

ENTORNO DE SIMULACIÓN PARA EL ENTRENAMIENTO MEDIANTE REINFORCEMENT LEARNING DEL VUELO AUTÓNOMO DE UN CUADRICÓPTERO

Autor: Daniel Jáuregui Cortizo

Tutor: Felipe Gil Castiñeira

Curso: 2018/19

Índice

1. Introducción	2
2. Objetivos	2
3. Resultados	3
3.1. Visión general	3
3.2. Desarrollo del proyecto	5
3.3. Arquitectura del sistema final	11
4. Conclusiones	13
4.1. Usabilidad	14
4.2. Líneas futuras	14
Anexo A. Estado del arte	15
Anexo B. Tecnologías utilizadas	17
B.1. Controlador de vuelo: PX4	18
B.2. Framework de programación: ROS	20
B.3. Entorno de simulación: AirSim	21
B.4. Reinforcement Learning: Gym	24

1. Introducción

Una de las tecnologías que más destacan y llaman la atención en estos últimos años son los llamados "drones", definidos formalmente como *aeronaves no tripuladas* por la RAE. Estos vehículos se están utilizando en casi cualquier ámbito, desde usos militares hasta juguetes de ocio, no sin pasar por usos comerciales como el transporte o relacionados con seguridad y defensa.

Su bajo coste, reducido tamaño y la posibilidad de manejarlos remotamente les da un potencial inmenso para una infinidad de usos, entre otras cosas, en tareas que debería hacer una persona pudiendo tener que asumir riesgos o peligros, como puede ser la vigilancia desde el aire en zonas de conflicto, la búsqueda de supervivientes tras una catástrofe natural o la actuación en incendios forestales. Sin embargo, para algunas tareas los recursos humanos son limitados y no es viable ocupar a un operario pilotando un dron, apareciendo así la necesidad de diseñar aeronaves que puedan llevar a cabo su cometido de forma autónoma.

Volar de forma autónoma es una tarea muy compleja en la que pueden suceder innumerables situaciones diferentes e imprevisibles, a las que se debe dar respuesta de la forma más rápida y acertada posible. Es necesario dotar a los drones de inteligencia para que puedan desenvolverse por sí mismos y aprender de su propia experiencia. Esta necesidad se puede satisfacer con técnicas de *Reinforcement Learning* (RL o Aprendizaje por Refuerzo), un área del *Machine Learning* que agrupa técnicas de aprendizaje automático basadas en la experimentación y la experiencia de la propia máquina. Dichas técnicas comparten una etapa de entrenamiento basada en sucesivos episodios de prueba y error en los que la máquina debe procurar llegar a un objetivo. Durante estos episodios el aparato aprende de sus propias interacciones con el entorno que lo rodea. Precisamente por esa aproximación basada en la realización de pruebas y en el aprendizaje a partir de los errores, no es económicamente viable realizar el entrenamiento en un escenario real, ya que los daños que sufriría el dron solo permitirían completar unas pocas fases de entrenamiento. Por este motivo, y también con la pretensión de experimentar con estas recientes tecnologías y la ilusión de llegar a proporcionar una funcionalidad útil a los posteriores desarrolladores en este campo, en este trabajo fin de grado se construirá un *framework* de simulación, desarrollo y entrenamiento de sistemas RL orientado a la experimentación en el vuelo autónomo de drones.

2. Objetivos

El objetivo final de este trabajo es la obtención de un ***framework* de simulación, desarrollo y entrenamiento de sistemas inteligentes basados en Reinforcement Learning orientado a la experimentación en el vuelo autónomo de drones**. Es decir, se trata de crear y reunir un conjunto de herramientas que faciliten la experimentación, el desarrollo y el posterior entrenamiento en simulación de sistemas inteligentes (concretamente basados en técnicas de *Reinforcement Learning*) para drones.

Para crear esta arquitectura se partirá de una serie de tecnologías ya existentes como son: el entorno de simulación *Microsoft AirSim* [B.3], el controlador de vuelo *PX4 autopilot* [B.1], el *framework* robótico *ROS* [B.2], y el *toolkit* para desarrollo de sistemas basados en *Reinforcement Learning*, *Gym* [B.4]. Todas estas tecnologías se recogen en el apartado B de los anexos, donde son introducidas y brevemente explicadas, así como también se señalan las razones por las que se han escogido para este proyecto.

Durante la implementación de este *framework* deberán integrarse las 4 tecnologías, solucionando las diferentes incompatibilidades y las problemáticas integraciones que guardan algunas de ellas.

Es importante destacar que este trabajo pretende seguir una filosofía *plug&play*, traduciéndose en una migración directa (o casi directa) de las implementaciones creadas en el *framework* de simulación hacia el dron real. Esto se puede conseguir gracias a la integración de un controlador de vuelo real disponible en drones físicos.

3. Resultados

En el presente apartado se indicará el proceso seguido durante el desarrollo del proyecto, así como los resultados obtenidos a lo largo del mismo. Se comenzará dando una visión general del sistema que se ha creado para facilitar su comprensión. Posteriormente se explicarán los pasos seguidos en el desarrollo del trabajo: fases de la implementación, pruebas realizadas, problemas encontrados y soluciones tomadas. Para concluir, se mostrará la arquitectura del sistema final con el objetivo de comprender de forma global el funcionamiento del mismo.

3.1. Visión general

Con la finalidad de introducir y entender de forma general la constitución del *framework* creado en este proyecto, y para facilitar la posterior comprensión de las fases o etapas de trabajo que fueron necesarias en su desarrollo, se dará una breve visión general del sistema creado, distribuido por bloques.

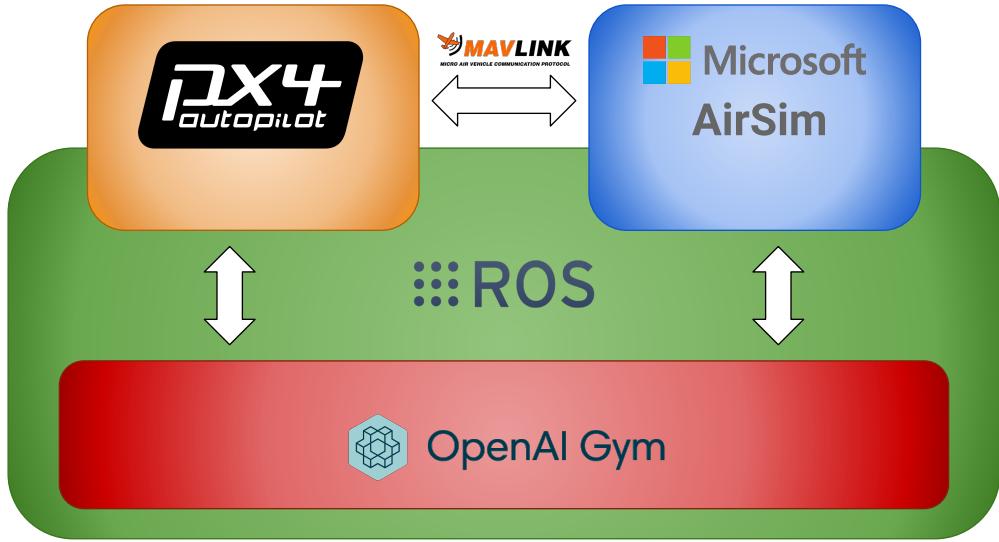


Figura 1: Arquitectura del sistema por bloques.

Tal y como se puede apreciar en la figura 1, el sistema se compone de 4 bloques. En la parte superior se encuentran los bloques de AirSim y PX4. Se trata de dos bloques indispensables e independientes del resto del sistema, ya que solo dependen el uno del otro para poder funcionar. Su comunicación se realiza a través de **MAVLink** [1], un protocolo estándar de comunicación para drones.

El **bloque de AirSim** [2] constituye el entorno de simulación, y en su interior conviven dos tecnologías: el motor Unreal Engine¹, y el propio plugin de AirSim. La utilidad de este bloque es aportar la simulación del vehículo y del entorno, e incluye tanto la simulación de los sensores y actuadores del cuadricóptero, como la generación de las imágenes que produce la cámara integrada en el dron.

El **bloque de PX4** [3] integra el controlador de vuelo. Este se encarga de controlar el comportamiento del vehículo en la simulación, comunicándose con el bloque de AirSim para leer los datos de los sensores y enviarle señales de control a los actuadores. PX4 es un controlador de vuelo diseñado para controlar drones reales, por tanto aunque esté funcionando en un entorno de simulación, su operativa es la misma que en un entorno real.

Como se puede apreciar, los dos bloques mencionados se intersecan con un tercero: el **bloque del sistema operativo robótico, ROS** [4, 5]. La razón es que dichos bloques utilizan información a la que el resto del sistema necesita acceder, como las imágenes de la cámara de AirSim o la información de posición y estado del vehículo de PX4. Por ello, en ambos bloques se han integrado nodos ROS que transmiten esta y otra información al resto del

¹Documentación oficial Unreal Engine 4: <https://docs.unrealengine.com/en-us/>.

sistema a través del **canal de comunicación de ROS**. ROS es un conjunto de librerías y herramientas que han sido diseñadas con este objetivo. De nuevo, se trata de una herramienta diseñada para ser implementada en sistemas reales, por lo que también está presente en el dron cuando está volando en un entorno real.

Por último, el **bloque de Gym Training** es un bloque de *software* completamente desarrollado con ROS y la librería Gym [6] de Python. Consiste en un conjunto de nodos ROS que representan el cerebro del sistema, ya que su función es decidir las acciones que debe llevar a cabo el vehículo, aprender de la experiencia acumulada, y gestionar el entrenamiento. Para todo ello, este bloque se debe comunicar con el bloque de PX4 tanto para leer el estado del vehículo como para enviarle el comportamiento deseado en cada momento; y con el bloque de AirSim tanto para recibir las imágenes de la cámara del dron, como para gestionar el reinicio de la simulación. Estas comunicaciones se realizan a través del canal de comunicación de ROS.

Como podrá intuirse, el ya mencionado **bloque de ROS** no es un bloque cerrado e independiente de los demás, sino más bien al contrario. La estructura distribuida de ROS en nodos de computación y la posibilidad de comunicar los diferentes nodos entre sí, además de la facilidad de integración de estos nodos con las demás tecnologías lo convierten en el elemento de unión perfecto. Así, ROS constituye parte del esqueleto del sistema, proporcionando una interfaz de comunicación en la que todos los bloques pueden acceder a la información que, en otro caso, no podrían intercambiar.

3.2. Desarrollo del proyecto

Una vez presentada la estructura del proyecto, se explicará el procedimiento llevado a cabo durante los últimos meses para llegar a cumplir el objetivo inicial. Este trabajo se ha dividido principalmente en 4 fases que se corresponden con la integración de cada uno de los bloques que forman la estructura del *framework* final.

Previo a las fases de desarrollo, tuvo lugar una etapa de documentación, análisis del estado del arte y análisis bibliográfico, que concluyó en la elección de las tecnologías concretas a emplear para desarrollar el proyecto (mencionadas en los objetivos y en los anexos). Una vez terminado este proceso, y en paralelo a la síntesis del estado del arte recogida en este documento (anexo A), se comenzó la experimentación y familiarización con las tecnologías en cuestión.

La primera fase empezó con la instalación del entorno de simulación, que comprendía el montaje del Unreal Engine 4 y el Microsoft AirSim. Como primera experimentación, se realizó esta instalación en Windows 10, y se consiguió poner a funcionar una versión estable del simulador. Durante esta fase no surgieron complicaciones relevantes.

La siguiente fase consistía en integrar el *firmware* del controlador de vuelo con el entorno de simulación. Dado que PX4 solo admitía su instalación en modo SITL (*Software In The*

Loop)² sobre sistemas Linux³, se decidió montarlo sobre un subsistema de Linux en Windows⁴. La solución tuvo éxito, y a pesar de que surgieron diversas complicaciones durante la instalación de los requisitos⁵, debido a que se trataba de tecnologías desconocidas y a la poca familiarización con el terminal de Linux, se consiguió hacer funcionar el *firmware* del PX4 y hacer volar el cuadricóptero de forma manual a través de la consola de comandos del *autopilot* para comprobar el funcionamiento correcto del sistema.

A continuación, llegó el turno de la tercera fase: integrar ROS. Sin embargo, esta fase se dividió en dos etapas consecutivas: en la primera se instalaría ROS y se publicaría la información del *autopilot* en el canal de comunicación del sistema operativo robótico; y en la segunda, una vez funcionase lo primero, se profundizaría más llegando a crear un nodo ROS que publicase las imágenes tomadas por las cámaras de AirSim.

Pero al emprender la primera etapa, surgió una complicación derivada de una mala planificación y análisis de las herramientas: ROS solo ofrecía compatibilidad con sistemas Linux (en concreto, Ubuntu o Debian). Para solucionar este inconveniente, se acordó instalar de nuevo PX4 junto a ROS en una máquina virtual con Ubuntu 16.04 (se escogió esta versión concreta porque garantizaba el funcionamiento de ambas tecnologías, según sus desarrolladores). La solución fue válida, y tras la instalación, se pudo lanzar PX4 junto a un nodo MAVROS (una librería propia de ROS que establece un bridge entre el protocolo MAVLink [1] y el canal de comunicación propio de la plataforma robótica) comprobando que la información del *autopilot* estaba siendo correctamente publicada en ROS y por tanto, la finalización de la primera etapa de esta integración.

Seguidamente, el comienzo de la etapa restante derivó en otra complicación: la integración de un nodo ROS con el entorno de simulación exigía que ambas tecnologías coexistieran en el mismo sistema operativo. Para solventar este percance, se idearon dos alternativas: la primera, e inicialmente más compleja, se basaba en modificar parte del código de AirSim para que, además de enviar las lecturas de los sensores y demás información a través del protocolo MAVLink, enviase también las imágenes capturadas por las cámaras. Posteriormente, esos mensajes con imágenes serían recibidos por un nodo ROS desde Linux, que publicaría dichos mensajes en el canal de comunicación. La segunda alternativa, aparentemente más sencilla, consistía en migrar todo el entorno de simulación (aprovechando la característica multiplataforma) al sistema operativo Ubuntu.

La solución escogida fue la segunda, ya que resultaba más interesante tener todos los elementos del *framework* sobre el mismo sistema operativo, en vez de obligar al futuro desarrollador a tener ambas plataformas, con el excesivo consumo de recursos que ello implica.

²Consultar anexo B.1

³Fuente: https://github.com/Microsoft/AirSim/blob/master/docs/px4_sitl.md.

⁴Guía de instalación del *Subsistema de Linux sobre Windows*: <https://docs.microsoft.com/es-es/windows/wsl/install-win10>.

⁵Guía de instalación de requisitos para PX4: http://dev.px4.io/en/setup/dev_env_linux.html.

Sin embargo, la aparentemente sencilla solución escogida resultó en una complicación mayor, aunque esta vez derivada de incompatibilidades con el *hardware* del equipo utilizado en el desarrollo. La máquina virtual utilizada imposibilitaba la virtualización de la GPU⁶ del equipo, evitando así que Linux tuviese acceso al procesador gráfico y se restringiese al uso de la CPU⁷. Debido a que el Unreal Engine tiene unos requisitos de rendimiento gráfico notablemente altos, resultó imposible ejecutar el entorno de simulación en el equipo virtualizado. La solución definitiva frente a este problema fue instalar una versión nativa del sistema operativo Ubuntu 16.04 directamente sobre el equipo (es decir, sin virtualización). Las consecuencias de esta concatenación de percances se tradujo en un consumo desmesurado de tiempo empleado en la investigación de causas y posibles soluciones, en las sucesivas pruebas para intentar compatibilizar los recursos gráficos con la máquina virtual, y posteriormente, en la repetición de la instalación y prueba de todos los componentes del sistema en el nuevo sistema operativo.

Finalmente, una vez hecha la instalación nativa de Linux, y sobre ella la del entorno de simulación, el *autopilot*, y el sistema operativo robótico, se procedió a continuar con la segunda etapa de la integración de ROS: crear un nodo robótico en el contexto de AirSim que leyese las imágenes de las cámaras y las publicase en el canal de comunicación. Llegados a este punto la implementación de dicho nodo apoyándose en las librerías ofrecidas por el simulador, resultó considerablemente sencilla, terminando así la tercera fase del desarrollo.

Tras esto, el desarrollo del *framework* pasó a su última fase de integración, donde se trataría de integrar algún tipo de plataforma para la experimentación con *Reinforcement Learning*. Durante la investigación acerca de la tecnología Gym, se encontraron varios artículos en los que se integraba el paquete Gym con la tecnología ROS para el entrenamiento de un vehículo mediante RL [7, 8]. Estos documentos se tomaron como referencia para crear el módulo de desarrollo y entrenamiento de algoritmos inteligentes.

Sin mayores complicaciones se desarrollaron un conjunto de archivos de prueba en Python que siguen la estructura de Gym: definen a un **agente** (el dron) con un grupo de posibles acciones, y con capacidad para conocer su estado en cada momento y para recibir las imágenes de la cámara; un **algoritmo** que escoge la acción que debe llevar a cabo el agente y aprende de lo que sucede; y un **entorno** (la simulación) que gestiona el reinicio de los episodios de entrenamiento y la inicialización del agente (figura 2).

⁶GPU: *Graphic Processing Unit*.

⁷CPU: *Central Processing Unit*.

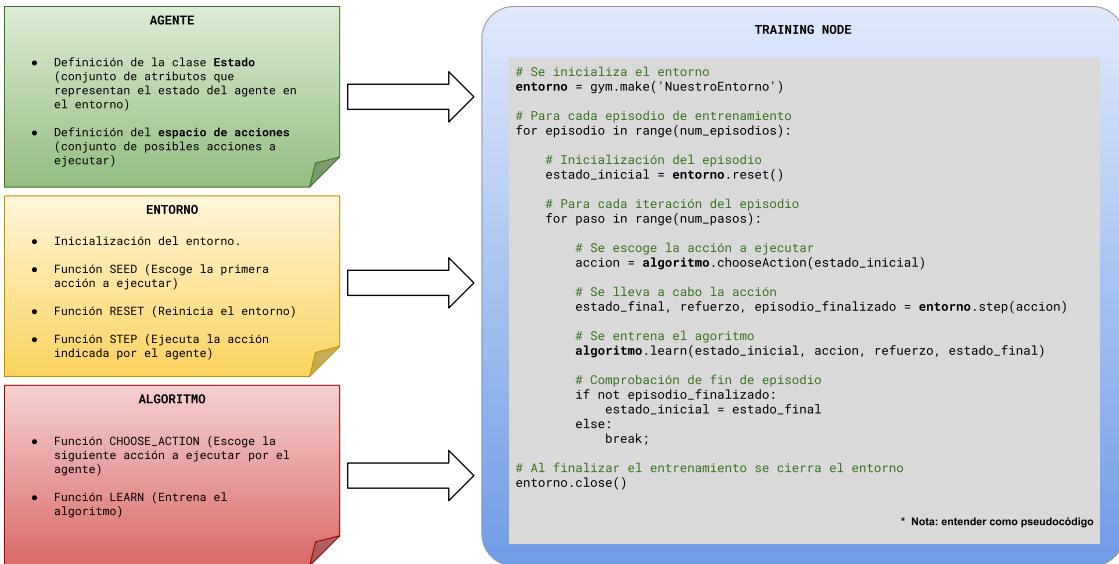


Figura 2: Estructura del módulo de Reinforcement Learning basado en Gym.

Para asegurar funcionamiento del código desarrollado se realizaron una serie de pruebas que comprobaban la correcta ejecución en la simulación de todas las acciones y comportamientos del dron. Pero durante estas pruebas, se detectó un nuevo problema: el reinicio de los episodios no funcionaba. Tal y como se describe en este documento cuando se introduce el campo del Aprendizaje por Refuerzo (tanto en la introducción, como en los anexos), las técnicas de RL se basan en un entrenamiento de “prueba y error”, haciendo necesaria una funcionalidad de “vuelta al estado inicial” o “reset” que se utilice después de cada error. Cuando el episodio de entrenamiento se inicia (o reinicia), lo primero que Gym exige es ejecutar la función *RESET*, que devuelve al dron a su posición inicial, y tras esta el vehículo debe despegar y situarse en su posición base desde la que arrancará el entrenamiento en sí.

Cuando el entrenamiento se inicia se detecta que, tras el *reset*, al dron le cuesta despegar, y cuando lo hace, lo hace sin control ni estabilidad. Después de un primer análisis del problema, y de una investigación en la red, se determina que el fallo proviene del *autopilot*. Esta causa se confirma al analizar el funcionamiento interno del mismo. Lo que ocurre es que el PX4 es un controlador de vuelo desarrollado para vuelo real, y no está optimizado para simulación, por lo tanto, cuando la simulación se reinicia (*reset*) los sensores del dron detectan una discontinuidad en las lecturas de la posición y estado del vehículo (algo análogo a un teletransporte), y dicha discontinuidad produce un desajuste del estimador de posición del *autopilot* que hace perder el control de la posición del cuadricóptero. En este punto, comienza una nueva fase del desarrollo no contemplada al inicio del proyecto, que se basa en resolver la incompatibilidad que existe a la hora de utilizar un controlador de vuelo real en un entorno de simulación.

Tras un tiempo de investigación se encuentra un comentario⁸ de Chris Lovett, uno de los desarrolladores de Microsoft AirSim, con la respuesta al problema que se plantea: la solución está en, de alguna forma, reiniciar el PX4 para que este vuelva a estimar la posición y estado del vehículo.

Un posterior estudio de las alternativas dirige a dos posibles soluciones: la primera de ellas, más conservadora, se reduce a reiniciar el *autopilot* a través de una función *reboot* ya existente destinada para ello; la segunda, en cambio, se basa en modificar el *firmware* del controlador para añadir una funcionalidad de reinicio del estimador.

Inicialmente se opta por la primera alternativa. Se intenta ejecutar esa función de reinicio tanto a través de un mensaje MAVLink, como desde la consola de comandos del propio *autopilot*, sin embargo en ambas pruebas la función devuelve errores y nunca cumple su cometido, por lo que se deduce que dicha función no está disponible en el *firmware*, al menos utilizando el método SITL. Además de las pruebas y comprobaciones llevadas a cabo, se analiza el código del *firmware* que implementa la mencionada funcionalidad, llegando a la conclusión de que la correcta ejecución del reinicio, simplemente apagaría el controlador y volvería a encenderlo. De aquí se deduce que el uso de esta funcionalidad no solucionaría el problema, pues una vez conectados el simulador y el controlador de vuelo, el reinicio del controlador haría que se rompiera la conexión entre ambos y no volviese a establecerse. Esto se debe a que el *autopilot* se inicia cuando recibe ciertos mensajes que los entornos de simulación envían al arrancar, por lo que si el simulador no se apaga y vuelve a encender, el *autopilot* no podrá recibir esos mensajes y conectarse a la simulación.

De este modo, se abandona la funcionalidad de reinicio del controlador, y se pasa a la segunda alternativa: encontrar una forma de reiniciar el módulo ekf2 (módulo del estimador en uso, basado en un *Extended Kalman Filter*⁹). Dicho módulo incluye 2 funcionalidades por defecto: *ekf2 start* y *ekf2 stop*, que inician o paran el estimador, respectivamente. Tras una considerable cantidad de pruebas con la consola de comandos de PX4, se determina que tras ejecutar el *reset* en el entorno de simulación, el reinicio del estimador (ejecución consecutiva de los comandos *ekf2 start* y *ekf2 stop*) consigue recalibrar el sistema y recuperar la posición, de modo que se confirma la utilidad de la solución escogida. En este punto, se comienza la implementación de dicha solución.

⁸Comentario de Chris Lovett en GitHub: <https://github.com/Microsoft/AirSim/issues/276#issuecomment-308551443>.

⁹Documentación acerca del estimador EKF2: https://dev.px4.io/en/tutorials/tuning_the_ecl_ekf.html.

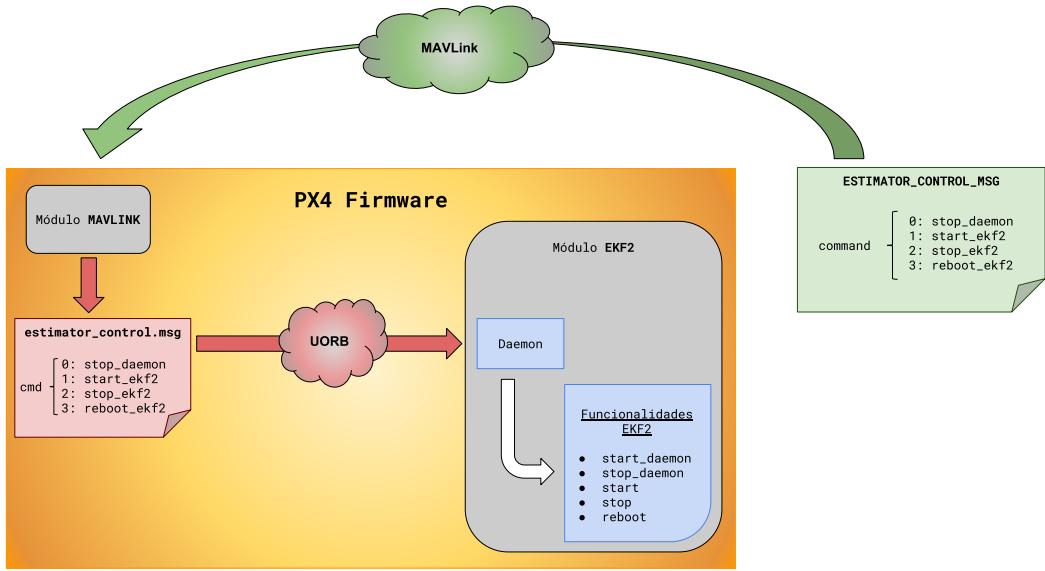


Figura 3: Esquema de la modificación realizada.

La figura 3 ilustra la funcionalidad implementada: para reiniciar el estimador, debe enviarse un mensaje MAVLink de tipo “**ESTIMATOR_CONTROL_MSG**” con el comando 3 (*reboot ekf2*). El *firmware* del PX4 recibirá este mensaje, y al comprobar su tipo, enviará un mensaje uORB¹⁰ de tipo “**estimator_control**” con el mismo comando que recibió. En el módulo *ekf2* habrá un *daemon* (esto es, un proceso que se mantiene en ejecución en segundo plano) esperando a recibir mensajes uORB que invocará a la función que corresponda, según el comando que haya recibido.

La implementación de estos cambios se divide en 2 partes: por un lado se realizaron cambios en los **módulos mavlink y ekf2**, y por otro se crearon las especificaciones del mensaje propio del estimador, para los **protocolos MAVLink y uORB**.

En el módulo *mavlink* se añadió la recepción y procesamiento del mensaje del estimador, donde se convierte el mensaje MAVLink a su correspondiente mensaje uORB, y se envía a través de este último protocolo.

En el módulo del estimador (módulo *ekf2*) se añadió un *daemon* que escucha mensajes uORB y ejecuta, según el comando que haya recibido, la función que corresponda. Por otra parte, se han añadido nuevas funcionalidades para arrancar y parar el *daemon*, así como una función de *reboot* equivalente a invocar las funciones de *stop* y *start* consecutivamente.

Tras una serie de pruebas de la solución implementada, se comprueba satisfactoriamente que el algoritmo de reinicio funciona, permitiendo su integración con el resto del sistema.

¹⁰uORB: es un protocolo asíncrono de comunicación basado en publicación/subscripción empleado por PX4 internamente. Documentación: <https://dev.px4.io/en/middleware/uorb.html>.

Volviendo al bloque de entrenamiento, se añade a la funcionalidad de *RESET* el envío de un mensaje MAVLink de tipo ESTIMATOR_CONTROL_MSG que reinicia el estimador, tras ejecutar el reinicio de la simulación. Una vez probado esto y confirmado su buen funcionamiento, se concluye la última etapa de integración de Gym, terminando así el desarrollo del trabajo.

3.3. Arquitectura del sistema final

Una vez descrito el desarrollo del proyecto, se concluirá el trabajo presentando una visión global de la arquitectura del sistema final.

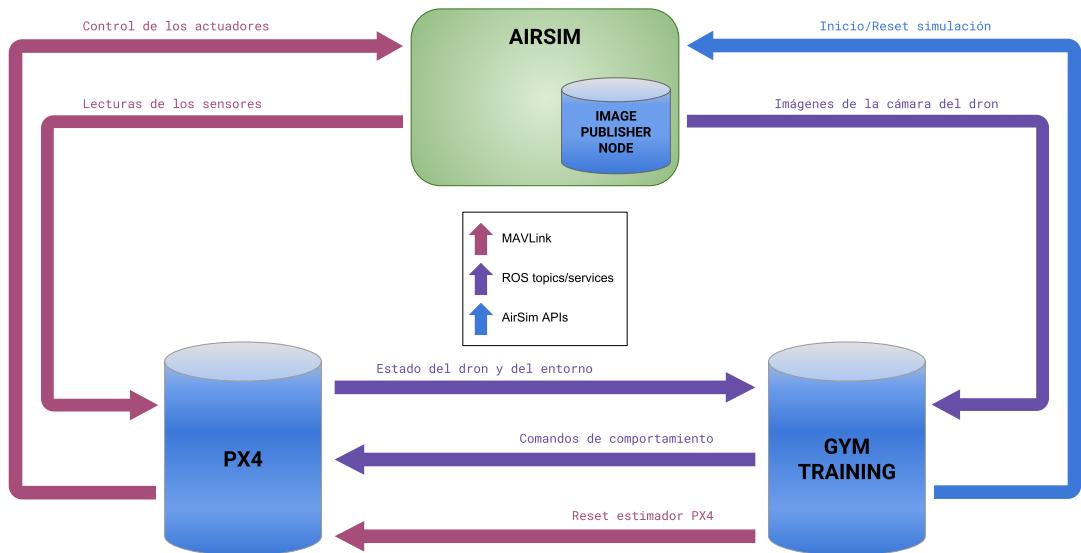


Figura 4: Arquitectura del sistema completo (alto nivel).

En la figura 4 se puede apreciar un esquema de la arquitectura global del sistema a alto nivel. En ella, son distinguibles los 4 bloques mencionados en los apartados anteriores: el bloque del entorno de simulación (**AirSim**), el bloque del controlador de vuelo (**PX4**), el bloque del sistema de entrenamiento y desarrollo de IA (**Gym Training**) y por último el bloque del sistema operativo robótico (**ROS**), que en este caso se puede apreciar en la presencia de nodos de computación así como en los canales de comunicación que se utilizan para interconectar el resto de bloques.

El funcionamiento del sistema comienza y termina en el bloque Gym Training, el “cerebro” encargado de controlar y gestionar el resto de los componentes. En primer lugar, el Gym Training inicia el entrenamiento reseteando el estado del vehículo a través de las APIs de AirSim y reiniciando los estimadores del *autopilot* mediante un mensaje MAVLink. A continuación, envía los comandos necesarios al PX4 para llevar el dron a la posición de inicio

del entrenamiento. A través de MAVLink, AirSim y PX4 intercambian en todo momento los datos de los sensores y los datos para los actuadores, controlando el comportamiento del dron. Al mismo tiempo, los datos de los sensores recibidos por el PX4 son publicados en ROS y leídos por el Gym Training, de igual forma que lo hacen las imágenes de AirSim publicadas por el nodo ROS. Todos estos datos son empleados por el módulo inteligente (Gym Training) para calcular la mejor acción en cada momento y enviársela a través de ROS al PX4. Una vez se realiza la acción, el Gym Training vuelve a leer el estado del dron (datos de los sensores y datos de las imágenes) y vuelve a calcular la mejor acción, al tiempo que aprende de la experiencia que ha vivido: haber pasado de un estado a otro a través de una acción. Este proceso se repite una y otra vez hasta el final del episodio, tras el cual se vuelve a hacer el *reset* y se inicia el siguiente episodio.

Una vez comprendida la arquitectura global, merece la pena adentrarse a más bajo nivel para analizar la parte implementada a mayores de los sistemas previamente existentes.

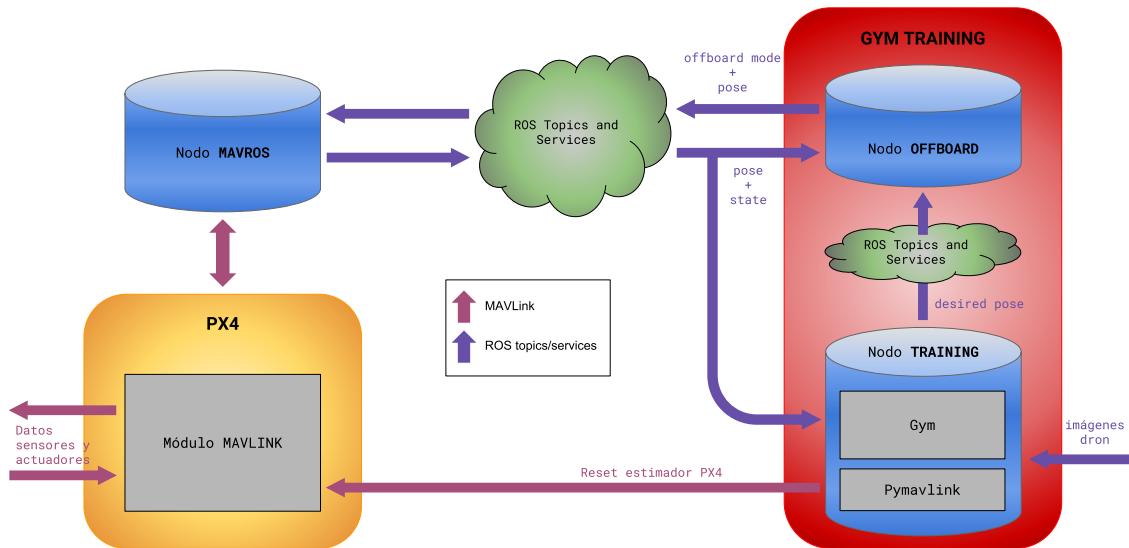


Figura 5: Arquitectura del sistema completo (bajo nivel).

En la figura 5 se puede observar un esquema a más bajo nivel de los bloques PX4 y Gym Training, y de las interfaces que utilizan para compartir información.

Abstrayéndose del complejo funcionamiento interno del *autopilot*, que puede ser consultado en [9], cabe destacar que la única interfaz de comunicación presente en el PX4 es MAVLink. A través del módulo MAVLink se realizan tanto las comunicaciones con el entorno de simulación (para leer los datos de los sensores y enviar los de los actuadores) como la activación del reinicio de los estimadores. Sin embargo, el *autopilot* es compatible con la integración de un nodo MAVROS¹¹, esto es, un nodo ROS que implementa la interfaz de comunicaciones

¹¹ Documentación paquete MAVROS: <http://wiki.ros.org/mavros>

MAVROS, actuando de proxy entre el canal de comunicación de ROS y los mensajes MAVLink en ambas direcciones, y permitiendo así recibir y enviar mensajes para interactuar con el controlador a través del sistema operativo robótico, y no solo con MAVLink.

En cuanto al bloque Gym Training, puede observarse que está compuesto por dos nodos diferentes: un nodo OFFBOARD, que es el encargado de mantener el vehículo en modo “*offboard*” para poder dirigirlo de forma automática (en lugar de ser controlado por un piloto humano)¹², y de enviar la pose (posición y orientación) deseada del vehículo; y un nodo TRAINING, que es el que implementa el sistema inteligente y el que, en función de la acción escogida, le envía la pose deseada al nodo OFFBOARD. Esta comunicación entre nodos se realiza mediante el canal de comunicación de ROS. Para finalizar, el nodo TRAINING emplea una implementación del protocolo MAVLink en Python (Pymavlink¹³) para poder enviar mensajes MAVLink y reiniciar el estimador del *autopilot*.

4. Conclusiones

El *framework* creado a lo largo de este proyecto cumple el objetivo de proporcionar un conjunto de potentes herramientas a los desarrolladores de sistemas autónomos para drones.

En cuanto a las ventajas que ofrece el resultado de este trabajo, destaca la gran versatilidad y flexibilidad que ofrece la integración de las tecnologías ROS, Gym y Python, no solo facilitando el desarrollo de cualquier tipo de algoritmo de entrenamiento, procesado de datos, interacción con elementos *hardware* a bajo nivel, etc; sino también ofreciendo la posibilidad de compatibilizar nuevas tecnologías y sistemas para seguir ampliando el *framework*.

Sin embargo, la ventaja más saliente viene dada por la filosofía *plug&play* que sigue el proyecto. La integración de PX4, un *firmware* de *autopilot* diseñado para vehículos reales, permite que toda la implementación del sistema inteligente se pueda migrar directamente al sistema *onboard*¹⁴ del equipo real, ya que la interfaz de comunicación con el *autopilot* será exactamente la misma. Además, a pesar de que el dron real no utilice la versión de PX4 modificada empleada en la simulación, sino la oficial, el comportamiento del sistema será idéntico (ya que la modificación únicamente añade una funcionalidad de reseteo que en una situación real no tiene cabida). Esto implica que una vez se haya entrenado el algoritmo de control en el entorno de simulación, se podrá conectar directamente al dron real para que realice el pilotaje.

¹²Documentación "modo OFFBOARD": https://docs.px4.io/en/flight_modes/offboard.html

¹³Documentación Pymavlink: <https://www.ardusub.com/developers/pymavlink.html>

¹⁴Sistema *onboard*: unidad hardware y software integrada en el vehículo.

4.1. Usabilidad

De cara al desarrollador que desee emplear el *framework* propuesto, se ofrecerá una documentación¹⁵ que guiará la instalación de todos los componentes necesarios. **METER REFERENCIA A LA DOCUMENTACION DE INSTALACION**

Con el sistema instalado, el desarrollador contará con una potente herramienta que le permitirá desde adaptar el módulo Gym Training (agente, entorno y algoritmo) para hacer un entrenamiento rápido, hasta profundizar en la integración de nuevas tecnologías que le permitan hacer algoritmos más complejos o sistemas de entrenamiento diferentes de Gym.

Una vez el sistema esté entrenado, el desarrollador podrá migrar el algoritmo entrenado al dron real, si este integra los mismos componentes que el *framework* (PX4 y ROS). La única modificación que deberá implementar será el cambio implícito que supone pasar de un entorno de entrenamiento a un entorno de explotación: eliminar la repetición de episodios, y con ello el reseteo de la simulación y el controlador, así como cambiar el objetivo simulado por el real.

4.2. Líneas futuras

Con una visión de mejora continua, existen dos líneas que sería interesante trabajar en el futuro:

La primera se basa en experimentar con el *framework* entrenando un dron con diferentes algoritmos de *Reinforcement Learning*, para posteriormente comparar los resultados y llevar el experimento a un dron real, comprobando así el gran potencial del trabajo.

Por otra parte, el continuo avance de la tecnología hace imprescindible la compatibilización de este trabajo con las nuevas versiones de las herramientas empleadas, no solo para evitar que se quede obsoleto, sino para aprovechar las diferentes mejoras y cambios que se vayan publicando como oportunidades de mejorar el *framework* y adaptarlo a las tecnologías punteras de cada momento.

¹⁵Documentación disponible en:

Anexos

Anexo A Estado del arte

Desarrollar y experimentar con algoritmos para vehículos autónomos es un proceso muy complejo que requiere tiempo y muchas pruebas que no siempre terminan bien. Además, para desarrollar sistemas inteligentes a menudo es necesario recopilar una gran cantidad de datos de entrenamiento en diferentes condiciones y entornos. Realizar este proceso en el mundo real no es viable, por ello para llevarlo a cabo se utilizan entornos de simulación, sistemas informáticos que procuran imitar lo más fielmente posible el comportamiento que tendría el vehículo en un entorno real. Este tipo de *software* nos permite experimentar, analizar y entrenar a nuestros aparatos evitando los importantes inconvenientes que implican hacer lo propio en la realidad: accidentes, golpes, costosas roturas, desajustes, etc.

Cabe destacar que no solo el **entorno de simulación** es importante, sino también su compatibilidad e integración con otras tecnologías tales como los **controladores de vuelo**, que son dispositivos encargados de controlar los diferentes componentes del vehículo (toman información de los sensores, y en función del estado del vehículo actúan sobre los motores para mantener el sistema en vuelo); los **protocolos de comunicación** para interconectar el simulador y el controlador, así como también otros componentes; o los **frameworks de programación** para desarrollar el *software* que definirá el comportamiento del dron.

Debido a que el interés de este trabajo reside en encontrar un sistema que agrupe todas estas tecnologías para facilitar la implementación de sistemas de vuelo autónomo, es esencial hacer un análisis del estado del arte al respecto¹⁶:

A día de hoy existe una gran variedad de entornos de simulación para drones, y entre ellos, uno de los más utilizados es **Gazebo** [10]. Un simulador de código abierto para todo tipo de robots, capaz de simular ambientes interiores y exteriores complejos con alto grado de fidelidad. Provee una amplia biblioteca de robots y entornos, gran cantidad de sensores, y *plugins* para integrar muchas otras tecnologías, tales como controladores de vuelo, *frameworks* de programación, sistemas de entrenamiento, etc. Algunas de las grandes ventajas que tiene utilizar Gazebo son la amplia comunidad de usuarios que participan activamente generando conocimiento y la gran cantidad de documentos académicos basados en la experimentación con este simulador.

La principal desventaja de Gazebo es su falta de realismo en la simulación, debido a su orientación a robots genéricos y a la complejidad que implica crear entornos realistas a gran escala y con gran cantidad de detalles que simulen mejor el mundo real.

Una alternativa que pretende solventar la falta de realismo es **Hector** [11], un trabajo orientado específicamente a cuadricópteros, y que integra el simulador Gazebo junto al *framework*

¹⁶Fuente de datos comparativos entre *frameworks*: [2]

robótico ROS [B.2] . Lo que ofrece Hector¹⁷ es una simulación más realista de la física del vuelo y el comportamiento del aparato, pero por el contrario carece de soporte para la integración de controladores de vuelo como PX4 [B.1] y protocolos de comunicación estándar como MAVLink¹⁸ [1].

De forma similar, **RotorS** [12, 13] es un *framework* modular de simulación que permite diseñar drones y desarrollar algoritmos para controlar su comportamiento y estimar su estado de forma más precisa. Al igual que en el caso anterior, RotorS¹⁹ se basa en ROS [B.2] y Gazebo, pero en este caso sí se soporta la integración de controladores de vuelo.

Garantizando la compatibilidad de controladores de vuelo y alejándose de la simulación de Gazebo, existe otro *software* llamado **jMAVSIM**²⁰, un extremadamente sencillo entorno de simulación diseñado en el contexto del proyecto Pixhawk²¹ con el objetivo de experimentar con el *firmware* de la plataforma del propio proyecto. Se trata de un entorno muy ligero y fácil de utilizar que consta de un motor de renderizado muy simple, imposibilitando la generación de objetos en la escena. Además no aporta ningún tipo de integración con otras tecnologías.

Otro grupo de entornos de simulación es aquel basado en el potente motor de videojuegos creado por Epic Games²², el Unreal Engine²³. Este grupo de *frameworks* se caracteriza por el alto grado derealismo tanto en gráficos como en física de comportamiento de los vehículos, debido a las avanzadas tecnologías desarrolladas por su creador. Notables ejemplos de este grupo son Sim4CV, DroneSimLab y Microsoft AirSim.

El primero de ellos, **Sim4CV**²⁴ [14, 15], se presenta como un simulador fotorrealista orientado claramente a la experimentación en campos muy avanzados de la visión artificial. Gracias al potente motor gráfico, consigue alcanzar un alto nivel de realismo en la física de los vehículos. Como desventaja está su limitación para integrar tecnologías externas al propio *framework*.

En contraste con el anterior, **DroneSimLab**²⁵ [16] es un *framework* con énfasis en la modularidad y la flexibilidad. Tanto es así, que no solo integra varios controladores de vuelo y motores de simulación de sensores, además del motor gráfico aportado por Unreal, sino que facilita la integración de otros muchos componentes mediante el uso de contenedores. Este conjunto de herramientas tiene una orientación genérica, permitiendo experimentar en visión artificial, vehículos autónomos y otros muchos campos para gran cantidad de vehículos. Claramente, este *framework* es la alternativa más cercana e interesante al objetivo del

¹⁷ Documentación del paquete ROS *Hector*: http://wiki.ros.org/hector_quadrotor

¹⁸ Documentación oficial protocolo MAVLink: <https://MAVLink.io/en/>

¹⁹ Documentación del paquete ROS *RotorS*: http://wiki.ros.org/rotors_simulator

²⁰ Documentación jMAVSIM: <https://dev.px4.io/en/simulation/jmavsim.html>

²¹ Sitio web del proyecto Pixhawk con acceso a la documentación oficial: <http://pixhawk.org/>

²² Sitio web de Epic Games: <https://www.epicgames.com/site/es-ES/home>

²³ Documentación Unreal Engine 4: <https://docs.unrealengine.com/en-us/>.

²⁴ Sitio web de Sim4CV: <https://sim4cv.org>

²⁵ Sitio web de DroneSimLab: <https://github.com/orig74/DroneSimLab>

presente trabajo.

Microsoft AirSim [2] forma parte del proyecto *Aerial Informatics and Robotics Platform*²⁶ desarrollado por Microsoft, y consiste en un entorno de simulación basado en el motor Unreal Engine y en un conjunto de herramientas para el desarrollo de sistemas autónomos. Se trata de una plataforma que cuenta con las tecnologías más avanzadas de simulación en cuanto a física de los vehículos y visualización del entorno, reduciendo al máximo la diferencia entre realidad y simulación, y ayudando así a crear sistemas autónomos más avanzados, seguros y eficaces.

AirSim es un *framework* de código abierto en continuo desarrollo que pretende convertirse en una plataforma modular, altamente compatible y tecnológicamente avanzada para contribuir al desarrollo e investigación en vehículos autónomos, con la intención de crear “*una comunidad de desarrolladores que impulse el estado del arte en este campo*” [2].

La novedad de AirSim y su apuesta por la muy alta fidelidad de la simulación, además de la creciente presencia de trabajos realizados con esta plataforma y el prestigio de la propia compañía, han sido los factores impulsores de la elección de este entorno de simulación para realizar este trabajo fin de grado.

Existen muchos más entornos de simulación, pero la mayoría están orientados a robots de tipo genérico (como **ARGoS**²⁷ o **V-REP**²⁸), por lo que no incorporan una simulación realista del comportamiento de un dron ni soportan compatibilidad con la mayoría de tecnologías necesarias, o bien no son de código abierto (como **XPlane**²⁹ o **RealFlight**³⁰).

Anexo B Tecnologías utilizadas

El *framework* de simulación, desarrollo y entrenamiento de sistemas *Reinforcement Learning* que se crea, explica y documenta en este trabajo fin de grado, se construye a partir de diferentes tecnologías específicas del campo de la robótica, la aeronáutica y la inteligencia artificial. Puesto que estos campos no se consideran en absoluto comúnmente conocidos, en los siguientes apartados se explicarán algunos conceptos para el lector que no esté familiarizado con la materia.

²⁶Sitio web del proyecto AIRP: <https://www.microsoft.com/en-us/research/project/aerial-informatics-robotics-platform/>.

²⁷Sitio web de ARGoS: <http://www.argos-sim.info/index.php>

²⁸Sitio web de V-REP: <http://www.coppeliarobotics.com/index.html>

²⁹Sitio web de XPlane: <http://www.x-plane.com/>

³⁰Sitio web de RealFlight: <https://www.realflight.com/>

B.1 Controlador de vuelo: PX4

El controlador de vuelo o “*autopilot*” es uno de los componentes base más esenciales de cualquier vehículo multi-rotor. Se trata de una placa electrónica a la que se conectan los sensores (GPS, barómetros, giroscopios, acelerómetros, etc) y los actuadores (motores) del vehículo. Esta placa lleva instalado un *firmware* que se encarga de controlar el comportamiento de los actuadores en función de las lecturas de los sensores y del estado o posición del vehículo al que se desee llegar, con la finalidad de mantener la aeronave estable así como de cambiar la velocidad o la dirección de movimiento de la misma, de forma controlada.

Siendo más concretos, la labor principal del *autopilot* es tomar el estado o posición deseada del vehículo como entrada, estimar el estado real a partir de los datos que generan los sensores, y controlar el comportamiento de los actuadores de tal forma que el estado real sea lo más parecido posible al estado deseado.³¹

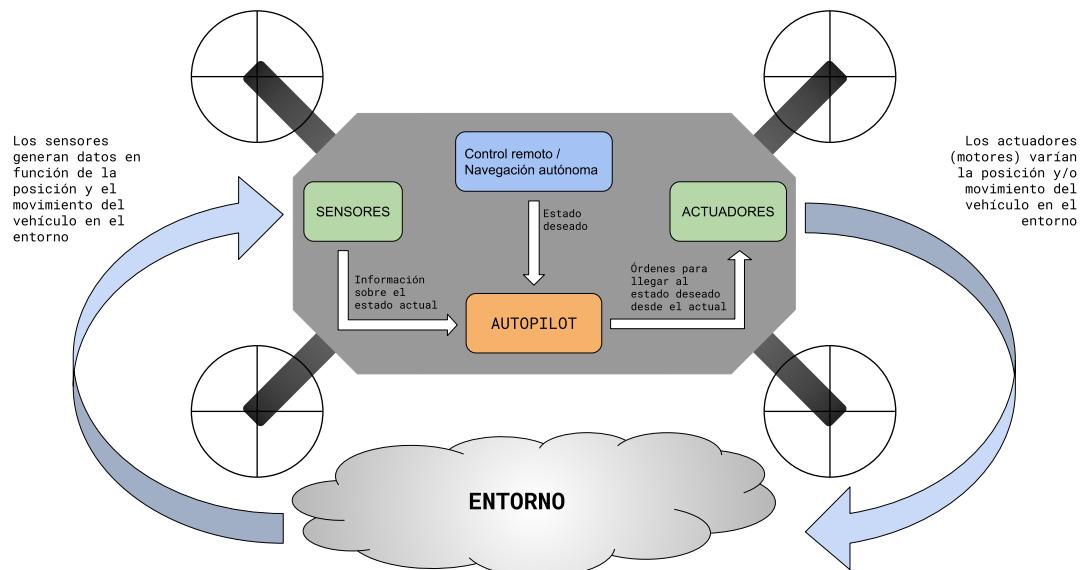


Figura 6: Arquitectura general de un controlador de vuelo.

Cuando se utilizan entornos de simulación, los sensores y los actuadores son emulados por la física del vehículo, de forma que el controlador de vuelo consume los datos simulados de los sensores, y genera unas señales que recibirán los actuadores del vehículo simulado como entrada. Para integrar el *autopilot* en un entorno de simulación, existen dos métodos diferentes: SITL y HITL.

SITL (Software In The Loop) [17, 18] es una técnica que se basa en emular el *hardware* del controlador ejecutando directamente el *firmware* del *autopilot* sobre el equipo de

³¹Datos recogidos de la documentación de AirSim sobre controladores de vuelo: https://github.com/Microsoft/AirSim/blob/master/docs/flight_controller.md

desarrollo, de forma que interactúe con los sensores y actuadores del entorno de simulación.

HITL (Hardware In The Loop) [17] [18], en cambio, se basa en conectar el *hardware* real del *autopilot* al equipo de desarrollo, de forma que el controlador de vuelo interactúe directamente con los sensores y actuadores del entorno de simulación.

El método HITL es el más cercano al funcionamiento real, pero la ventaja más significativa del método SITL es que elimina por completo la necesidad de adquirir una plataforma *hardware*. Además, con el método HITL pueden darse problemas de desincronización si el reloj del simulador y del controlador tienen velocidades y/o precisiones distintas, o si la conexión entre el *autopilot* y el equipo de desarrollo no es capaz de transferir los datos suficientemente rápido. Sin embargo, cuando se trabaja con equipos de desarrollo con recursos limitados, el método HITL evita la sobrecarga de trabajo del equipo, permitiendo emplear toda la potencia disponible en el entorno de simulación, que normalmente requiere gran cantidad de recursos.

El *projeto Dronecode*³² es una plataforma de código abierto que trabaja en el desarrollo de diversas herramientas y tecnologías para drones, como protocolos de comunicación, controladores de vuelo (*hardware* y *firmware*), simuladores, etc. Su objetivo es crear una comunidad de desarrolladores que contribuyan al proyecto para ofrecer una plataforma tecnológicamente avanzada de herramientas para vehículos no tripulados. Uno de los proyectos más destacados de la *Dronecode Platform* es **PX4 autopilot**³³, un *firmware* de controlador de vuelo³⁴ muy versátil y ampliamente utilizado. Sus características más destacables son la gran cantidad de vehículos que puede controlar (no solo todo tipo de aeronaves, sino también vehículos terrestres y submarinos), la variedad de modos de vuelo y características de seguridad que ofrece, y especialmente su compatibilidad con multitud de sensores, periféricos y plataformas *hardware*³⁵.

En este trabajo fin de grado, se utilizará el PX4 en modo SITL.

La elección de este controlador se debe a diversos factores: en primer lugar, es un *autopilot* compatible con el entorno de simulación de Microsoft; además utiliza el protocolo MAVLink [1] (también soportado por AirSim), y es compatible con el *framework* ROS [B.2] para el desarrollo de algoritmos, el envío de comandos, etc; por otra parte, el hecho de que este *autopilot* se pueda montar sobre diferentes plataformas³⁶ (Pixhawk³⁷, Qualcomm Snapdragon

³²Sitio web de Dronecode con documentación disponible: <https://www.dronecode.org/>

³³Sitio web de *PX4 autopilot* con documentación disponible: <http://px4.io/>

³⁴Repositorio del *firmware* disponible en: <https://github.com/PX4/Firmware>

³⁵Características de *PX4 autopilot*: https://docs.px4.io/en/getting_started/px4_basic_concepts.html

³⁶*Hardware* compatible con el *firmware* PX4: https://docs.px4.io/en/flight_controller/

³⁷Referencia de *hardware* Pixhawk Series: https://docs.px4.io/en/flight_controller/pixhawk_series.html

Flight³⁸, Intel Aero Compute Board³⁹, Raspberry Pi 2/3 Navio2⁴⁰, etc) aporta una gran flexibilidad y compatibilidad al sistema.

La integración de PX4 con AirSim se realizará por el método SITL, ya que se pretende obtener un *framework* de simulación que no dependa de ningún elemento *hardware* más allá del propio equipo de desarrollo.

B.2 Framework de programación: ROS

ROS (Robot Operating System) [4] consiste en un conjunto de herramientas, librerías y convenciones de código abierto, desde drivers hasta avanzados algoritmos estado del arte, que componen un *framework* de programación orientado a robótica⁴¹.

A pesar de su nombre, a menudo se define ROS como un meta sistema operativo más que como un sistema operativo convencional. Esto se debe a que, aunque tiene características propias de los sistemas operativos, como la abstracción de hardware, el control de dispositivos a bajo nivel o la gestión de paquetes; también tiene rasgos más propios de *middlewares* o *frameworks*, como la implementación de funcionalidades básicas del sistema, el paso de mensajes entre procesos, y la existencia de herramientas y librerías de algoritmos a alto nivel⁴².

ROS se caracteriza por basarse en un sistema de computación distribuida, donde cada proceso se denomina nodo, y cada nodo es responsable de una tarea. Los diferentes nodos pueden comunicarse mediante un modelo de **publicación/subscripción**, para comunicaciones asíncronas y visibles para todos los nodos, o bien mediante un modelo de **petición/respuesta**, si lo que se pretende es establecer una comunicación síncrona y privada entre dos nodos⁴³.

³⁸Referencia de *hardware* Qualcomm Snapdragon Flight: https://docs.px4.io/en/flight_controller/snapdragon_flight.html

³⁹Referencia de *hardware* Intel Aero Compute Board: <https://software.intel.com/en-us/aero/compute-board>

⁴⁰Referencia de *hardware* Raspberry Pi Navio2: https://docs.px4.io/en/flight_controller/raspberry_pi_navio2.html

⁴¹Fuente: <http://www.ros.org/about-ros/>

⁴²Fuente: http://wiki.ros.org/ROS/Introduction#What_is_ROS.3F

⁴³Fuente: <http://wiki.ros.org/ROS/Concepts>

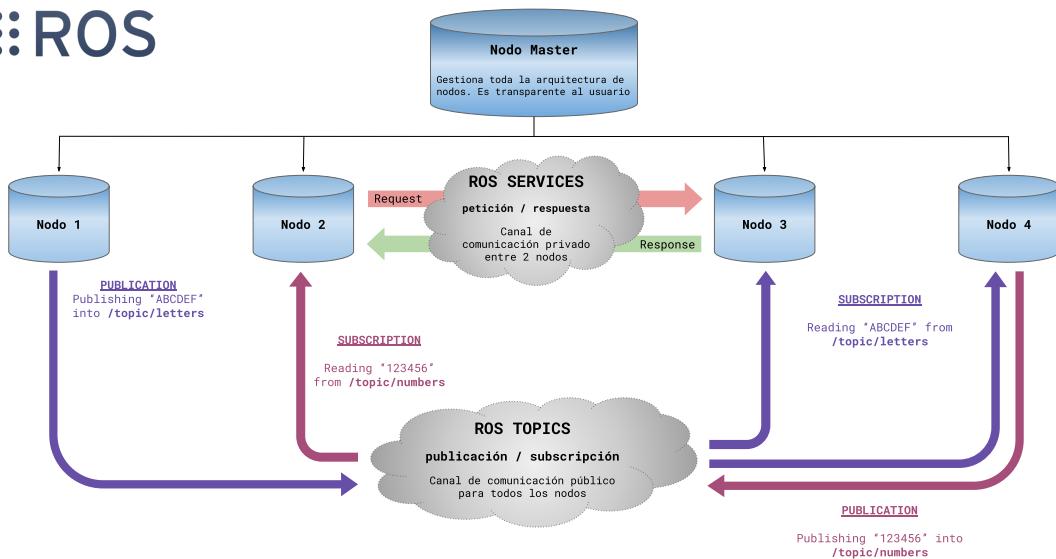


Figura 7: Esquema de la arquitectura de ROS a alto nivel.

En el contexto de este trabajo, ROS resulta un *framework* de gran utilidad ya que permite acceder a las lecturas de los sensores y las cámaras, enviar comandos a través de MAVLink [1], e implementar hasta los algoritmos más complejos. Además, esta plataforma ofrece integración con otros proyectos de código abierto como OpenCV⁴⁴, una importante librería para visión artificial, o PointCloudLibrary⁴⁵, una librería de percepción 3D, entre otros⁴⁶. Estas librerías son de gran utilidad a la hora de implementar algoritmos de visión artificial para guiar el dron.

Por otra parte, existen implementaciones de ROS en Python y en C++ (entre otros lenguajes) que añaden a la versatilidad implícita de ROS toda la potencia de dichos lenguajes.

Por todas estas razones, en este trabajo fin de grado se integrará ROS como *framework* de desarrollo general.

B.3 Entorno de simulación: AirSim

Nota: en este apartado se mencionan los conceptos de controlador de vuelo, autopilot, PX4, modo SITL, modo HITL y ROS, que se explican en los anexos B.1 y B.2 . Se recomienda al lector ajeno a la materia, que revise dichos apartados antes de continuar con el actual.

⁴⁴OpenCV: <https://opencv.org/>

⁴⁵PointCloud: <http://pointclouds.org/>

⁴⁶Integración de ROS con otras tecnologías (fuente documentación): <http://www.ros.org/integration/>

Microsoft AirSim⁴⁷ es un entorno de simulación desarrollado sobre el motor Unreal Engine (UE) que ofrece simulaciones física y visualmente realistas (figura 8). Está implementado como un plugin multiplataforma para UE que se puede integrar en cualquier proyecto de Unreal.



Figura 8: Captura de pantalla del simulador AirSim en un entorno urbano. En la parte inferior se observan 3 tipos de cámaras (profundidad, segmentación y visión real). [2]

Se trata de una plataforma modular en continuo desarrollo, con gran énfasis en la escalabilidad, dando la posibilidad de añadir nuevos vehículos, plataformas *hardware* y protocolos *software*. Además se trata de un proyecto de código abierto, por lo que cualquier desarrollador puede contribuir y utilizarlo.

Incluye un conjunto de APIs⁴⁸ que permiten leer el estado del vehículo, de los sensores y de las cámaras, así como también enviar comandos al vehículo.

Este entorno de simulación provee 3 interfaces diferentes de uso⁴⁹:

- **Control manual:** se trata de una interfaz en la que el usuario controla el comportamiento del vehículo con ayuda de un *joystick* control remoto conectado al simulador.
- **Control automático:** con esta interfaz, el usuario no tiene control directo sobre el

⁴⁷Repositorio AirSim: <https://github.com/Microsoft/AirSim>

⁴⁸Documentación APIs AirSim: <https://github.com/Microsoft/AirSim/blob/master/docs/apis.md>

⁴⁹Fuente (Readme AirSim) <https://github.com/Microsoft/AirSim/blob/master/README.md>

vehículo. Con la ayuda de las diferentes APIs y *frameworks* disponibles, es posible crear algoritmos para experimentar con vehículos autónomos.

- **Modo Visión Artificial:** esta interfaz está contemplada para usuarios que solamente están interesados en la parte gráfica del entorno. En este modo el simulador elimina el vehículo y su física de comportamiento, y ofrece el uso de APIs de visión artificial para explorar el entorno y experimentar con algoritmos en este campo.

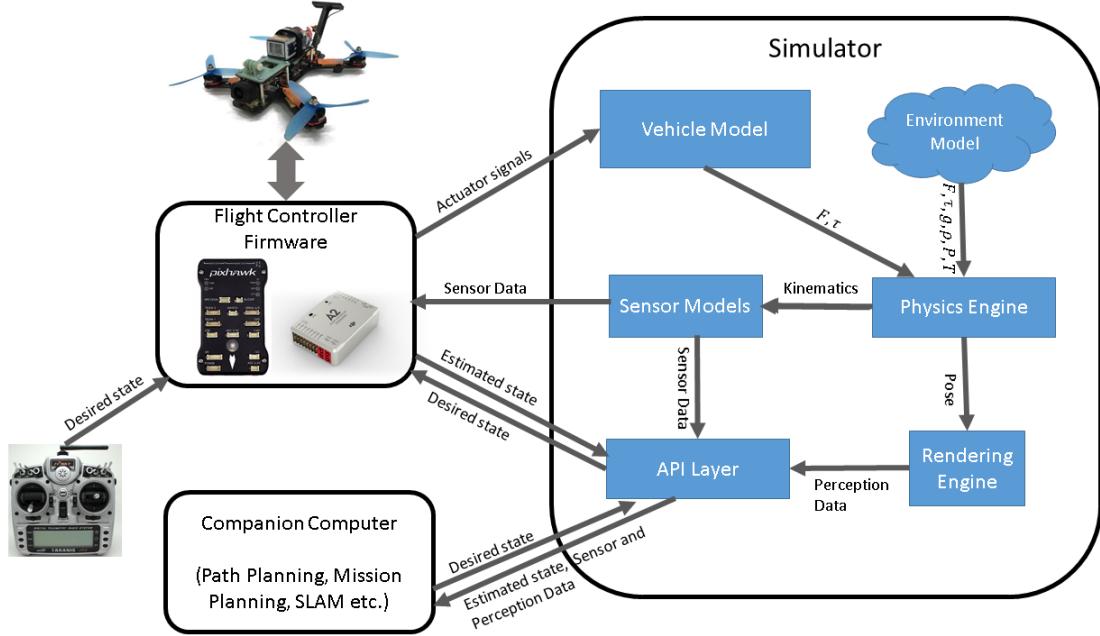


Figura 9: Arquitectura de Microsoft AirSim. [2]

Actualmente AirSim ofrece la posibilidad de utilizar dos controladores de vuelo: **Simple Flight** en modo SITL o **PX4** en modo SITL o HITL. En el futuro se pretende compatibilizar otros *autopilots* como *ROSflight*⁵⁰ y *Hackflight*⁵¹.

Simple Flight es un controlador de vuelo desarrollado por Microsoft y entendido como un conjunto de algoritmos empaquetados en una librería de cabeceras C++ sin dependencias. La principal ventaja que ofrece es la posibilidad de utilizar este *autopilot* en simulación o sobre un *hardware* real indistintamente, al contrario que la mayoría de controladores, que requieren una complicada configuración para hacerlos funcionar en modo SITL. Actualmente Simple Flight está en desarrollo, y la intención es adaptarlo a diferentes controladores *hardware*

⁵⁰ ROSflight <http://docs.rosflight.org/en/latest/>

⁵¹ Hackflight <https://github.com/simondlevy/Hackflight>

ampliamente utilizados como son las placas Pixhawk V2⁵² y Naze 32⁵³.

PX4 es un controlador de vuelo real de código abierto ampliamente utilizado y con soporte para multitud de dispositivos *hardware* y sensores. AirSim proporciona soporte para integrar este *autopilot* en modo SITL o HITL, sin embargo, la integración de este controlador sigue resultando considerablemente complicada en ambos casos.

En cuanto a compatibilidad con otras tecnologías, actualmente AirSim utiliza una librería propia⁵⁴ basada en MAVLink [1], un protocolo de comunicación para drones comúnmente utilizado, que permite la integración de PX4. Además también ofrece soporte para integrar ROS [B.2], permitiendo interactuar con los datos de la simulación directamente desde dicho *framework*.

B.4 Reinforcement Learning: Gym

El *Reinforcement Learning* (RL) o Aprendizaje por Refuerzo es un área del *Machine Learning* que permite a un agente (entidad que va a aprender) descubrir un comportamiento óptimo de forma autónoma a través de interacciones de prueba y error con el entorno que lo rodea. En lugar de proporcionarle al agente la solución a un problema, en RL el entorno le devolverá una recompensa positiva (refuerzo) o negativa (castigo) al agente según la acción que haya llevado a cabo. Con esto, el agente aprenderá qué tipo de conductas son las que debe mantener para obtener recompensas cada vez más positivas [19, 20, 21, 22].

La razón de utilizar este tipo de aprendizaje para desarrollar sistemas autónomos reside en la inmensa complejidad que supone programar las infinitas situaciones que se pueden dar en un entorno real y que un vehículo debe saber resolver. Ante esta complejidad, resulta muy atractiva la solución de crear un sistema que pueda generar ese conocimiento por sí mismo a través de la asociación de relaciones causa-efecto de su propia experiencia.

⁵²Referencia hardware Pixhawk V2 https://docs.px4.io/en/flight_controller/pixhawk-2.html

⁵³Referencia hardware Naze 32 https://quadquestions.com/Naze32_rev6_manual_v1.2.pdf

⁵⁴MAVLinkCom: <https://github.com/Microsoft/AirSim/blob/master/MavLinkCom>

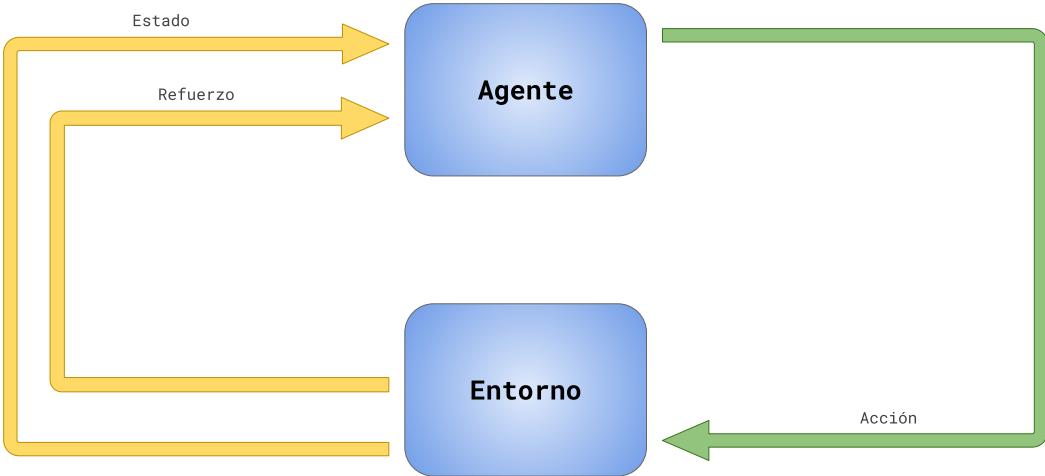


Figura 10: Ilustración del funcionamiento de un sistema RL: el agente aprende que partiendo de un estado S_0 y aplicando una acción A_1 , llega a un nuevo estado S_1 y recibe un refuerzo R_1 .

El objetivo de este trabajo es la obtención de un *framework* destinado a desarrollar y entrenar sistemas RL, por lo que dicho *framework* deberá tener un sistema de entrenamiento basado en prueba y error, es decir, en la repetición sucesiva de un episodio de entrenamiento al final del cual, se volverá a la situación inicial y se empezará de nuevo el episodio. Durante cada episodio, el dron intentará alcanzar su objetivo a base de ejecutar las diferentes acciones que tenga definidas (por ejemplo, moverse en una dirección o girar) e irá acumulando experiencia en función de los refuerzos que el sistema le devuelva.

Para implementar esta metodología de entrenamiento, se integrará Gym en nuestro proyecto, un *toolkit* de RL creado por OpenAI.

OpenAI⁵⁵ es una compañía de investigación sin ánimo de lucro especializada en Inteligencia Artificial (IA), que tiene como objetivo crear una *Inteligencia Artificial General* (AGI, por sus siglas en inglés) segura y accesible para toda la humanidad.

Gym⁵⁶ [6] consiste en un conjunto de herramientas para desarrollar y comparar algoritmos de RL. Se trata de un paquete de Python que contiene una colección de entornos y que provee unas guías de referencia para desarrollar agentes que interactúen con dichos entornos, así como para crear entornos nuevos. Tanto los entornos como los agentes siguen unas estructuras e interfaces determinadas, permitiendo así intercambiar los agentes y los algo-

⁵⁵Más información sobre OpenAI: <https://openai.com/about/>

⁵⁶Más información sobre OpenAI Gym: <http://gym.openai.com/>

ritmos de aprendizaje que estos utilizan. De este modo, se pueden comparar diversos tipos de agentes entrenados en el mismo entorno.

En el contexto del presente proyecto, el entorno será la simulación, y los agentes serán los drones, implementando diferentes algoritmos de entrenamiento, de forma que se pueda comparar el rendimiento de los diferentes algoritmos en las mismas condiciones.

Referencias

- [1] S. Atoev, K.-R. Kwon, S.-H. Lee, and K.-S. Moon, “Data analysis of the mavlink communication protocol,” *2017 International Conference on Information Science and Communications Technologies (ICISCT)*, pp. 1–3, 2017.
- [2] S. Shah, D. Dey, C. Lovett, and A. Kapoor, “Airsim: High-fidelity visual and physical simulation for autonomous vehicles,” in *Field and Service Robotics*, 2017. [Online]. Available: <https://arxiv.org/abs/1705.05065>
- [3] L. Meier, D. Honegger, and M. Pollefeys, “Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015.
- [4] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA Workshop on Open Source Software*, 2009.
- [5] A. Mahtani, L. Sánchez, E. Fernández, and A. Martínez, *Effective robotics programming with ROS: find out everything you need to know to build powerful robots with the most up-to-date ROS*, 3rd ed. Packt Publishing, 2016.
- [6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *CoRR*, vol. abs/1606.01540, 2016.
- [7] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, “Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo,” *CoRR*, vol. abs/1608.05742, 2016.
- [8] R. Tellez, “Using openai with ros,” 2018. [Online]. Available: <http://www.theconstructsim.com/using-openai-ros/>
- [9] D. P. Inc, “Px4 architectural overview.” [Online]. Available: <https://dev.px4.io/en/concept/architecture.html>
- [10] O. S. R. Foundation, “What is gazebo?” [Online]. Available: http://gazebosim.org/tutorials?tut=guided_b1&cat=#WhatisGazebo?
- [11] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. von Stryk, “Comprehensive simulation of quadrotor uavs using ros and gazebo,” in *SIMPAR*, 2012.
- [12] M. A. Fadri Furrer, Michael Burri and R. Siegwart, “Rotors - a modular gazebo mav simulator framework,” 2015.
- [13] A. S. Lab and E. Zurich, “Rotors simulator.” [Online]. Available: <https://www.autonomousrobotslab.com/rotors-simulator2.html>

- [14] M. Mueller, V. Casser, J. Lahoud, N. Smith, and B. Ghanem, “A photo-realistic simulator for computer vision applications,” 2018.
- [15] M. Müller, V. Casser, J. Lahoud, N. Smith, and B. Ghanem, “Sim4cv: A photo-realistic simulator for computer vision applications,” 2018. [Online]. Available: <https://youtu.be/SqAxzsQ7qUU>
- [16] O. Ganoni and R. Mukundan, “A framework for visually realistic multi-robot simulation in natural environment,” *arXiv preprint arXiv:1708.01938*, 2017, wSCG 2017 proceedings.
- [17] E. N. Johnson and S. Fontaine, “Use of flight simulation to complement flight testing of low-cost uavs,” 2001.
- [18] C. Coopmans, M. Podhradský, and N. Hoffer, “Software- and hardware-in-the-loop verification of flight dynamics model and flight control simulation of a fixed-wing unmanned aerial vehicle,” *2015 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED-UAS)*, pp. 115–122, 2015.
- [19] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.
- [20] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” *I. J. Robotics Res.*, vol. 32, pp. 1238–1274, 2013.
- [21] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed. Pearson, 2016.
- [22] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, 2nd ed. The MIT Press, 2018.