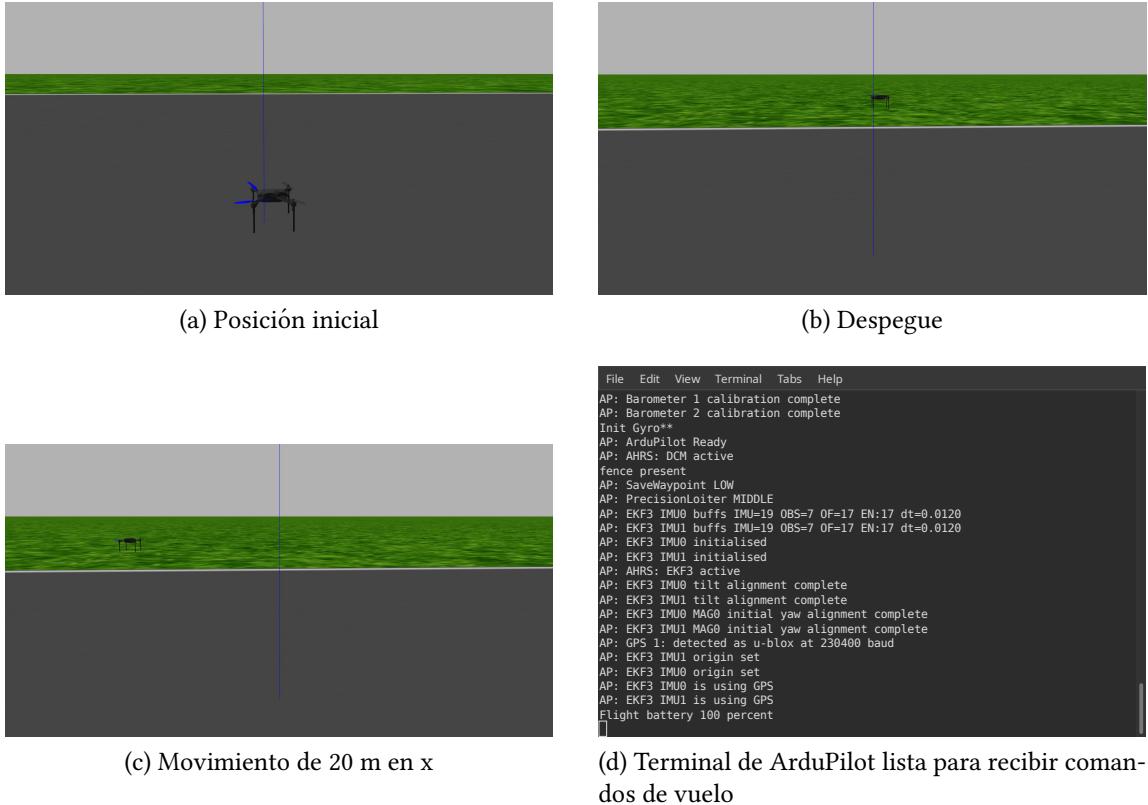


Figura 4.13: Prueba de integración entre Gazebo y el SITL de ArduPilot

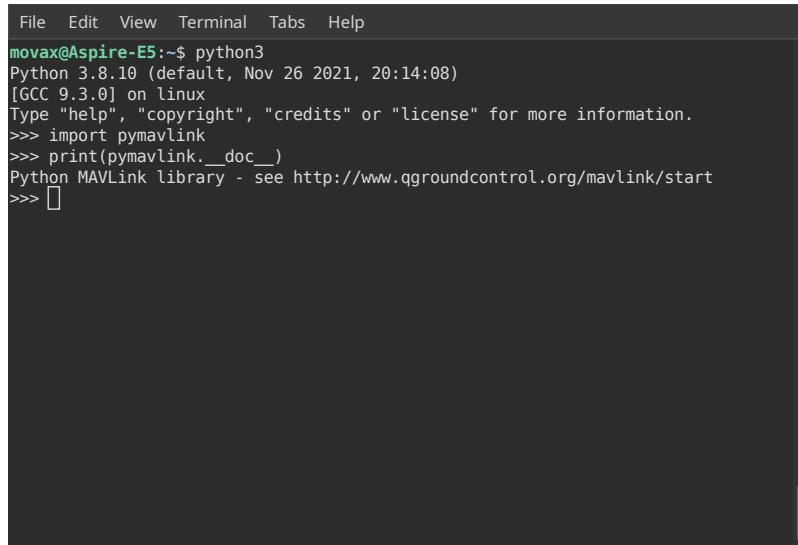
```
$ python3
>>> import pymavlink
>>> print(pymavlink.__doc__)
```

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

4.2. Gazebo

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 el 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 [20], 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 sobrelevar el problema anterior; para que el menú se muestre 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.

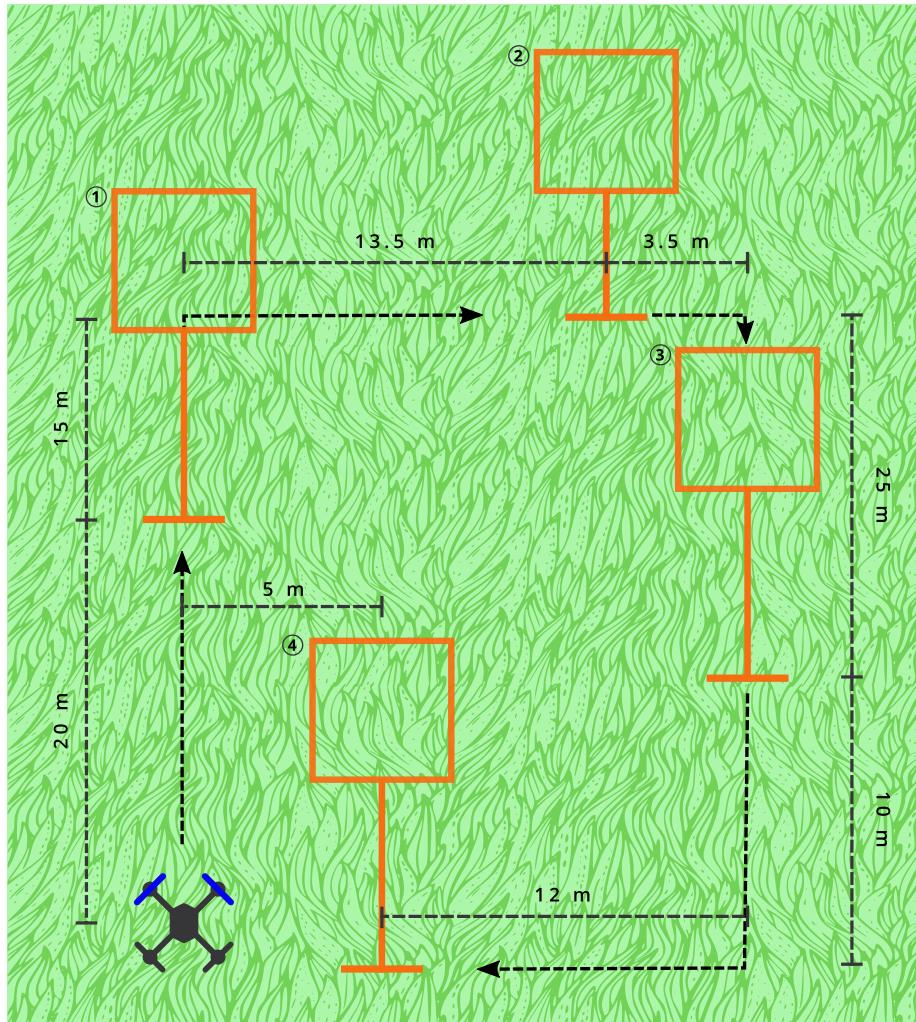


Figura 4.15: Esquema detallado del circuito de vuelo elaborado en simulación.

Entonces, para solucionar este segundo problema se debe de crear un archivo utilizando el comando `touch` con la extensión `.world`. Lo anterior generar 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 y el modelo incluido en la simulación de demostración del plugin de Ardupilot para Gazebo, es la posición 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.

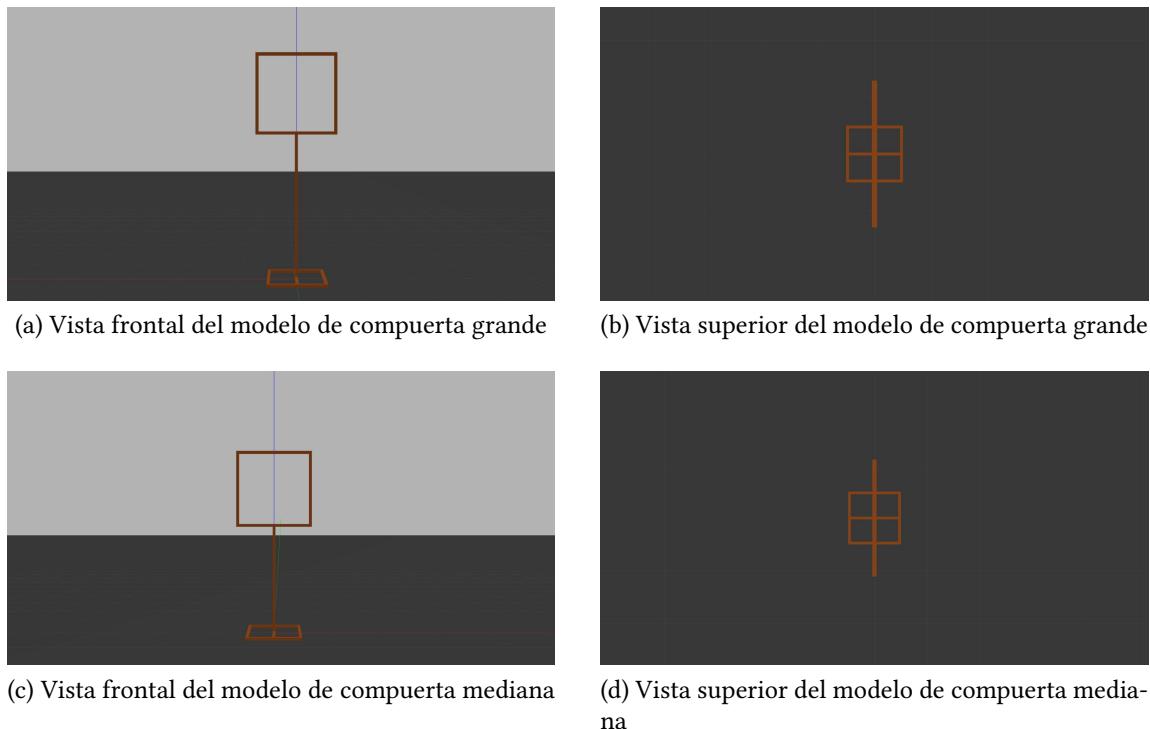


Figura 4.16: Modelos de compuerta proveídos por [20]

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 pano-

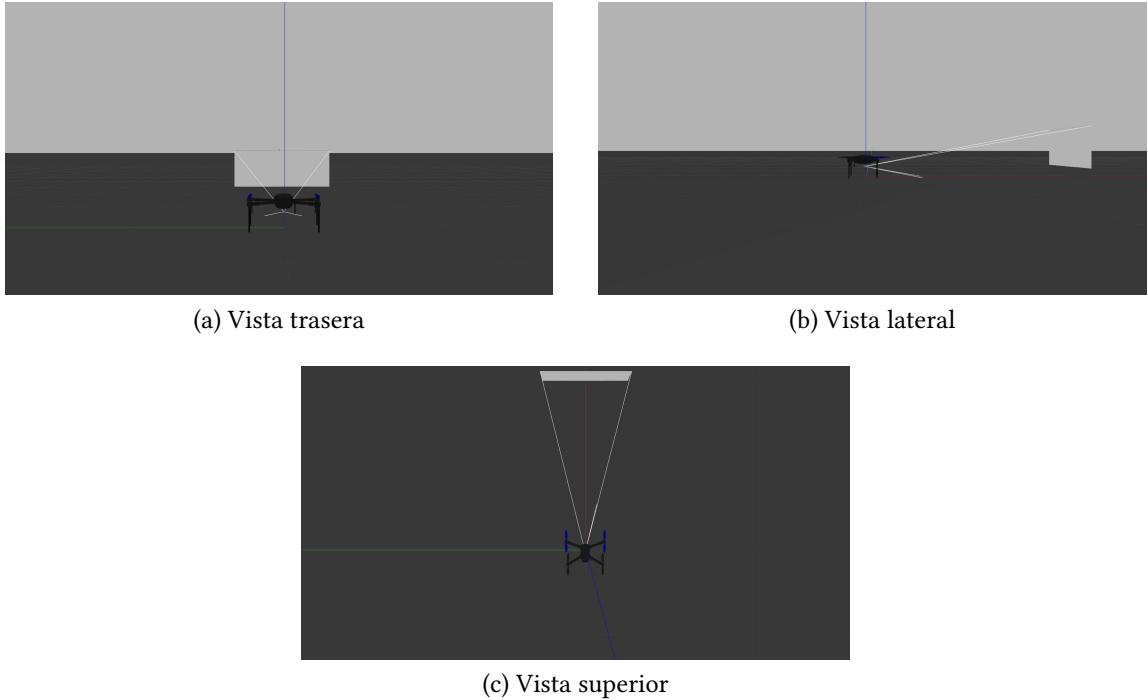


Figura 4.17: Modelo de dron iris con cámara frontal provisto por

rama 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.

4.3. Sistema de Visión Artificial

Para realizar la sintonización del rango de valores para el espacio de color HSV se utilizaron algunas imágenes tomadas dentro del proyecto de simulación elaborado en Gazebo.

Debido al alcance y las limitaciones del presente trabajo, el circuito elaborado es sencillo y consta de 4 compuertas con el diseño utilizado en el IROS. Tres de las compuertas tienen una altura de 3.5 m y la compuerta restante mide 2.5 m de altura. Las cuatro compuertas se encuentran distanciadas por longitudes distintas y describen un arreglo

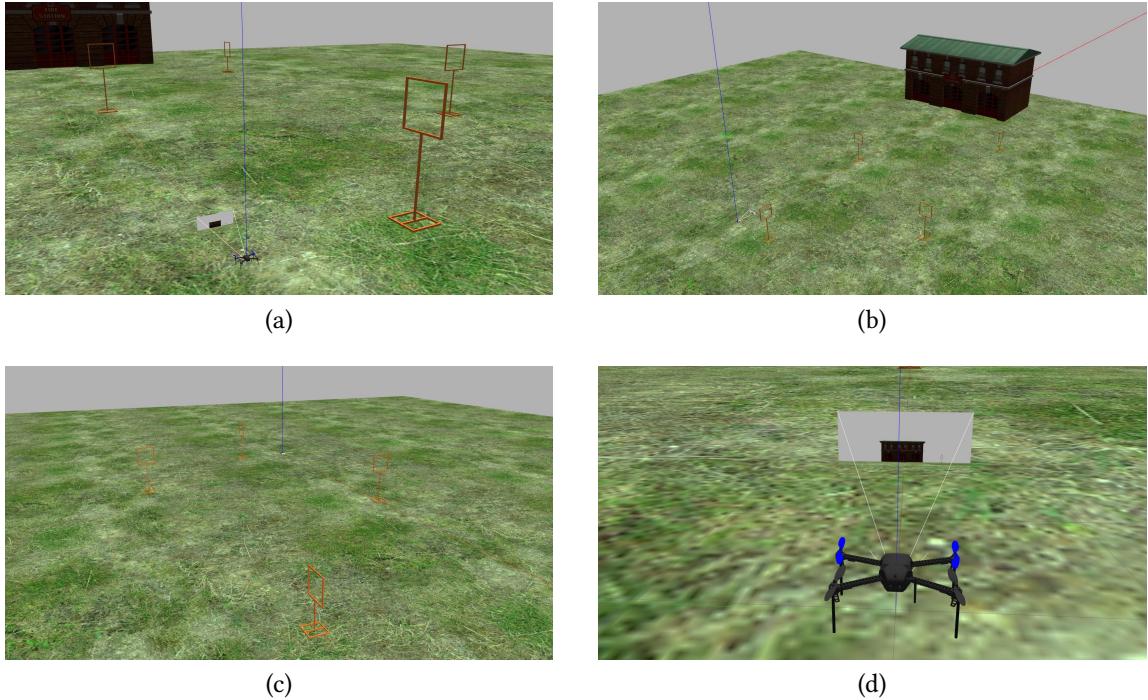


Figura 4.18: Capturas del ambiente de simulación implementado

rectangular.

4.4. Pymavlink

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

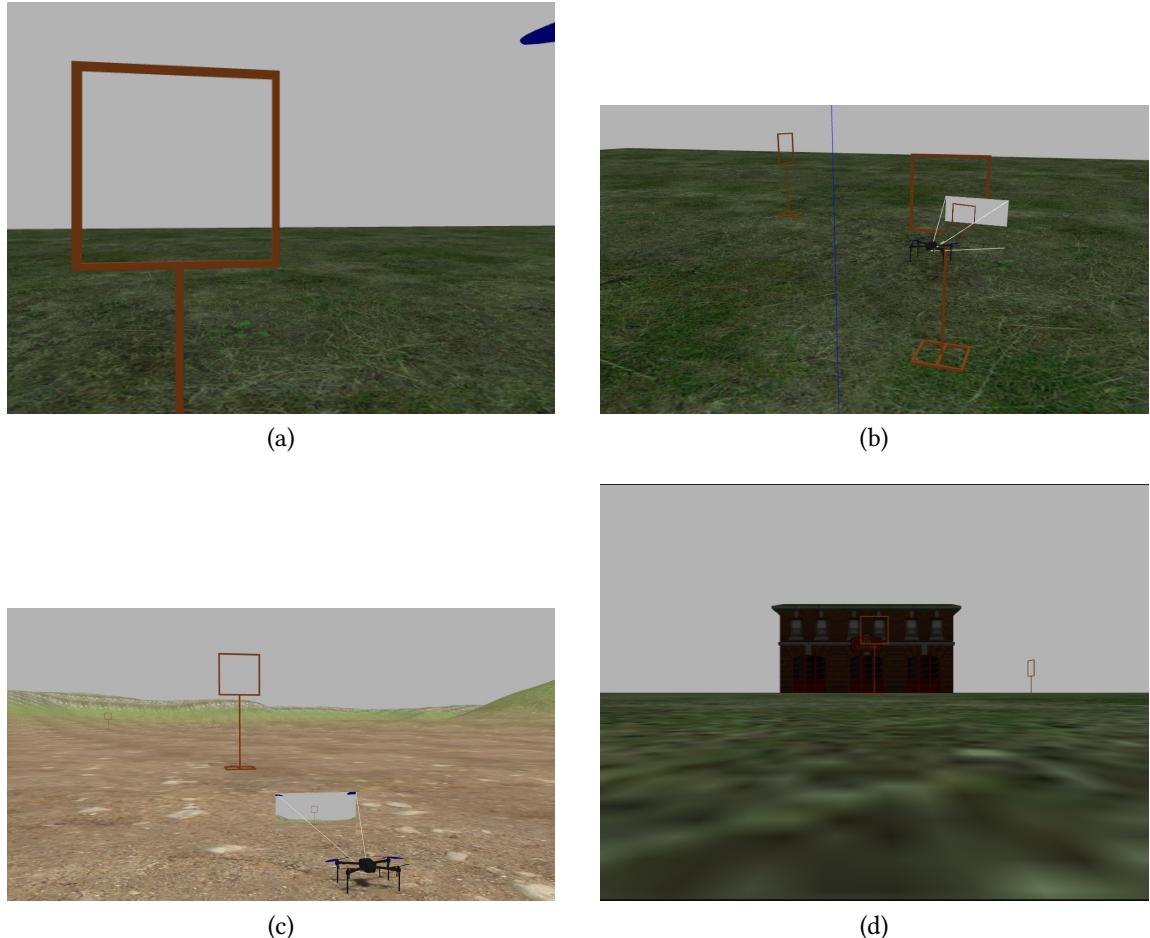


Figura 4.19: Imágenes utilizadas para la sintonización del rango de color

tabla 4.2 muestra la relación que existe entre cada etapa y su script en el repositorio de GitHub.

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 la inclusión de la librería Pymavlink y la creación

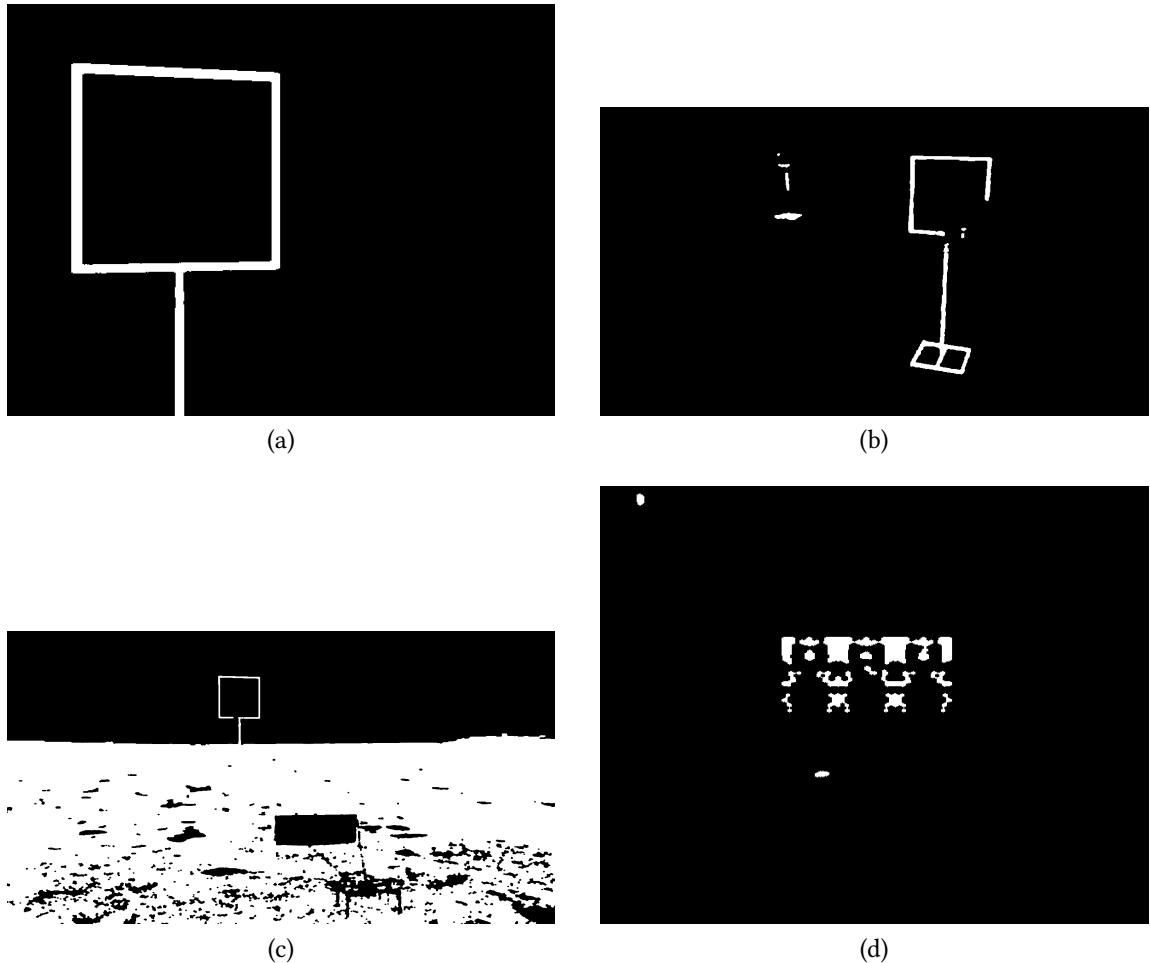


Figura 4.20: Detección de compuerta con el primer rango de color

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 4.23 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 detenida es posible percibirse que hay dos direcciones detonadas por la palabra *.out*, la

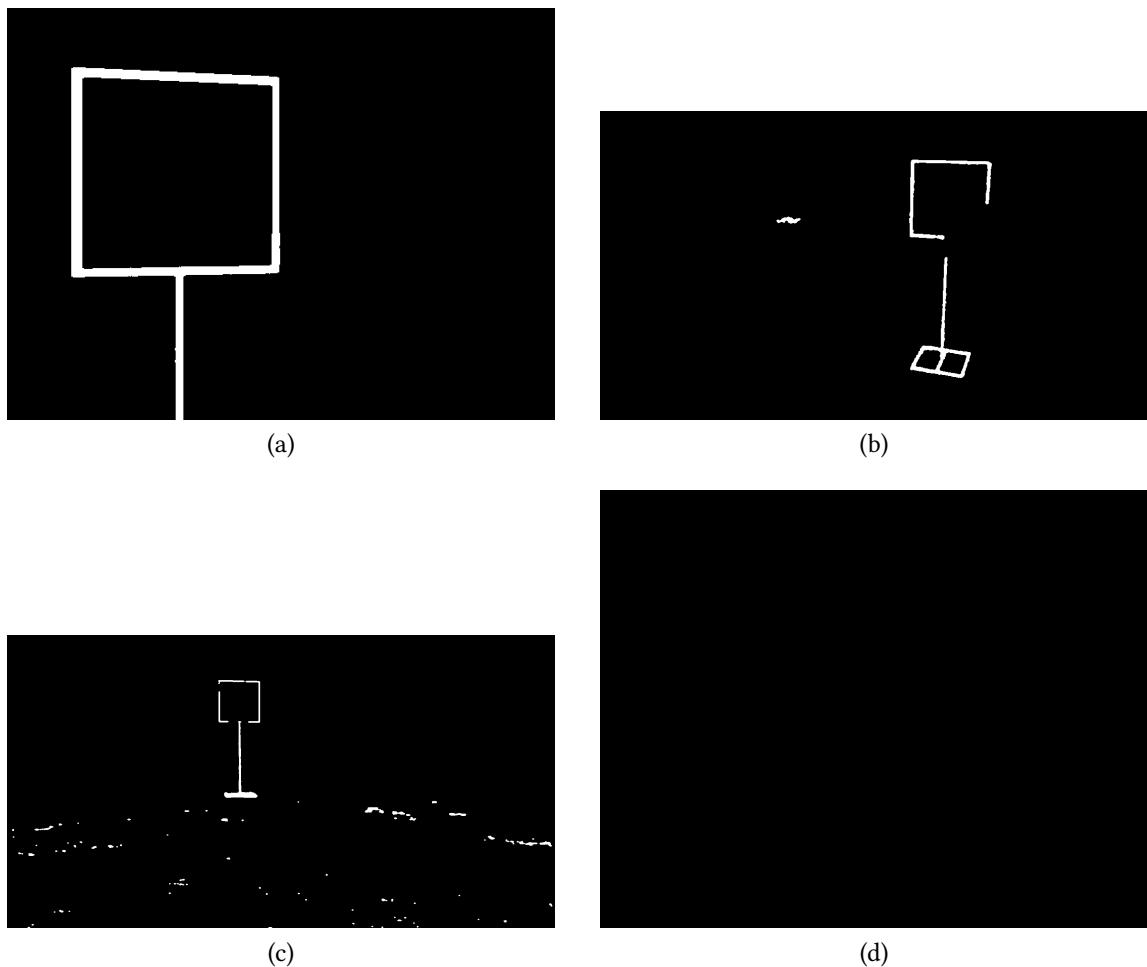


Figura 4.21: Resintonización del rango de color

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.

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 esto se creó un objeto de Python para establecer la conexión, es posible acceder a un dato 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 de dron con el fin de comprobar que la comunicación se dio de forma exitosa.



Figura 4.22: Intento de sintonización utilizando la cuarta imagen de referencia como base

	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

Por último con respecto a esta subsección, la figura 4.24 muestra el resultado de haber entablado una comunicación exitosa entre Pymavlink y el piloto automático, en la figura 4.24a 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 4.24b presenta esta trama de forma filtrada, de tal manera que solo se imprime el ángulo de roll del dron.

4.4.2. Configuración de modo de vuelo

De acuerdo con la documentación oficial de ArduPilot [6], 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

Capítulo 4. Resultados

```
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" "-I0"  
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"  
RTTW: 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 -I0  
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 4.23: Datos de conexión de ArduPilot

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 código perteneciente de 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 4.25 muestra el resultado que se obtuvo al ejecutar el código desarrollado para esta subsección, en la figura 4.25a se observa el mensaje de ejecución del script como tal, y por otro lado, la figura 4.25b presenta la aceptación del comando y

Capítulo 4. Resultados

```

File Edit View Terminal Tabs Help
ATTITUDE {time_boot_ms : 265732, roll : 0.001770165393813777, pitch : 0.0017731741535684, yaw : 0.01775225514679227, rollspeed : 0.0002960892676598, pitchspeed : 0.000275457200997154, yawspeed : 0.001189235350182486}
ATTITUDE {time_boot_ms : 266023, roll : 0.0017786281295120716, pitch : 0.0017730752163646, yaw : 0.0177307453737686, rollspeed : 0.0002690583837518573, yawspeed : 0.001226210950612868}
ATTITUDE {time_boot_ms : 267075, roll : 0.00182844682129961082, pitch : 0.001817225229397146, yaw : 0.01776266145467758, rollspeed : 0.0003738130701316476, pitchspeed : 0.0002095154486595843, yawspeed : 0.001173458527738187}
ATTITUDE {time_boot_ms : 267325, roll : 0.001820808919254449964, pitch : 0.001746699998070465, yaw : 0.01767144352197647, rollspeed : 0.0003178101554132998, pitchspeed : 0.000295556501540038, yawspeed : 0.0011867942118644714}
ATTITUDE {time_boot_ms : 267447, roll : 0.001820808919254449964, pitch : 0.001746699998070465, yaw : 0.01767144352197647, rollspeed : 0.0003178101554132998, pitchspeed : 0.000295556501540038, yawspeed : 0.0011867942118644714}
ATTITUDE {time_boot_ms : 267823, roll : 0.001289963537832471, pitch : 0.000135278208619, yaw : 0.01762440800666809, rollspeed : 0.000308102854961443, pitchspeed : 0.00023027609609506567, yawspeed : 0.0011555940789923}
ATTITUDE {time_boot_ms : 268073, roll : 0.0012868151534348726, pitch : 0.001087028874597097, yaw : 0.017610162622210995, rollspeed : 0.0003122246125712991, yawspeed : 0.001186649486825169}
ATTITUDE {time_boot_ms : 268073, roll : 0.0012868151534348726, pitch : 0.001087028874597097, yaw : 0.017610162622210995, rollspeed : 0.0003122246125712991, yawspeed : 0.001186649486825169}
ATTITUDE {time_boot_ms : 268252, roll : 0.0012868151534348726, pitch : 0.001087028874597097, yaw : 0.017610162622210995, rollspeed : 0.0003122246125712991, yawspeed : 0.001186649486825169}
ATTITUDE {time_boot_ms : 268252, roll : 0.0012868151534348726, pitch : 0.0010857674063894, yaw : 0.01753657493765683, rollspeed : 0.000262185570016581, pitchspeed : 0.0003656083913968457, yawspeed : 0.0011836469639062684}
ATTITUDE {time_boot_ms : 269075, roll : 0.00130000706899502664, pitch : 0.0010869584443435973, yaw : 0.017512233687400818, rollspeed : 0.00034063847187799215, pitchspeed : 0.00024063853769252033, yawspeed : 0.0011292471317574802}
ATTITUDE {time_boot_ms : 269075, roll : 0.00130000706899502664, pitch : 0.0010869584443435973, yaw : 0.017512233687400818, rollspeed : 0.00034063853769252033, pitchspeed : 0.00024063853769252033, yawspeed : 0.0011292471317574802}
ATTITUDE {time_boot_ms : 269573, roll : 0.001301456498054727, pitch : 0.00108935544267968, yaw : 0.017480126490772166, rollspeed : 0.00037809336780807294, pitchspeed : 0.000287417984738412, yawspeed : 0.0011307694949209691}
ATTITUDE {time_boot_ms : 269823, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 270073, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 270223, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 270575, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 270823, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 271273, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 271323, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 271573, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 271823, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}
ATTITUDE {time_boot_ms : 272073, roll : 0.0013018061939367092, pitch : 0.0010905694081053326, yaw : 0.01744552766051995, rollspeed : 0.000364325707783346, pitchspeed : 0.000287417984738412, yawspeed : 0.001129461350019347}

```

(a) Trama de datos de la actitud del dron

```

File Edit View Terminal Tabs Help
mavproxy@e5:~/pymavlink$ python3 listen.py
Heartbeat from system (system 1 component 0)
0.000348931411212807
0.00032771643686426614
0.00039273897351477444
0.00026154887780676987
0.0003030644608203493
0.000243469728775496513
0.00021968834973745793
0.0001981121313293532
0.00018420109669945807
0.00016397135914363362
0.0001589677780573948
0.0001232232317206548
0.00010289751177831424
0.670622628415903e-05
0.916467827977613e-05
5.1426794731199e-05
2.7764397609549273e-05
1.843062812950857e-05
-3.332352663238e-06
-1.665580111546442e-05
-3.477477002888918e-05
-4.752289861180745e-05

```

(b) Datos del roll

Figura 4.24: Datos obtenidos a partir de la conexión entre ArduPilot y Pymavlink

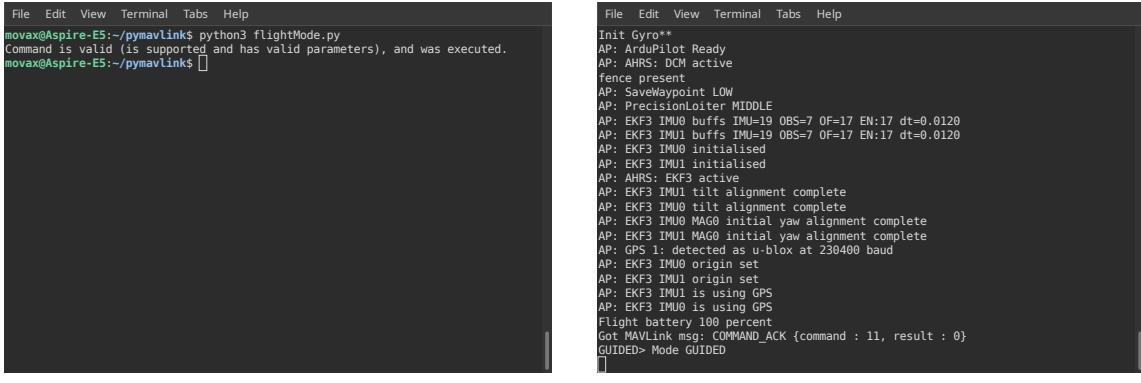
el cambio de modo de vuelo reflejado en la terminal de ArduPilot.

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 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

Capítulo 4. Resultados



```
File Edit View Terminal Tabs Help
movax@Aspire-E5:~/pymavlink$ python3 flightMode.py
Command is valid (is supported and has valid parameters), and was executed.
movax@Aspire-E5:~/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 bufffs IMU=19 OBS=7 OF=17 EN:17 dt=0.0120
AP: EKF3 IMU1 bufffs IMU=19 OBS=7 OF=17 EN:17 dt=0.0120
AP: EKF3 IMU0 initialised
AP: EKF3 IMU1 initialised
AP: AHRS: EKF3 active
AP: EKF3 IMU1 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
```

(a) Respuesta en Pymavlink

(b) Respuesta en ArduPilot

Figura 4.25: Ejecución del script para la configuración del modo de vuelo

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.

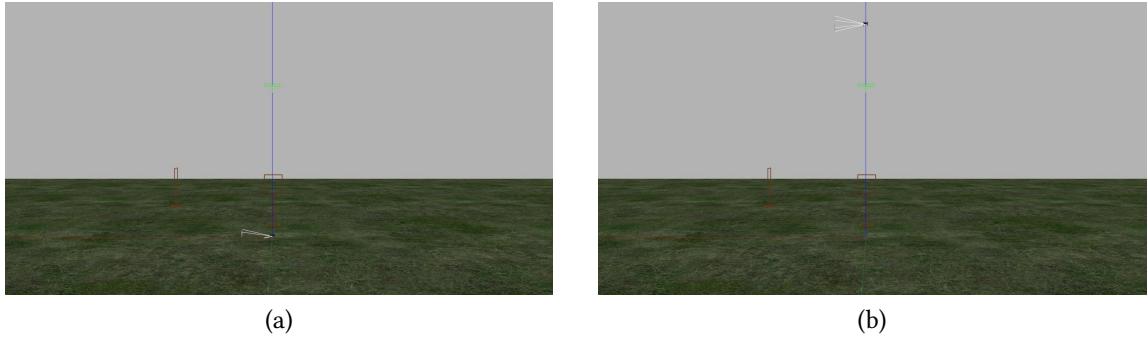


Figura 4.26: Comportamiento de la simulación ante el comando de despegue



Figura 4.27: Respuesta en la altura para el comando de despegue.

4.4.4. Seguimiento de trayectoria

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

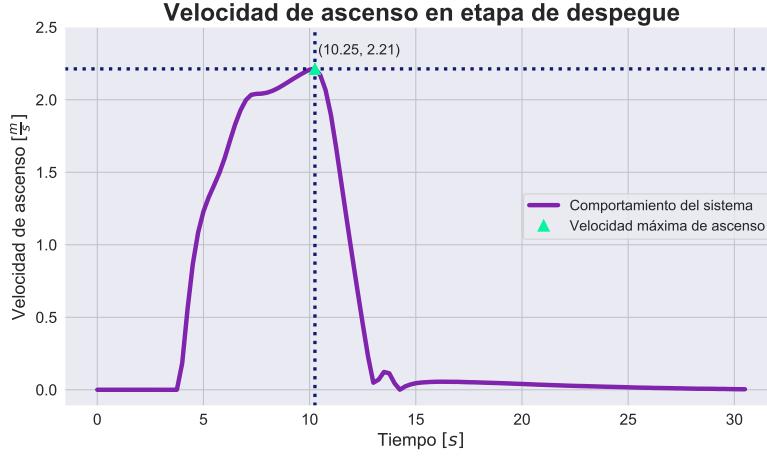


Figura 4.28: Gráfica de velocidad de despegue.

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 . Por otro lado, a pesar de que se mencionó que solo se envió un waypoint para establecer la trayectoria de vuelo, el script utilizado en realidad contiene 2 waypoints, [10,0,-3] y [30 0 -3]; Dicho esto, 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 primer waypoint. La solución para el problema anterior se describe en la siguiente subsección.

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

Tabla 4.3: Parámetros de la instrucción para seguimiento de waypoints

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

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.

En sí, esta etapa es un compilado de las etapas anteriores, de tal forma que la integración sigue el orden de las etapas conforme fueron explicadas en las subsecciones anteriores. Adicionalmente, esta etapa representa la culminación del seguimiento de trayectoria, pues se buscó eliminar la necesidad de que el usuario ingresara algún tipo de comando de forma manual en la terminal de ArduPilot, y que el dron fuera capaz de seguir una serie de waypoints, por lo que también se dio solución a algunos de los errores especificados en las etapas anteriores, entre ellos el hecho de que sin la existencia de alguna bandera que indicara la llegada a cierto waypoint, el script de seguimiento de trayectoria ejecuta el último waypoints enviado, ignorando los posibles waypoints que se hubieran enviado antes.

Lo anterior se solucionó realizando la lectura del mensaje de estado generado por el controlador de vuelo, existe un parámetro que indica la distancia faltante para llegar al último waypoint enviado. Realizada esta lectura, el siguiente waypoint es enviado hasta que la distancia leída es cero.

La solución que se acaba de mencionar, permitió establecer un arreglo con los waypoints necesarios para que el dron completara el circuito de vuelo, de forma sistemática y cíclica.

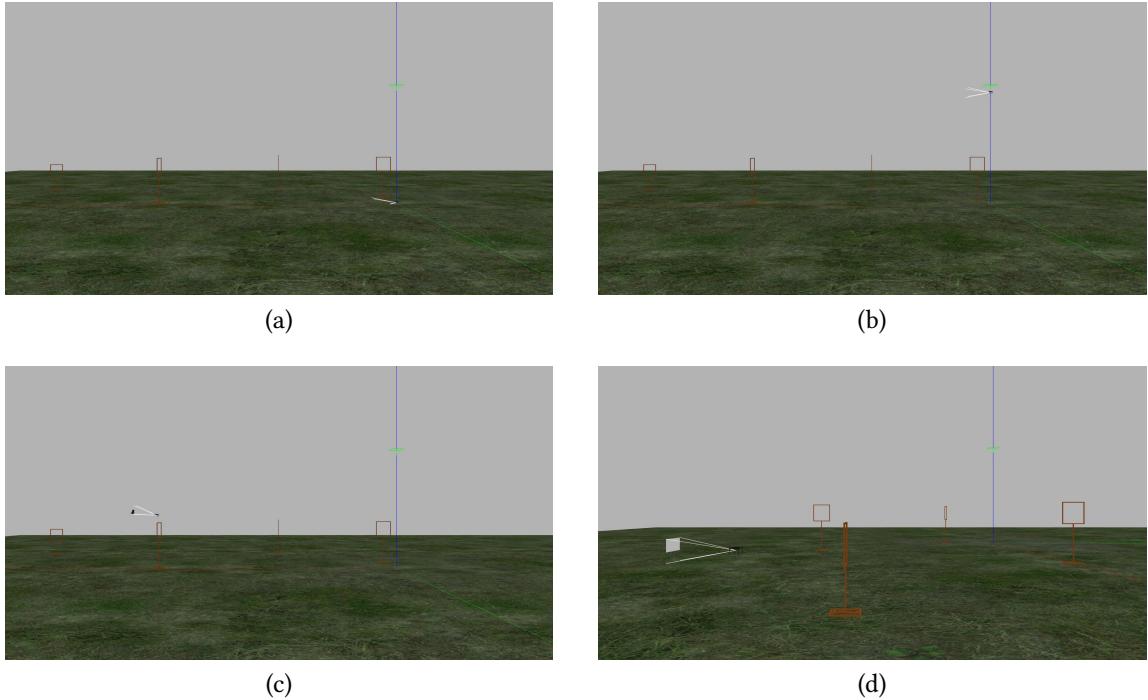


Figura 4.29: Etapas de la prueba de seguimiento de trayectoria

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 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



Figura 4.30: Recorrido realizado por el dron

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

Capítulo 4. Resultados

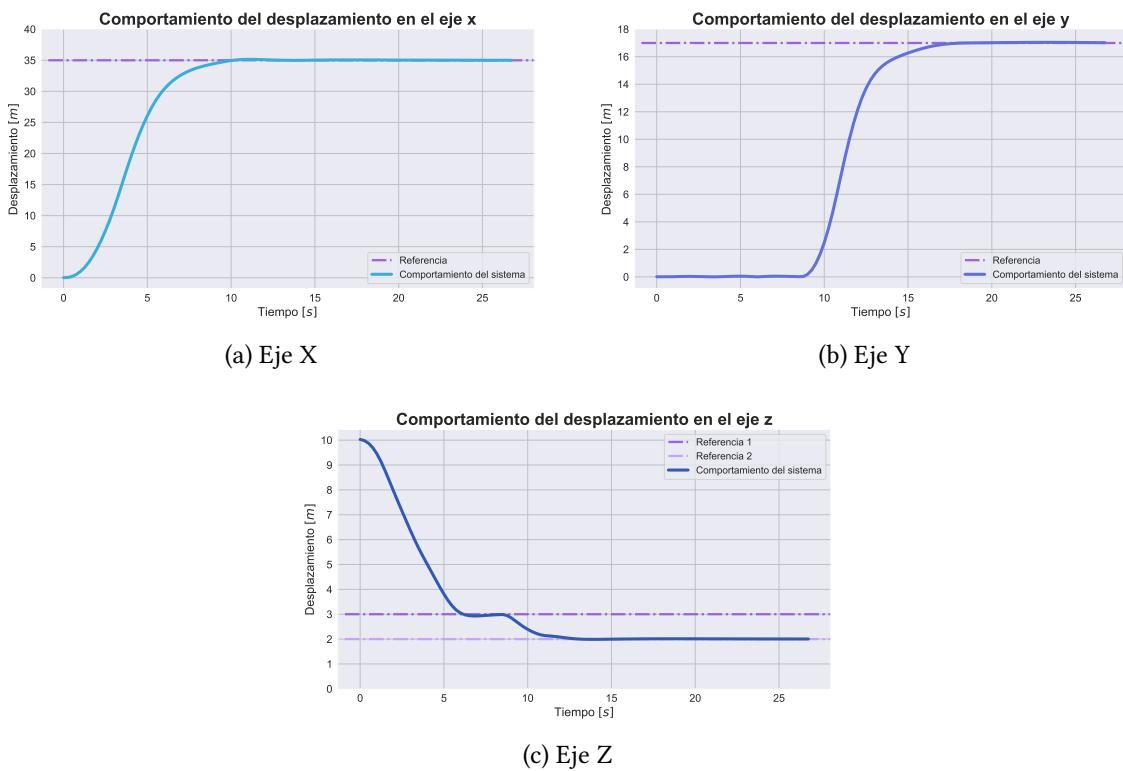


Figura 4.31: Gráficas de desplazamiento para los tres ejes

Capítulo 4. Resultados

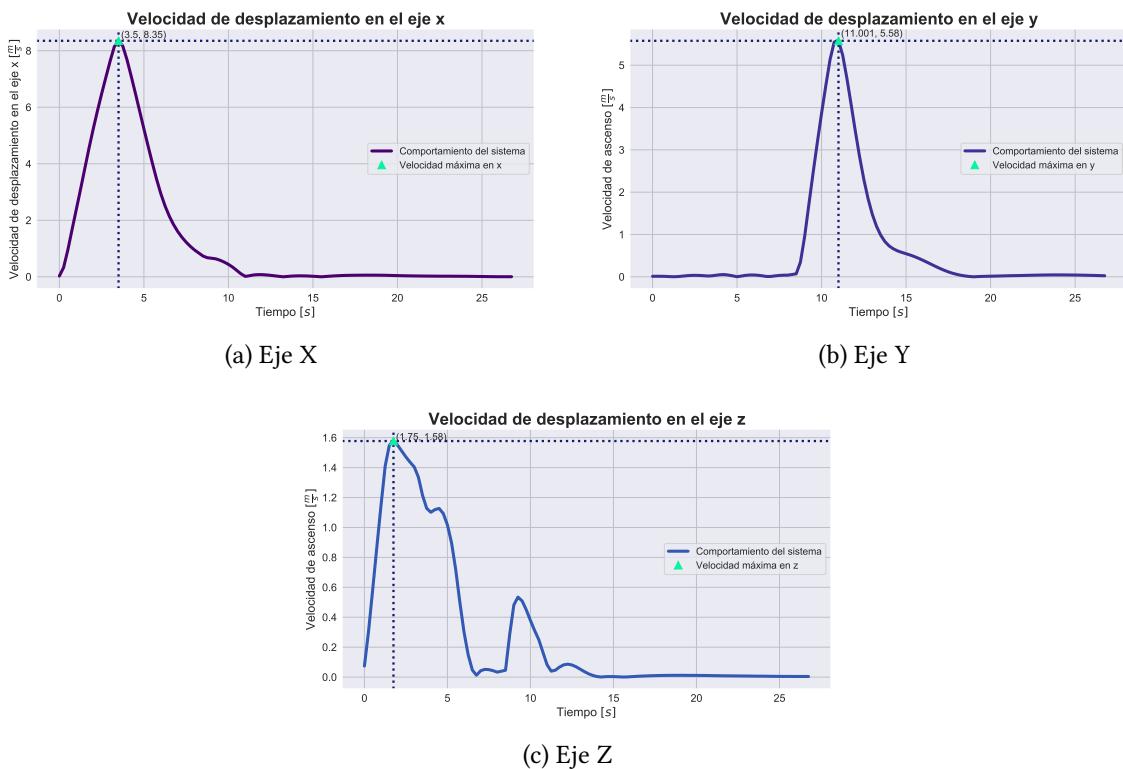


Figura 4.32: Comportamiento de la velocidad lineal de vuelo en los tres ejes

Capítulo 4. Resultados

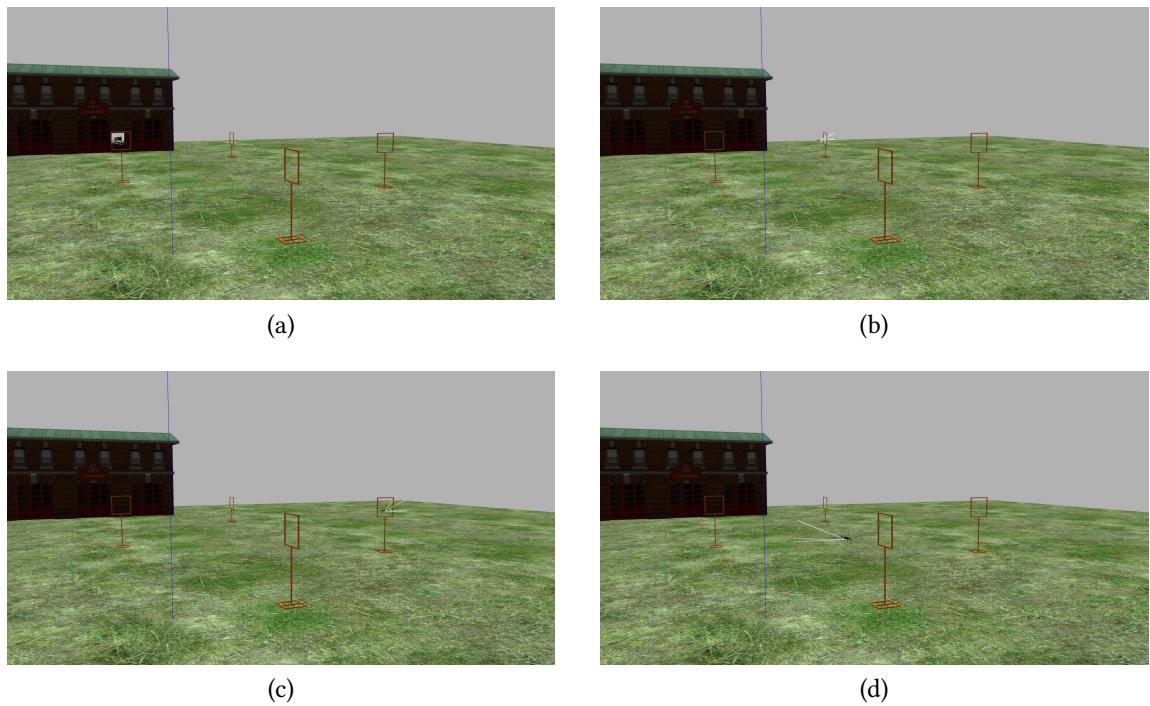


Figura 4.33: Seguimiento de misión de vuelo completa



Figura 4.34: Recorrido realizado por el dron

Capítulo 4. Resultados

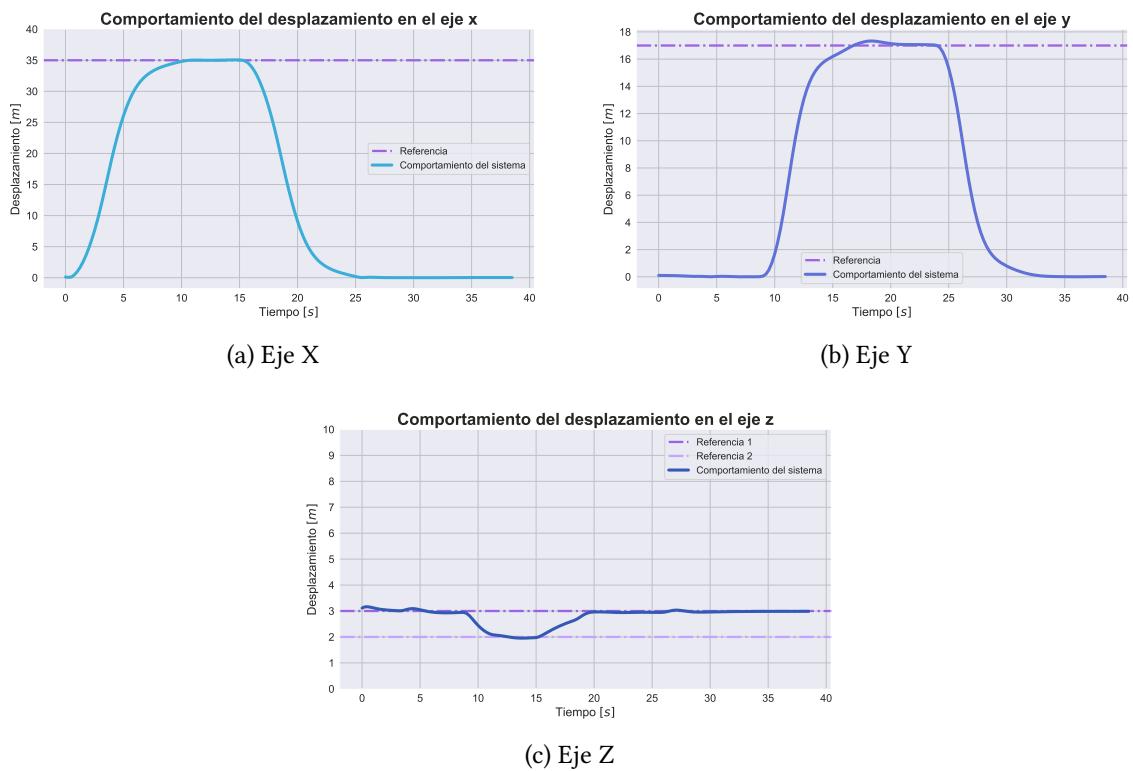


Figura 4.35: Gráficas de desplazamiento para los tres ejes

Capítulo 4. Resultados

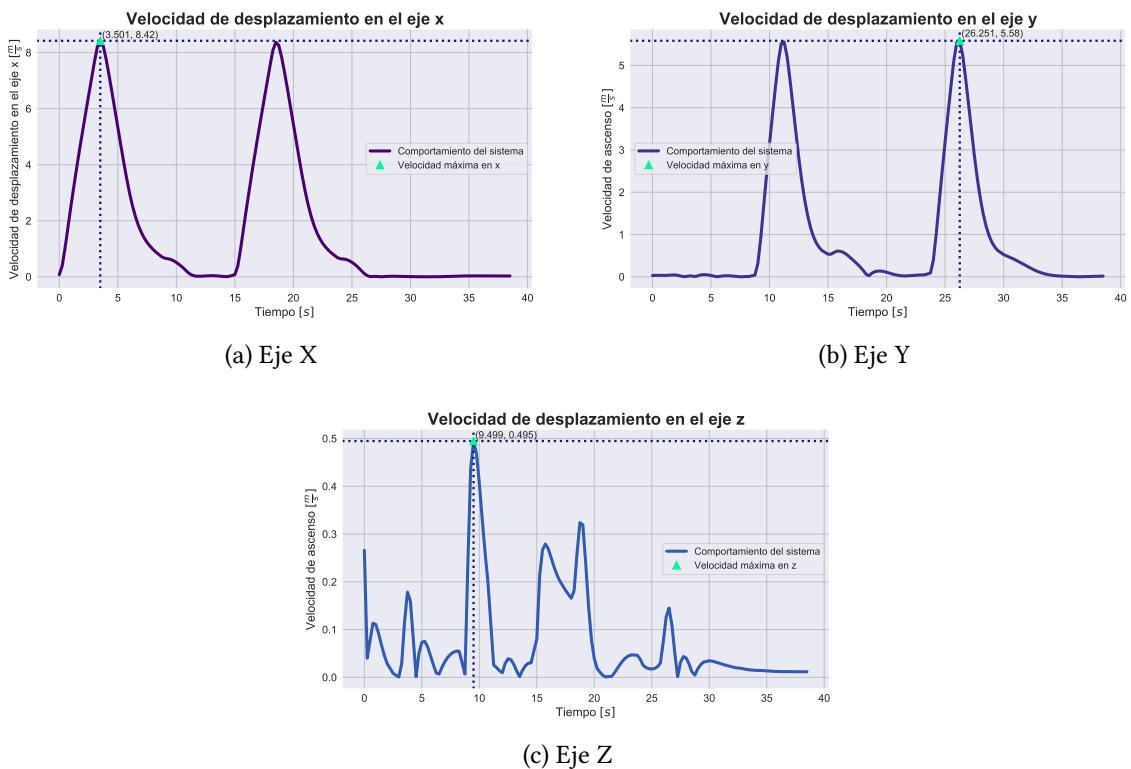


Figura 4.36: Comportamiento de la velocidad lineal de vuelo en los tres ejes

BIBLIOGRAFÍA

- [1] Cabrera-Ponce, A. A., Rojas-Perez, L. O., Carrasco-Ochoa, J. A., Martinez-Trinidad, J. F., and Martinez-Carranza, J. (2019). Gate detection for micro aerial vehicles using a single shot detector. *IEEE Latin America Transactions*, 17(12):2045–2052.
- [Ermakov] Ermakov, V. mavros. <http://wiki.ros.org/mavros>. Accessed: 2022-01-04.
- [3] Ferrari, P. (2018). Ssd: Single-shot multibox detector implementation in keras. https://github.com/pierluigiferrari/ssd_keras.
- [4] Foehn, P., Brescianini, D., Kaufmann, E., Cieslewski, T., Gehrig, M., Muglikar, M., and Scaramuzza, D. (2020). Alphapilot: Autonomous drone racing. *arXiv preprint arXiv:2005.12813*.
- [5] Foehn, P., Romero, A., and Scaramuzza, D. (2021). Time-optimal planning for quadrotor waypoint flight. *Science Robotics*, 6(56).
- [6] Johnson, E. (2022). Flight modes. <https://ardupilot.org/copter/docs/flight-modes.html>.
- [7] Kaufmann, E., Loquercio, A., Ranftl, R., Dosovitskiy, A., Koltun, V., and Scaramuzza, D. (2018). Deep drone racing: Learning agile flight in dynamic environments. In *Conference on Robot Learning*, pages 133–145. PMLR.

BIBLIOGRAFÍA

- [8] Loquercio, A., Maqueda, A. I., Del-Blanco, C. R., and Scaramuzza, D. (2018). Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095.
- [9] Lorenz Meier, Andreas Antener, t. (2021). Mavlink developer guide. <https://mavlink.io/en/>. Accessed: 2022-01-04.
- [10] Madaan, R., Gyde, N., Vemprala, S., Brown, M., Nagami, K., Taubner, T., Cristofalo, E., Scaramuzza, D., Schwager, M., and Kapoor, A. (2020). Airsim drone racing lab. In *NeurIPS 2019 Competition and Demonstration Track*, pages 177–191. PMLR.
- [11] Mellinger, D. and Kumar, V. (2011). Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE international conference on robotics and automation*, pages 2520–2525. IEEE.
- [12] 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.
- [13] Moon, H., Sun, Y., Baltes, J., and Kim, S. J. (2017). The iros 2016 competitions [competitions]. *IEEE Robotics and Automation Magazine*, 24(1):20–29.
- [14] Mueller, M. W., Hehn, M., and DÁndrea, 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*, pages 3480–3486. IEEE.
- [15] Open-Robotics (2014). <http://gazebosim.org/>.
- [16] Open-Robotics (2021). Installing ros 2 on ubuntu linux. <https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Binary.html>. Accessed: 2022-01-12.
- [17] OpenRobotics (2021). Ros 2 documentation: foxy documentation. <https://docs.ros.org/en/foxy/index.html>.
- [18] Robotics, O. (2021a). Building ros 2 on ubuntu linux – ros 2 documentation: foxy documentation. <https://docs.ros.org/en/foxy/Installation/Ubuntu-Development-Setup.html>.

BIBLIOGRAFÍA

- [19] Robotics, O. (2021b). Open robotics. <https://www.openrobotics.org/>.
- [20] Rojas-Perez, L. O. and Martinez-Carranza, J. (2020). Deepilot: A cnn for autonomous drone racing. *Sensors*, 20(16):4524.
- [21] Rojas-Perez, L. O. and Martinez-Carranza, J. (2021). On-board processing for autonomous drone racing: An overview. *Integration*.
- [22] Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2017). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*.
- [23] Stevens, J.-L. (2021). Autonomous visual navigation of a quadrotor vtol in complex and dense environments. *Jean-Luc Stevens*.
- [24] Xia, X., Xu, C., and Nan, B. (2017). Inception-v3 for flower classification. In *2017 2nd International Conference on Image, Vision and Computing (ICIVC)*, pages 783–787. IEEE.