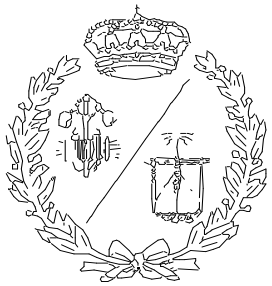
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA



Proyecto Fin de Grado

ENTORNO VIRTUAL PARA LA SIMULACIÓN DE UN VEHICULO SUBMARINO OPERADO REMOTAMENTE (ROV)

(VIRTUAL ENVIRONMENT FOR THE SIMULATION OF AN UNMANNED UNDERWATER VEHICLE)

Para acceder al Título de

GRADUADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: Mario Charlón Soldevilla
Director: Luciano Alonso Rentería

Julio - 2024

Agradecimientos,

A mis padres y amigos, por haberme apoyado en todo momento. A mis profesores, por el conocimiento que me han proporcionado durante estos años.

RESUMEN

En este trabajo se aborda el diseño de una escena virtual para la simulación de un vehículo operado de forma remota (ROV).

El vehículo utilizado será el BlueROV2, de la empresa BlueRobotics, diseñado para la exploración marina. Este modelo tiene un tamaño compacto, es fácil de utilizar y es bastante asequible. El ROV será operado de manera autónoma siguiendo una trayectoria especificada por el usuario.

Se utilizarán diferentes herramientas para programar un entorno subacuático realista en el que se realice un control del vehículo BlueROV2 para el seguimiento de una ruta de puntos específica. Se creará una interfaz gráfica que agrupe todos los ejecutables necesarios para realizar la simulación del ROV incluyendo el protocolo de comunicación MAVLink y un entorno de simulación en bucle cerrado (Software In The Loop), permitiendo la simulación del ROV sin hardware físico. En esta interfaz se podrá seleccionar la escena creada y simular el seguimiento de una ruta de puntos que se desee.

Al finalizar la simulación se podrá realizar un análisis de los datos recopilados y exportarlos a cualquier herramienta que el usuario desee, facilitando en este caso una opción que exporte los datos a la plataforma MATLAB.

ABSTRACT

This project consists in the design of a virtual scene for the simulation of a remotely operated vehicle (ROV).

The vehicle used will be the BlueROV2 from BlueRobotics, designed for marine exploration. This model is compact, easy to use, and quite affordable. The ROV will be operated autonomously, following a trajectory specified by the user.

Various tools will be used to program a realistic underwater environment in which the BlueROV2 will be controlled to follow a specific waypoint route. A graphical interface will be created to integrate all the necessary executables for the ROV simulation, including the MAVLink protocol and a Software In The Loop (SITL) simulation environment, allowing for the simulation of the ROV without physical hardware. In this interface, users can select the created scene and simulate the desired waypoint route.

Upon completion of the simulation, the collected data can be analyzed and exported to any tool the user wishes, providing an option to export the data to the MATLAB platform.

INDICE GENERAL

DOCUMENTO 1: MEMORIA

INDICE

INDICE DE FIGURAS

INDICE DE ECUACIONES

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

El mundo de la robótica submarina es un campo que se encuentra en constante crecimiento debido al desarrollo tecnológico, de manera que presenta oportunidades de aplicación para la exploración marina, así como tareas de inspección de estructuras subacuáticas. Debido a estos avances, existe la capacidad de operar pequeños vehículos remotamente (ROV) para realizar dichas tareas.

Uno de estos vehículos es el BlueROV2, de la empresa Blue Robotics, siendo este el ROV sumergible que ofrece la mejor relación entre su rendimiento y su coste.

Con la realización de este proyecto se intenta diseñar una escena virtual en la que se pueda desarrollar una tarea de control sobre el vehículo submarino BlueROV2 permitiendo así realizar pruebas en un entorno que se asemeje a la realidad, optimizando así la eficiencia y los recursos al no requerir del ROV físicamente.

Se diseñará un entorno submarino que simule en mayor medida la realidad, así como se realizará un control del movimiento del ROV sobre dicha escena utilizando un controlador PID sobre sus motores.

Todo el desarrollo del proyecto se realizará dentro de una máquina virtual que contiene todos los programas y archivos que se utilizaran, donde destacan programas de diseño como Blender, el software ArduSub y el lenguaje de programación Python.

1.2 OBJETIVOS

El principal propósito de este proyecto consiste en el diseño de un entorno virtual que permita una fiel simulación tanto del ambiente submarino como del vehículo sumergible. De esta manera se podrán demostrar los conocimientos que se han adquirido en los años de carrera. Se desea también ampliar los conocimientos en áreas desconocidas, así como documentar su proceso de forma adecuada.

Otro objetivo claro es la integración de diferentes softwares y herramientas utilizadas en la simulación para aplicar los conocimientos adquiridos en programación, en este caso en lenguaje Python, siendo un apartado fundamental de uso diario en numerosas tareas de desarrollo en el ámbito de la ingeniería.

1.3 ALCANCE

Primero, se establecerán los conocimientos acerca de los vehículos ROV, su funcionamiento y los elementos que contienen, para luego introducir el que se usará en el proyecto. A continuación, se explicarán los diferentes softwares presentes y su implicación individual en el proyecto.

Posteriormente se diseñará el entorno 3D y una interfaz gráfica, mediante lenguaje de programación Python, para la simulación que incluya todos los archivos y ejecutables necesarios, de forma que facilite al usuario las pruebas de simulación.

Para el diseño de la escena 3D se utilizará el motor grafico Panda3D, enlazado a Python, que incluye diferentes funciones y características que permiten simular aplicaciones gráficas en tiempo real de forma estable. El diseño de esta se centra tanto en el apartado grafico de la aplicación como en la actualización constante de la posición y orientación del submarino en el entorno. La escena 3D diseñada incluirá diferentes modelos de objetos que pueden encontrarse en el fondo marino y un modelo del BlueROV2 heavy que simulara al vehículo real.

Se complementará lo anterior con una prueba de simulación del ROV sobre una trayectoria a seguir especificada previamente. El vehículo se cargará en una ubicación específica gracias al ejecutable de ArduSub y se establecerá una conexión de forma que se pueda mantener una comunicación entre el equipo y el ROV ya sea para mandar ordenes o para requerir mensajes de datos del vehículo. Con todo lo anterior y con una correcta conexión, se armará el vehículo y comenzara su movimiento, controlado por los motores que a su vez están regulador por el controlador PID, sobre la trayectoria. Se realizará un seguimiento durante el desarrollo de la simulación hasta que esta finalice correctamente. Paralelamente a este seguimiento se podrá observar el movimiento del ROV sobre la escena diseñada y en el software QGroundControl.

Finalmente, algunos datos del ROV son almacenados en un archivo para su posterior análisis. Se puede elegir diferentes herramientas externas, pero se introduce una propia opción dentro de la interfaz que permite cargar los datos en la plataforma MATLAB donde se mostraran diferentes gráficas analizando los valores obtenidos.

2 VEHÍCULOS SUBMARINOS OPERADOS REMOTAMENTE

2.1 Introducción

Los ROV son vehículos que están controlados por un operador humano que no está físicamente dentro del vehículo. Pueden estar operados por señales de radio o mediante un cable o línea que conecte el vehículo al lugar donde se encuentre el operador humano [1]. La energía y las ordenes se envían desde la superficie a través del cable al ROV.

2.1.1 Componentes de un ROV

La configuración, tamaño y capacidades de los ROV varía enormemente entre los diferentes tipos. Sin embargo, existen ciertas características que todos los vehículos submarinos comparten [2]:

- **Bastidor** (frame): Los vehículos submarinos se encuentran construidos normalmente alrededor de un bastidor que hace la función de "columna vertebral" de los ROV. Sobre dicho bastidor se irán acoplando los demás elementos del vehículo.
- **Propulsores** (thrusters): Los ROV utilizan, normalmente, hélices accionadas por un mecanismo de funcionamiento hidráulico o eléctrico.
- **Cámara**: Actúa como sensor principal del vehículo debido a ser los "ojos" del operador, permitiéndole tener conciencia de todo tipo de situaciones en las que se encuentre el ROV.
- **Capsula electrónica** (EPOD): Este componente tiene como misión traducir y transmitir los comandos recibidos desde la superficie hasta los diferentes actuadores. De forma análoga, debe recabar las señales de los sensores, así como prepararlas para su trasmisión y enviarlas al operador en la superficie.
- **Umbilical**: Se utiliza para la comunicación y suministro de energía.
- **Sensores y actuadores**: La cantidad de sensores y actuadores que se pueden montar en un ROV es amplia. Sin embargo, los más comunes son los manipuladores, los sonares de localización, luces, sistemas de posicionamiento, LIDAR…
- **Sistema de flotabilidad positiva**: Los ROV están diseñados para que su flotabilidad total sea neutra, de forma que los propulsores únicamente actúen para generar desplazamiento y no para contrarrestar la fuerza de la gravedad. Esta se consigue mediante el uso de bloques de plástico o goma espuma.

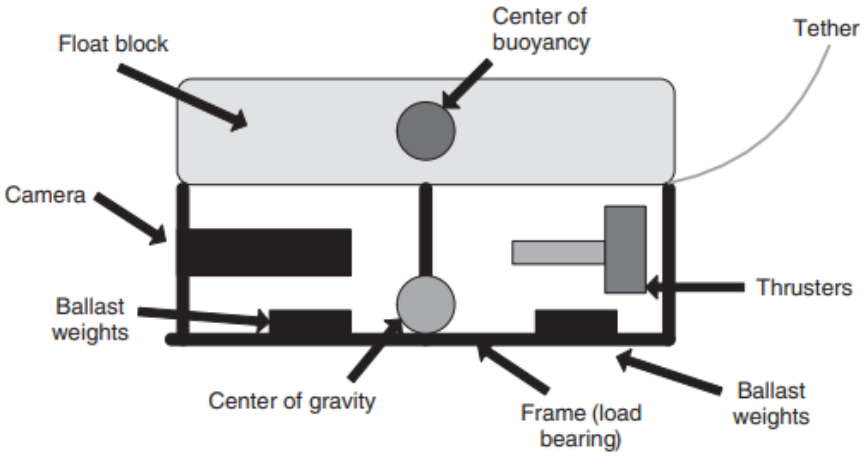


Figura 1: Elementos de un submarino ROV.

2.1.2 Tipos

Los ROV tienen dos tipos de clasificación:

- **Por tamaño**
  - Pequeños: Se encuentran todos los ROV que únicamente poseen de una cámara de video y un par de elementos de iluminación. Su peso máximo no supera los 10 kg, tienen una potencia de 2KW con tres o cuatro motores y una profundidad máxima aproximada de 200 metros (a partir de los 50 metros se dificulta el pilotaje). El uso principal es la observación e inspección de instalaciones o de fondos marinos
  - Medianos: Estos vehículos poseen un peso de entorno a los 100 o 200 kg, poseen varias cámaras de video, varias luces, 6 motores, potencia eléctrica entre 5 y 10 KW y trabajan en una profundidad máxima de 1000 metros. Estos vehículos pueden portar un par de manipuladores para pequeñas tareas. Su principal uso es, al igual que los ROV pequeños, la observación.
- **Por clase**
  - ROV Working Class: En esta clase se encuentran los grandes ROV con accionamiento hidráulico, 150/200 hp y con una profundidad de trabajo que se encuentra en el rango de los 2000-4000 metros. Estos se utilizan principalmente para construcciones submarinas.
  - ROV prototypes: Estos tienen como objetivo alcanzar profundidades extremas con el fin de explorar e investigar el fondo marino.

2.2 Descripción del submarino BlueRov2

Primeramente, se va a describir el ROV que será utilizado. El BlueROV2, desarrollado por BlueRobotics, es un vehículo submarino que puede ser operado de forma remota y que se caracteriza por ser asequible y tener un tamaño compacto, lo que permite multitud de aplicaciones en entornos subacuáticos. Este posee una gran cantidad de sensores, así como una cámara de alta definición que permitirá realizar tareas específicas bajo el agua con gran eficacia [3]. En este proyecto se va a utilizar el BlueROV2 con la configuración “Heavy”, que presenta significantes diferencias con la versión estándar.

Figura 2: BlueROV2 Heavy de BlueRobotics.

La estructura principal del vehículo está construida con aluminio anodizado y plásticos de alta resistencia que protege sus componentes. Los ocho propulsores que posee son el modelo T200 desarrollados por la propia BlueRobotics y tienen una configuración que permite el movimiento en seis grados de libertad; hacia adelante/atrás, arriba/abajo, izquierda/derecha, cabeceo, balanceo y guiñada [4]. Esto permite una mayor maniobrabilidad y estabilidad que la versión estándar. Está equipado con una cámara de alta definición y potentes luces LED que ofrece una imagen clara y detallada del entorno submarino. La capsula electrónica alberga el controlador ArduSub y las baterías de ion-litio. El modelo incluye un cable umbilical y un sistema de flotabilidad ajustable mediante bloques de goma espuma. Además, cuenta con una variedad de sensores integrados, como sensores de profundidad, temperatura y presión, que proporcionan datos ambientales y de navegación en tiempo real, aumentando su capacidad de adaptación en diferentes situaciones.

El BlueROV2 puede ser operado mediante un controlador. Sin embargo, para este proyecto, el submarino debe realizar la tarea de forma autónoma, por lo que se deberá programar el recorrido o ruta que seguirá durante la misma.

2.3 Dinámica del vehículo

El BlueROV2 se mueve en el agua utilizando sus propulsores de forma que le permite realizar movimientos en seis grados de libertad. Dichos movimientos se logran mediante el control diferencial de la velocidad de los propulsores para maniobrar de forma precisa en el entorno subacuático.

2.3.1 Principios de movimiento

Para que el vehículo se mueva eficazmente y de la forma deseada se deben conocer primeramente como afectan dos factores importantes:

- **Propulsión**: Los propulsores son esenciales para generar fuerza y moverse en el agua en la dirección deseada.
- **Estabilidad**: Se intenta que el vehículo mantenga una posición estable y que no realice movimientos indeseados.

2.3.2 Tipos de movimiento

El BlueROV2 puede realizar distintos tipos de movimientos gracias al control diferencial de la velocidad de sus propulsores. Estos movimientos pueden dividirse en dos apartados:

- **Movimientos lineales**
  - Propulsión lateral (Sway): Movimiento lateral del vehículo usando los propulsores laterales generando fuerza en la dirección deseada.
  - Propulsión longitudinal (Surge): Movimiento hacia adelante o hacia atrás utilizando los propulsores frontales y traseros.
  - Propulsión vertical (Heave): Movimiento hacia arriba o hacia abajo utilizando los propulsores verticales.
- **Movimientos rotacionales**
  - Yaw: Rotación del vehículo alrededor de su eje vertical.
  - Pitch: Rotación del vehículo alrededor de su eje transversal.
  - Roll: Rotación del vehículo alrededor de su eje longitudinal.

Para realizar las maniobras de forma precisa en el entorno subacuático se combinan los movimientos mencionados logrando una gran maniobrabilidad. En la siguiente figura se muestra de forma clara los seis grados de libertad del vehículo, tanto los ejes de movimiento como las distintas rotaciones.

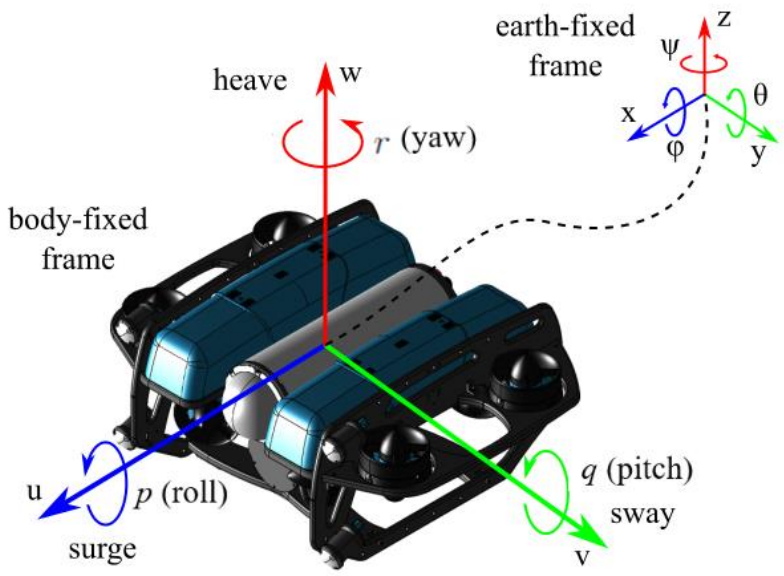


Figura 3: Representación de los seis grados de libertad del BlueROV2 heavy [5].

2.3.3 Fuerzas externas

El ROV puede verse afectado por diferentes fuerzas externas que pueden perturbar su movimiento y afectar directamente a la estabilidad del vehículo:

- **Fuerzas de la corriente**: Las corrientes marinas pueden desplazar al ROV de su trayectoria inicial.
- **Fuerza de arrastre (Drag force)**: Se produce por la resistencia del agua al movimiento del vehículo. Esta fuerza depende del tamaño y superficie del ROV y es proporcional a su velocidad.
- **Flotabilidad**: Es una fuerza ascendente que afecta al ROV debido a tener una densidad mayor que la del agua. El submarino debe estar diseñado para flotar en el agua, donde su peso es soportado por las fuerzas de flotación debidas al desplazamiento del agua por su casco [6].

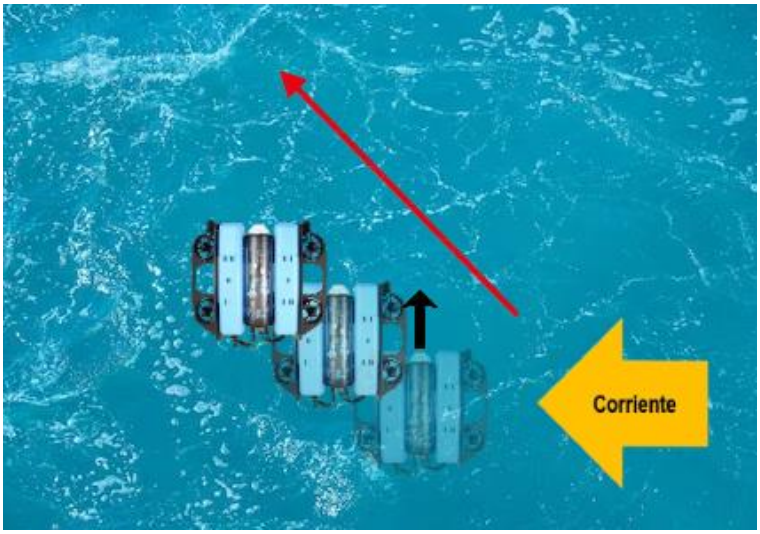


Figura 4: Demostración de como las corrientes afectan al submarino.

2.3.4 Control de movimiento mediante un controlador PID

Teniendo en cuenta lo explicado anteriormente, se va a implementar un regulador PID (Proporcional-Integral-Derivativo) para conseguir un control preciso tanto de la velocidad como la posición del BlueROV2. Con el controlador se va a ajustar la velocidad de los motores de los propulsores para intentar en mayor medida que el vehículo siga la trayectoria especificada.

El controlador PID es un mecanismo de control que permite regular distintas variables mediante un lazo de retroalimentación. El regulador consta de tres parámetros: el proporcional, el integral y el derivativo. La suma de estas acciones se utilizará para ajustar y controlar la velocidad del submarino.

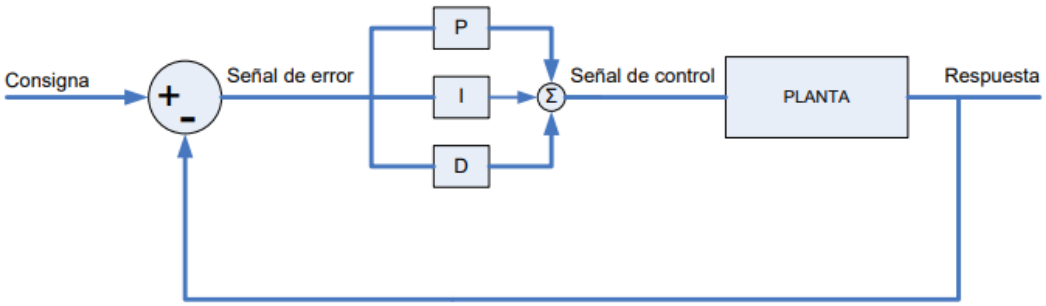


Figura 5: Sistema de control con regulador PID básico [7].

- **Proporcional (P)**: Produce un valor de salida proporcional al valor del error actual. Este reduce el error sin llegar a eliminarlo de forma completa.

- **Integral (I)**: Se ocupa de la acumulación de errores pasados. El control integral actúa cuando se produce una desviación entre el valor de referencia y la variable. Este ayuda a eliminar el error residual que el control proporcional no puede eliminar por sí mismo.
- **Derivativo (D)**: Predice el error futuro teniendo en cuenta su tasa de cambio. Esto permite a la acción derivativa evitar posibles oscilaciones del error y dota la capacidad al controlador de adaptarse a imprevistos del sistema.

Entendido el funcionamiento del regulador, se debe implementarlo siguiendo algunos pasos. En primer lugar, se debe medir el error comparando los valores deseados (SP) con los valores actuales (PV(t)).

$$e(t) = SP - PV(t)$$

Ecuación 1: Calculo del error en función del tiempo.

Después se calcularán las acciones de control siguiendo estas ecuaciones:

- Acción proporcional:

$$P(t) = K_p \times e(t)$$

Ecuación 2: Calculo de la acción proporcional del controlador PID.

- Acción integral:

$$I(t) = K_i \times \int_0^t e(\tau)d\tau$$

Ecuación 3: Calculo de la acción integral del controlador PID.

- Acción derivativa:

$$D(t) = K_d \times \frac{de(t)}{dt}$$

Ecuación 4: Calculo de la acción derivativa del controlador PID.

Tras calcular las acciones estas se sumarán para obtener la salida del controlador PID $u(t)$:

$$u(t) = P(t) + I(t) + D(t)$$

Ecuación 5: Calculo de la salida del controlador PID.

Con esto ya estaría correctamente implementado el regulador PID. A la hora de utilizarlo en el proyecto se deberán tener en cuenta los valores de los parámetros del PID (Kp, Ki y Kd) para obtener los resultados deseados. Para ajustar dichos parámetros se pueden utilizar diferentes técnicas como el método de Ziegler-Nichols. Sin embargo, se usará un método de prueba y error para ajustar los parámetros del controlador.

3 SOFTWARE

Para este proyecto se ha optado por utilizar una máquina virtual en la que se encuentran todos los archivos y programas necesarios para el desarrollo y simulación de este. La máquina virtual esta creada mediante el software de Oracle VM Virtual Box, donde se utiliza una imagen del sistema operativo Linux Ubuntu 20.04. Dentro de esta máquina virtual se emplearán los programas necesarios para el diseño y simulación del entorno, priorizando el uso de programas de software libre.

De esta manera, se logra agrupar todos los archivos necesarios para este proyecto, permitiendo que cualquier persona con acceso a la máquina virtual pueda utilizar estos recursos.

3.1 Máquina virtual

El concepto de máquina virtual puede definirse como equipos virtuales o equipos definidos por software dentro de servidores físicos, donde solo existen como código. Una máquina virtual (Virtual machine en inglés abreviado VM) es una réplica, en cuanto a comportamiento, de un equipo físico, como una PC, teléfono inteligente o un servidor, etc [9].

Una máquina virtual cuenta con su propia CPU, memoria e incluso puede conectarse a internet. Sin embargo, están limitadas debido a que dependen de los recursos asignados en el momento de su creación. La máquina virtual es propiamente una partición del sistema original, de forma que el software contenido en la maquina no se encuentra relacionado con el sistema operativo del equipo en el que se ha creado.

3.1.1 Oracle VM Virtual Box

Oracle VM VirtualBox es un software de virtualización de código abierto que permite crear y gestionar múltiples máquinas virtuales en un único anfitrión físico. En VirtualBox, se pueden ejecutar sistemas operativos diferentes, como Windows, Linux, macOS y otros, simultáneamente en un mismo equipo, lo que proporciona un entorno de pruebas bastante flexible. VirtualBox ofrece la integración del sistema operativo invitado con el sistema anfitrión, soporte para la virtualización anidada, así como la gestión remota a través de interfaces de línea de comandos y gráficas. VirtualBox es una herramienta versátil y eficiente que es bastante utilizada tanto por usuarios como por empresas por su facilidad de uso.

Para este proyecto se utilizará un anfitrión con un sistema operativo Windows en el que estará simulado un sistema operativo de Linux, la versión de Ubuntu 20.04. Una vez instalado el

software de VirtualBox, se debe descargar la imagen del sistema operativo que se va a simular en la propia página del distribuidor [10].

Con la imagen del sistema operativo descargada, debe configurarse la máquina virtual, donde se debe elegir el nombre de esta, el número de procesadores que tendrá a su disposición, espacio de memoria etc. Hay que considerar el tipo de uso que se le va a dar a la máquina virtual a la hora de asignar estos recursos ya que esto afectara al desarrollo de las tareas en el sistema y, por tanto, deberá procurarse en mayor medida de que estas se realicen de forma eficiente.

Una vez que configurada la máquina virtual, se podrá ver la interfaz como en la siguiente figura en la que se podrá elegir entre las distintas máquinas virtuales que se hayan creado. A continuación, se elige la maquina llamada Ubuntu 20.04 y de esta manera se ejecutará de forma correcta en el sistema.

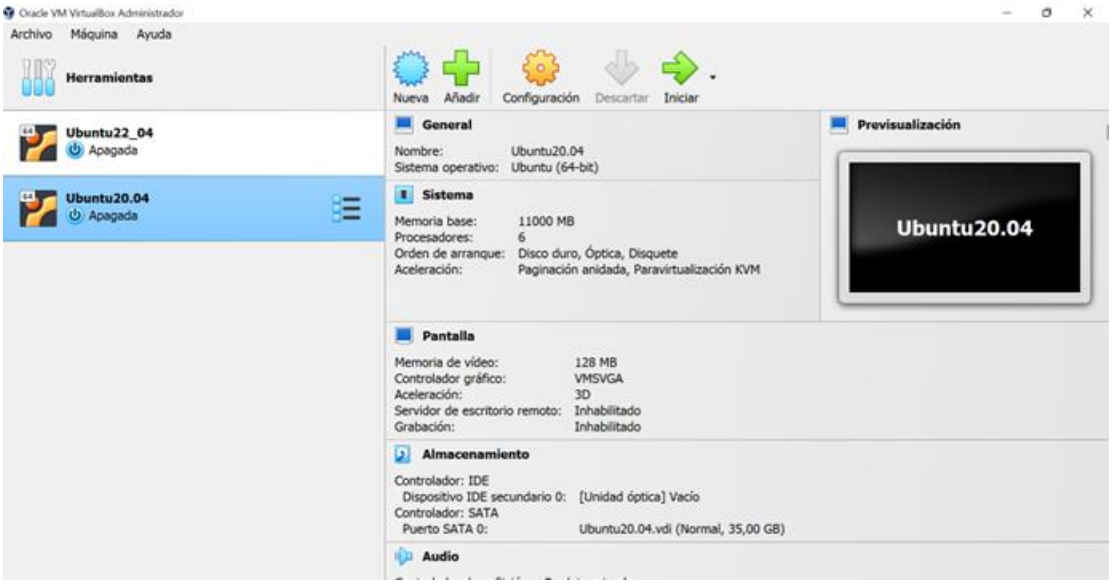


Figura 6: Interfaz de VirtualBox

Al ejecutar la máquina virtual seleccionada por primera vez, se instalará el sistema operativo Ubuntu y realizada la configuración inicial luce como se aprecia en la siguiente figura.

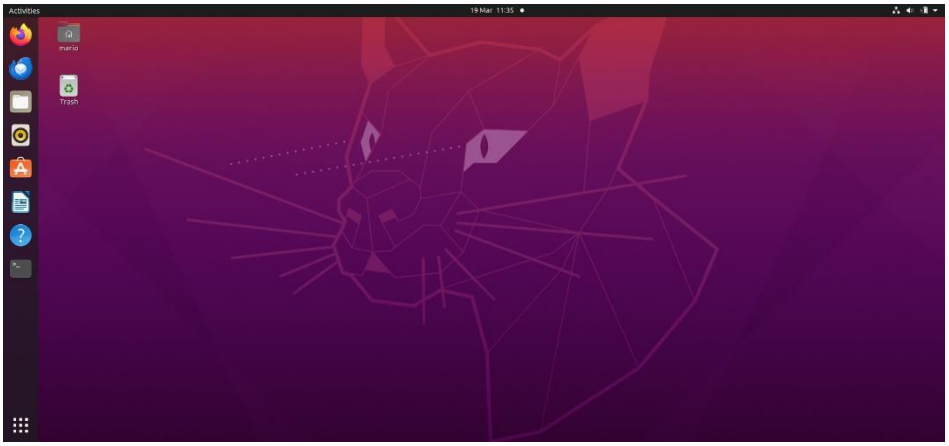


Figura 7: Interfaz de la versión de Ubuntu 20.04

Con la máquina virtual correctamente configurada, el entorno de trabajo está listo para empezar a instalar y trabajar con los demás softwares necesarios.

3.2 ArduSub

ArduSub es una solución de código abierto con todas las funciones para vehículos submarinos operados remotamente (ROV) y vehículos submarinos autónomos (AUV). ArduSub es parte del proyecto ArduPilot y originalmente se derivó del código ArduCopter. ArduSub tiene amplias capacidades listas para usar que incluyen control de estabilidad por retroalimentación, control de profundidad y rumbo y navegación autónoma [11].

Ardusub es una derivación especializada de ArduPilot, diseñado específicamente para el control y la operación de vehículos submarinos no tripulados (ROV). Esta nos ofrece una amplia variedad de aplicaciones en entornos submarinos, desde la exploración y la investigación oceanográfica hasta la inspección de infraestructuras marinas y la búsqueda y rescate. Gracias a su integración con hardware personalizable y software de control avanzado, Ardusub permite a los usuarios construir y operar ROVs con capacidades de control manual y autónomo.

3.2.1 SITL

El SITL (Software en bucle, en español) permite la compilación de código necesario sin necesidad de ningún elemento hardware adicional. Esto es de gran utilidad a la hora de realizar pruebas, depurar errores o validar funcionalidades de forma eficiente y sin gran esfuerzo económico, ya que se pueden simular diversos escenarios sin poner en riesgo el hardware real en situaciones complejas. En el proyecto se podrá ejecutar el archivo de

ArduSub directamente desde el propio PC, y por tanto avanzar en el desarrollo del proyecto sin tener que recurrir a una prueba física con el vehículo ROV.

3.3 MAVLink

MAVLink o Micro Air Vehicle Link es un protocolo de comunicación principalmente utilizado en vehículos aéreos no tripulado (UAVs), pero también aplicable a vehículos ROVs. Este protocolo permite la comunicación entre la estación de control en tierra y el vehículo, facilitando el intercambio de datos, telemetrías, comandos y mensajes en tiempo real.

MAVLink es crucial para la integración y operación de la plataforma ArduSub, ya que estandariza la comunicación y permite la correcta compatibilidad entre diferentes softwares y componentes.

3.4 QGroundControl

QGroundControl es un software código abierto diseñada para ser utilizada con vehículos aéreos no tripulados (UAVs), vehículos submarinos no tripulados (ROVs) y otros vehículos robóticos autónomos. Esta aplicación proporciona una interfaz de usuario intuitiva que permite a los operadores controlar y monitorear los vehículos de forma remota. QGroundControl es compatible con una amplia gama de plataformas y sistemas de vehículos. Entre sus características destacadas se incluyen la planificación de misiones, la telemetría en tiempo real, la configuración del vehículo, la visualización de datos de sensores y la grabación de datos de vuelo para análisis posterior. Además, QGroundControl es altamente personalizable y admite la integración con herramientas de terceros, lo que lo convierte en una herramienta versátil para una variedad de aplicaciones, desde la investigación y la exploración hasta la agricultura de precisión y la vigilancia.

QGroundControl es compatible con el vehículo BlueRov2, que utiliza la plataforma de vuelo ArduSub, por lo que es indispensable a la hora de realizar una simulación con el vehículo.

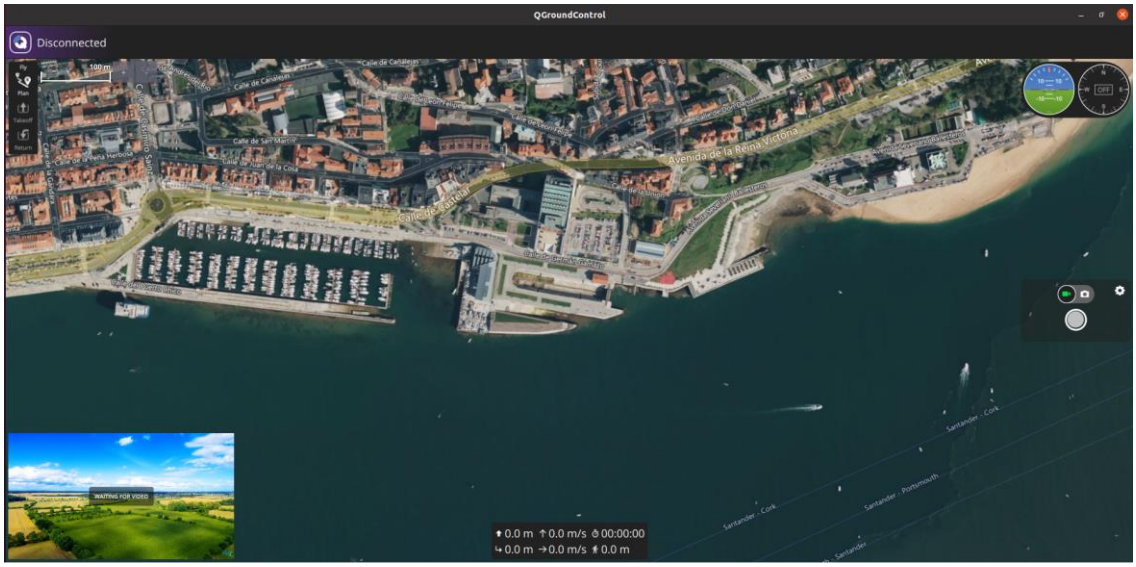


Figura 8: Interfaz de QGrounControl.

3.5 Visual Studio Code

Visual Studio Code es un editor de código fuente desarrollado por Microsoft. Este está construido sobre el framework Electron y es compatible con una amplia gama de lenguajes de programación.

Este editor va a ser fundamental durante el desarrollo del proyecto ya que a la hora de programar en Python la escena 3D y scripts relacionados con ella se va a poder trabajar en un espacio que recopila todos estos archivos, implicando una mejora de la eficiencia a la hora de programar los códigos y mayor facilidad a la hora de compilarlos sin necesidad de acceder a una terminal externa.

3.6 Python

Python es un lenguaje de programación interpretado, orientado a objetos y de alto nivel con semántica dinámica. Sus estructuras de datos integradas de alto nivel, combinadas con escritura y enlace dinámicos, lo hacen muy atractivo para el desarrollo rápido de aplicaciones, así como para su uso como lenguaje scripting o de unión entre componentes existentes. La sintaxis simple y fácil de aprender de Python enfatiza la legibilidad y, por lo tanto, reduce el coste de mantenimiento del programa. Python admite módulos y paquetes, lo que fomenta el modularidad del programa y la reutilización del código. El intérprete de Python y la extensa biblioteca estándar están disponibles en formato de código fuente o binario de forma gratuita para todas las plataformas principales y pueden distribuirse gratuitamente [12].

Durante el proyecto se va a realizar programación en Python, incluyendo estos códigos algunas librerías auxiliares que permiten realizar tareas concretas.

3.7 Qt Designer

Es una herramienta que permite diseñar y crear interfaces gráficas de usuario (GUI) mediante el uso de bibliotecas Qt. Los widgets son los elementos principales y pueden mostrar datos, información de estado, recibir inputs del usuario etc.

Esta plataforma va a servir de ayuda para crear una interfaz de usuario que recoja todos los archivos necesarios para la simulación del ROV, en el que se usaran cuadros de texto para especificar la ruta de los ejecutables, botones y demás elementos.

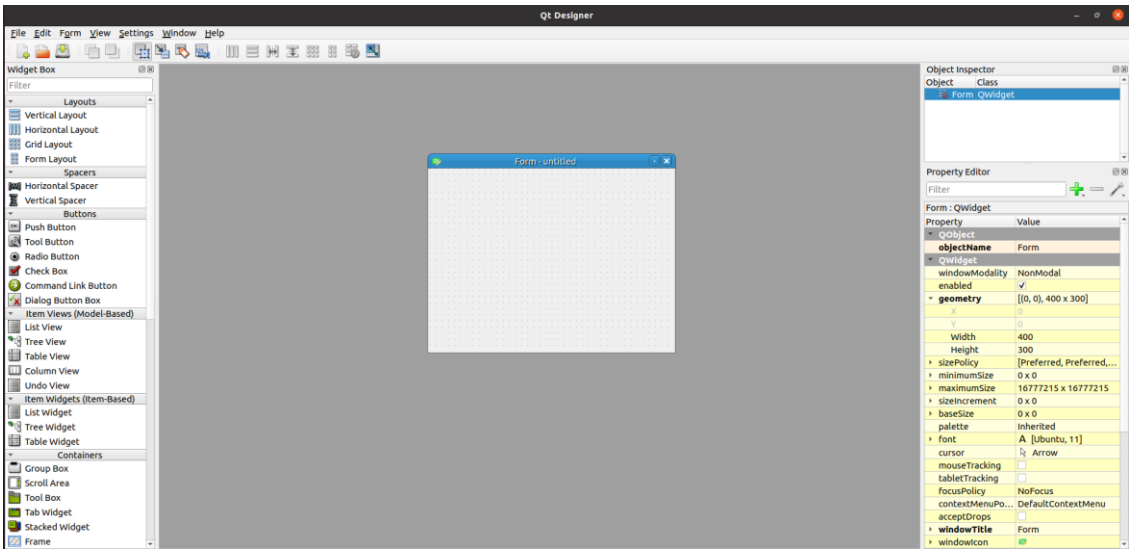


Figura 9: Interfaz de Qt Designer.

3.8 Blender

Blender es una herramienta de creación de elementos 3D de código abierto, dedicada especialmente al modelado, montaje, simulación, renderizado, iluminación y animación de un objeto 3D entre otros aspectos. Este software es ampliamente utilizado por artistas digitales y desarrolladores de videojuegos debido a su versatilidad y variedad de herramientas.

Con Blender se van a poder crear y modificar diferentes modelos que serán utilizado en la escena submarina. Para su instalación en el sistema operativo Linux con la versión correspondiente bastara con usar el comando “sudo apt install blender” en una terminal.

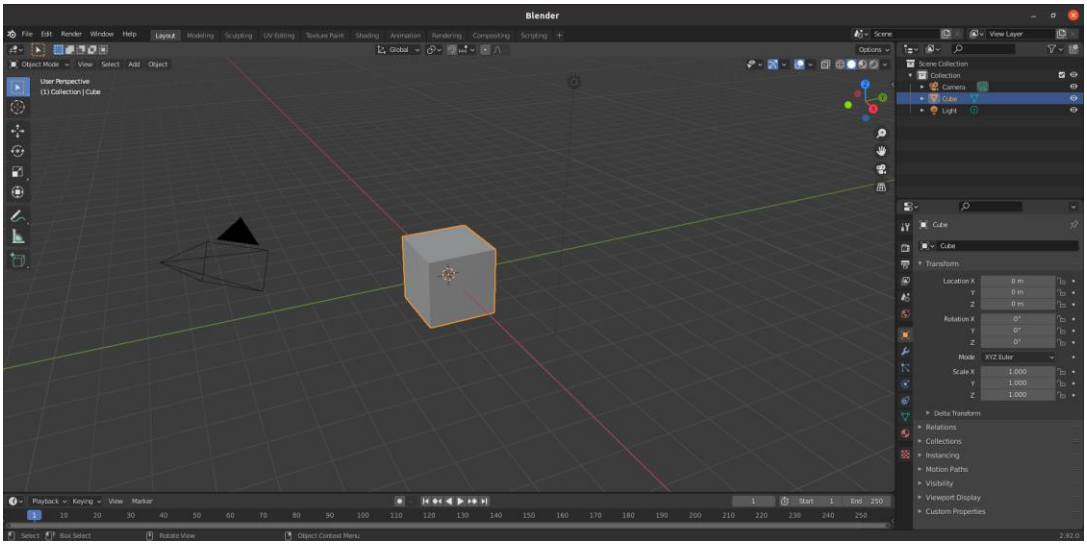


Figura 10: Interfaz de Blender.

3.9 MATLAB

MATLAB, de la abreviatura de “MATriz LABoratory”, es una plataforma de programación diseñada principalmente para el cálculo numérico utilizada por ingenieros y científicos para analizar, desarrollar algoritmos y crear modelos.

Las aplicaciones desarrolladas en MATLAB usan su propio lenguaje de programación, que puede ejecutarse en el propio entorno o a través de unos archivos con formato ‘.m’. Esta plataforma posee gran cantidad de librerías (Toolboxes) ya instaladas, así como otras que podrían descargarse desde la propia plataforma.

MATLAB va a ser utilizada en este proyecto como herramienta de análisis de los datos obtenidos, que se pasarán al archivo *.m programado, de forma que se podrán representar una serie de gráficas con los valores y en consecuencia dar pie a su análisis e interpretación.

4 PROGRAMACIÓN DE CODIGOS Y SCRIPTS

En esta sección se va a explicar el desarrollo del entorno 3D en el que el vehículo ROV realizara las simulaciones, así como otros códigos secundarios. Este entorno esta creado en código Python usando la librería Panda3D de manera que se podrá desarrollar un “mapa” con físicas realistas y numerosas funciones tanto a nivel grafico como la configuración de distintos elementos que serán de ayuda.

Primeramente, se van a explicar las librerías que incluyen la conexión con el BlueROV2 y que incluyen algunas funciones menores que serán aplicadas.

4.1 Comunicación con el vehículo y funciones adicionales

4.1.1 Conexión MAVLink

Previo al diseño de la escena, se deben configurar algunas librerías con distintas funciones que serán utilizadas en el código principal. La librería ‘MAVLINK_lib’ se utiliza para realizar una conexión ‘MAVLink’ con el BlueROV2, de forma que se puedan realizar diferentes tareas sobre el vehículo como armar/desarmar, modificar el modo de funcionamiento o solicitar al vehículo información necesaria.

Destacando algunas funciones del código, al principio de este se toma la dirección y puertos especificados para posteriormente crear la conexión MAVLink llamando a la función ‘create_connection’. En un primer momento el vehículo se establece en modo ‘stabiize’, de forma que estabiliza el balanceo y mantendrá la dirección mientras no se le ordene girar, desarmado y con los propulsores en un punto neutro para que permanezca estacionario.

La función ‘set_arm_disarm’ como su nombre indica armara o desarmara el vehículo dependiendo del valor booleano que se le proporcione teniendo en cuenta que con un ‘0’ se desarma el vehículo y con un ‘1’ se procede a armarlo. Dependiendo del modo que se requiera en el vehículo se llamara a la funcion ‘set_mode’ para establecer el modo de funcionamiento estando disponibles los siguientes:

- **ALT_HOLD**: Mantiene la altitud.
- **POSHOLD**: Permite mantener la posición y altitud actuales utilizando los sensores.
- **STABILIZE**: Estabiliza el cabeceo y balanceo, manteniendo la dirección estática.
- **MANUAL**: Proporciona control total al operador.
- **AUTO**: Ejecuta una misión programada.
- **GUIDED**: Permite controlar el vehículo mediante el uso de comandos.
- **CIRCLE**: El vehículo se mueve en círculos alrededor de un punto definido.

- **SURFACE**: Mantiene el vehículo en la superficie del agua.
- **ACRO**: Proporciona control avanzado para maniobras acrobáticas.

Si se requiere información del BlueROV2 se llamará a la función 'get_info' en la que se podrá solicitar un mensaje especifico o toda la información que esté disponible, así como se podrá controlar el envío de mensajes MAVLink mediante 'request_message_interval' en el que se da el id del mensaje que se desea solicitar información con una frecuencia específica relacionada. Con esto se consigue reducir la congestión de datos en sistemas que tengan recursos limitados y se requiere un tratamiento de los datos optimizado.

Al final del código se incluye la función encargada de cortar la conexión MAVLink con el vehículo en el caso de que se haya finalizado la tarea correspondiente o se necesite acabar la conexión por algún otro motivo.

4.1.2 Librería con funciones variadas

En la programación del código 'main' se intenta incluir el mínimo de funciones que requieran su presencia estricta en ese código por lo que se crea un script llamado 'utils_lib' en el que se incluyen algunas funciones que realizan transformaciones básicas o demás operaciones que por optimización del código se incluyen en una librería.

Se incluye la clase 'PID_control' implementando un controlador PID en el que se inicializan los coeficientes del controlador, así como el tiempo actual. La función 'update' se encarga de actualizar el valor del error actual, así como el tiempo y 'reset' reinicia los valores y restablece al controlador a su estado inicial.

Otras funciones incluidas son la de 'kml2list', que permite leer archivos en formato 'kml' que contienen coordenadas geográficas para convertirlas en listas de coordenadas que contengan los valores de latitud, longitud y altitud, y 'geodetic_to_geocentric' que servirá de ayuda al poder transformar las coordenadas geodésicas en coordenadas cartesianas.

- ✓ Estas funciones son muy necesarias a la hora de realizar el seguimiento del vehículo ya que permite controlar su movimiento (orientación, movimiento lateral, velocidad…) utilizando los coeficientes del PID o en el momento de obtener las coordenadas geográficas de una ruta con formato 'kml' y facilitar el seguimiento de esta.

4.2 Entorno3D

El archivo Python que incluye la simulación 3D se llama “3Dsimulation” y parte de una clase “Myapp” que hereda las funciones de “showbase”. Esto permite acceder a diferentes métodos que facilitan la gestión de distintos eventos, la carga de modelos 3D propios de la librería o los que se han descargado, así como la configuración de la ventana gráfica que visualiza el usuario.

Al principio del código se establecen los valores de algunos parámetros que serán utilizados más adelante como el tiempo de muestreo (“ts”), el factor de escala y la profundidad del agua. Se establecen también los factores que definen la iluminación ambiental del entorno y la configuración de la niebla, siendo estas dos últimas propiedades propias de Panda3D.

```
class MyApp(ShowBase):
    def __init__(self):
        ShowBase.__init__(self)
        random.seed()

        self.ts = 1.0/60
        xy_scale = 1 # factor de escala xy (escala 1 = 100 metros)
        water_depth = 28

        base.disableMouse()

        color_water=(0.1, 0.25,0.4,1)
        color_luz = (0.5, 0.5, 1, 1)

        base.setBackgroundColor(color_water)

        #setup
        size=512 # tamaño del buffer del agua
        self.update_speed=1.0/60.0 #wave rate (1/60=60fps)

        # niebla
        niebla = Fog("niebla")
        niebla.setColor(color_luz)
        #niebla.setExpDensity(0.01)
        niebla.setExpDensity(0.04)
        #Aplicar niebla
        self.render.setFog(niebla)

        # crear iluminacion
        # luz direccional
        dlight = DirectionalLight("luz direccional")
        dlight.setColor(color_luz)
        dlight.setDirection((0, 45, -45))
        self.dlnp = self.render.attachNewNode(dlight)
        # luz ambiental
        alight = AmbientLight("luz ambiental")
        alight.setColor(color_luz)
        self.alnp = self.render.attachNewNode(alight)
```

El tiempo de muestreo depende directamente de la tasa de fotogramas por segundo que se ha asignado a la hora de crear la ventana principal por lo que al establecer un valor de 60 FPS, por lo que se realizaran 60 muestreos por segundo, correspondiendo con un ts de 1/60.

Para elegir el color del background que se usara como color de la superficie del agua, se ha utilizado una herramienta de visión de colores RGB. A la hora de especificar el color en el código deben darse valores que van desde 0 a 1 por lo que, si se conoce el código del color, se podría pasar directamente a ese formato. El color elegido tiene el código #1a4066 y es un tono de azul que representa el color del mar.

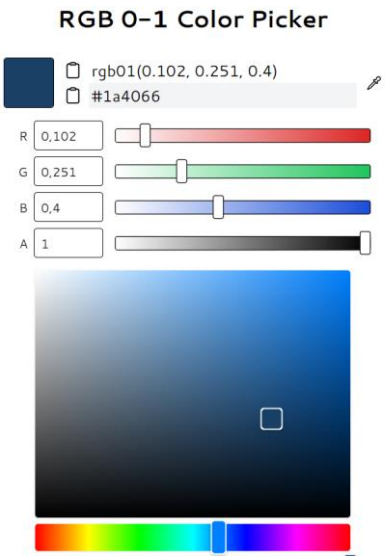


Figura 11: Configuración de colores RGB mediante herramienta gráfica.

La niebla es un elemento que posee de manera predefinida Panda3D por lo que únicamente se deberá configurar a gusto con lo deseado. Para ello, se carga el color de la luz que se había establecido anteriormente y se asignara una densidad con valor de 0.04.

- ✓ Con la niebla se consigue dar una visión realista del fondo marino en el que la visibilidad disminuye al descender debido a la absorción y dispersión de la luz.

Lo mismo ocurre con la iluminación, la plataforma incluye una función de luz direccional y ambiental en la que únicamente se configuran a las características necesitadas. Estos aspectos son meramente estéticos por lo que se podrían modificar dependiendo de lo que se desea. Para lo que se busca en este proyecto, bastara con la configuración explicada.

A continuación, se va a proceder cargando diversos modelos de texturas y objetos que se incluirán en el código.

Debido a que se está trabajando con un entorno que simula un fondo marino, se van a incluir algunos modelos de rocas con un formato ‘.glb’. Este formato se utiliza para almacenar modelos 3D, así como toda la información asociada a él (texturas, materiales, geometría…) y se diferencia de otros como ‘.obj’ ya que al ser un formato binario almacena la información en un único archivo.

Algunos de estos modelos, utilizados como base, se han descargado procedentes de algunas páginas web que ofrecen modelos de manera abierta como el modelo ‘.glb’ de “rock2”. Si se importa este modelo a un programa de animación como Blender, se podrá modificar y visualizar el modelo de forma que se adapte a la necesidad que se está buscando.

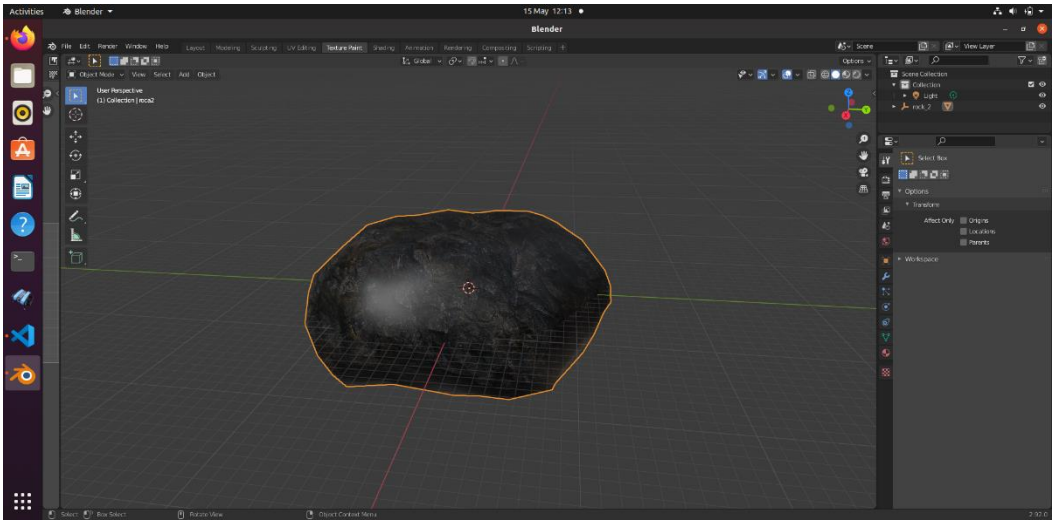


Figura 12: Modelo de "roca_2" vista en Blender.

Una vez que se tiene ya el modelo, ya sea el original o con las modificaciones oportunas, la vista final en el programa sería similar a la que se muestra en la siguiente figura.

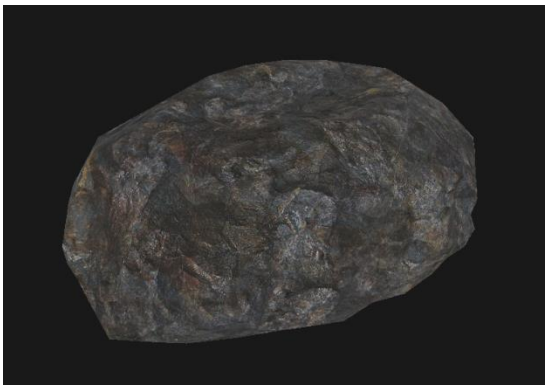


Figura 13: Modelo de "roca_2".

Después de obtener todos los modelos de las rocas que se van a añadir a el código, se van a cargar su modelo, en este caso de ejemplo se añade el modelo de "rock1", como un nodo hijo en relación con el nodo pariente, que sería el de "fondo", utilizando el comando "reparentTo". En este caso "fondo" actúa como el nodo padre por lo que todos los modelos de roca están vinculados directamente a él. Esta estructura facilita la gestión de dichos modelos ya que cualquier modificación o transformación que sufra el nodo "fondo" afectara a los nodos hijos, es decir, las rocas.

```
# cargar rocas
rock1 = self.loader.load_model("assets/roca1.glb")
rock1.reparentTo(fondo)
rock1.clearModelNodes()
rock1.flattenStrong()
rock1.setPos(0, 0, 0)
rock1.setScale(1, 1, 0.5)
rock1.setLight(self.dlnp)
rock1.setLight(self.alnp)
rock1.showThrough()
for x in range(-50*xy_scale, 50*xy_scale, 5):
    for y in range(-50*xy_scale, 50*xy_scale, 5):
        nueva_roca_nodo = fondo.attachNewNode("new-block-placeholder")
        escala = (1+2*random.random(), 1+2*random.random(), 0.5)
        nueva_roca_nodo.setScale(escala)
        posx = x+10*(random.random()-0.5)
        posy = y+10*(random.random()-0.5)
        posz = 0
        nueva_roca_nodo.setPos(posx, posy, posz)
        nueva_roca_nodo.setHpr(random.random()*360, 0, 0)
        nueva_roca_nodo.setLight(self.dlnp)
        nueva_roca_nodo.setLight(self.alnp)
        rock1.instanceTo(nueva_roca_nodo)
```

Para cada roca añadida al código, se le especifica una escala (dependiendo del modelo usado) y se le asigna la iluminación tanto direccional (‘dlnp’) como ambiental (‘alnp’). Debido a que se intenta conseguir un fondo fiel a la realidad, se va a utilizar un bucle de forma que las rocas cargadas sigan una distribución aleatoria en múltiples instancias dentro de un rango determinado. Con este método se consigue un fondo más dinámico sin la necesidad de recurrir a cientos de modelos de rocas diferentes.

Con los modelos que definen el ‘fondo’ submarino ya cargados y configurados, se pasara a cargar el modelo de ROV a utilizar, el submarino BlueROV2 con el kit de configuración ‘Heavy’, utilizando de nuevo un modelo con formato ‘.glb’. El modelo original se obtuvo de una web que ofrece modelos 3D de manera gratuita [13]. Sin embargo, este es demasiado complejo para su utilidad en el proyecto por lo que se deben realizar algunos cambios. Se ha exportado el modelo original, con formato ‘.stl’, al software Blender y se le ha eliminado algunos polígonos o formas que son demasiado realistas y precisas que exigirían un alto rendimiento del equipo. Obteniendo el número de polígonos del modelo en formato ‘.stl’ mediante la ayuda de la librería ‘numpy-stl’ se ha podido comprobar que está compuesto por un total de 6.177.690 polígonos, mientras que si se comprueba el número de polígonos del modelo final utilizando la biblioteca llamada ‘pygltflib’, se obtienen 977.200 polígonos.

- ✓ El objetivo principal era utilizar un modelo del BlueROV2 no demasiado complejo, por lo que tras comprobar que las modificaciones representan una reducción del 84% del

número de polígonos respecto al modelo original, se puede concluir que el modelo cumple con la condición.

- ✓ Al utilizar este modelo frente al original se consigue una mayor optimización del rendimiento en el entorno 3D, especialmente si se utiliza en dispositivos con recursos más reducidos.

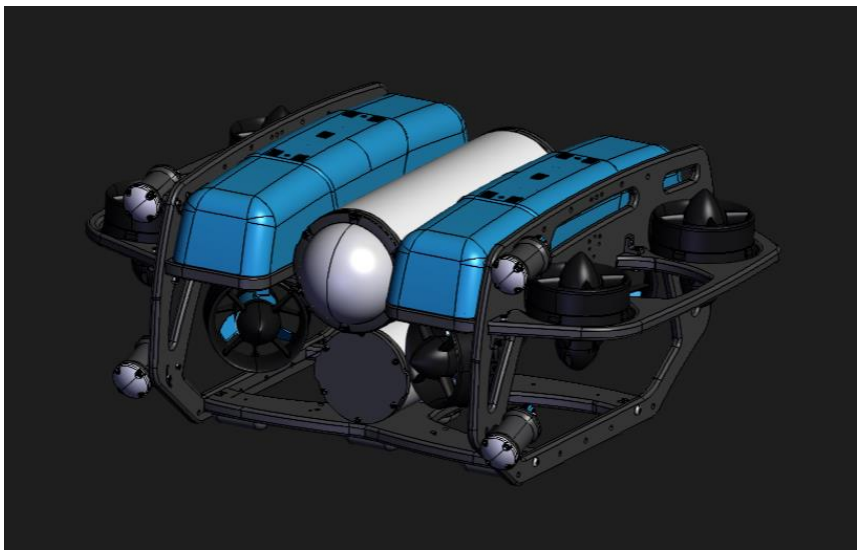


Figura 14: Modelo original del BlueROV2 heavy

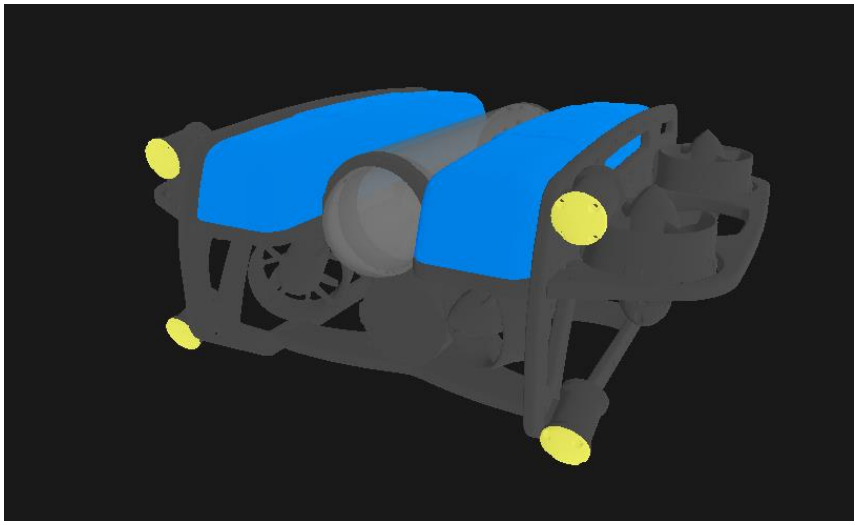


Figura 15: Modelo final del BlueROV2 heavy

El modelo final se exporta a un modelo '.glb' llamado 'bluerov23.glb' siendo este el modelo que aparecerá en el código. En el momento que se carga el modelo en el código, únicamente se van a modificar algunos aspectos como su orientación, la iluminación y la posición del objeto en el entorno. El valor que se indique en la coordenada 'z' en esta variable configura la altitud inicial que tendrá el BlueROV2 al momento de ejecutarse el código.

```
# cargar bluerov
self.bluerov = self.loader.loadModel("assets/bluerov23.glb")
self.bluerov.clearModelNodes()  # simplificar mesh
self.bluerov.flattenStrong()  # simplificar mesh
self.bluerov.reparentTo(self.render)

self.bluerov.setHpr(90, 90, 0)
self.bluerov.setPos(0, 0, 29)
self.bluerov.setScale(1)
self.bluerov.setLight(self.dlnp)
self.bluerov.setLight(self.alnp)
```

Con el fondo y el submarino ya configurados, se va a cargar en el entorno una ‘skybox’ de forma que de un aspecto de un entorno envolvente. Este consiste en una textura que cubre de forma completa el entorno y que consta de la apariencia tanto del cielo y del horizonte. La ‘skybox’ al tener forma de cubo tendrá seis texturas diferentes asociadas entre las que se encuentran:

- **Norte, Sur, Este y Oeste**: Estas texturas representan las distintas direcciones cardinales.
- **Arriba y Abajo:** La textura superior representara la vista de la parte más alta del cubo mientras que la textura inferior representara el fondo marítimo.

Si se juntan todas las texturas en el cubo, se obtendrá la ‘skybox’ que cerrará el entorno virtual. Dependiendo del punto de vista, únicamente serán apreciables al mismo tiempo cinco de las seis caras que componen el cubo.

A la hora de elegir las texturas que componen el cubo se ha de tener en cuenta el entorno con el que se está trabajando. Al trabajar con un submarino, esta ‘skybox’ se centra en simular un entorno marítimo por lo que se incluye un horizonte que simula la superficie marítima, así como el fondo submarino (que no debe confundirse con la textura “fondo” creada anteriormente).

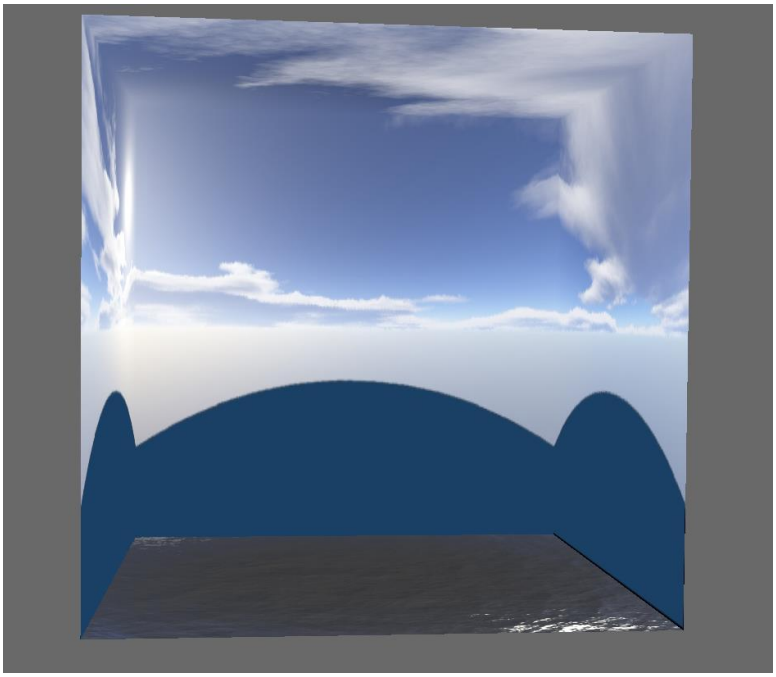


Figura 16: Representación de la Skybox añadida.

Lo único que se ha de tener en cuenta a la hora de añadir el modelo de la 'skybox' al código es la posición y escala de esta, debido a que debe acoplarse correctamente a los modelos que ya estaban configurados.

```
#skybox
skybox=loader.loadModel('skybox/skybox')
skybox.reparentTo(render)
skybox.setPos(0, 0, 40)
skybox.setScale(50*xy_scale)
```

Por último, se ha incluido en el código una textura de la superficie marítima creada en formato GLSL (Lenguaje de sombreado de gráficos) de forma que representa la animación del oleaje [14].

Los archivos con dicho formato suelen ser utilizados para conformar y controlar el aspecto y comportamiento visual de las texturas gráficas en tiempo real.

- ✓ Incluir esta textura que simula la superficie marítima aporta realismo a la escena creada, así como dinamismo y estética dado que se podrá interactuar también con la iluminación sobre esta textura.

En este punto ya se han añadido y configurado todos los modelos gráficos que componen el entorno por lo que se va a representar en la siguiente figura como luce la escena en una visualización básica.

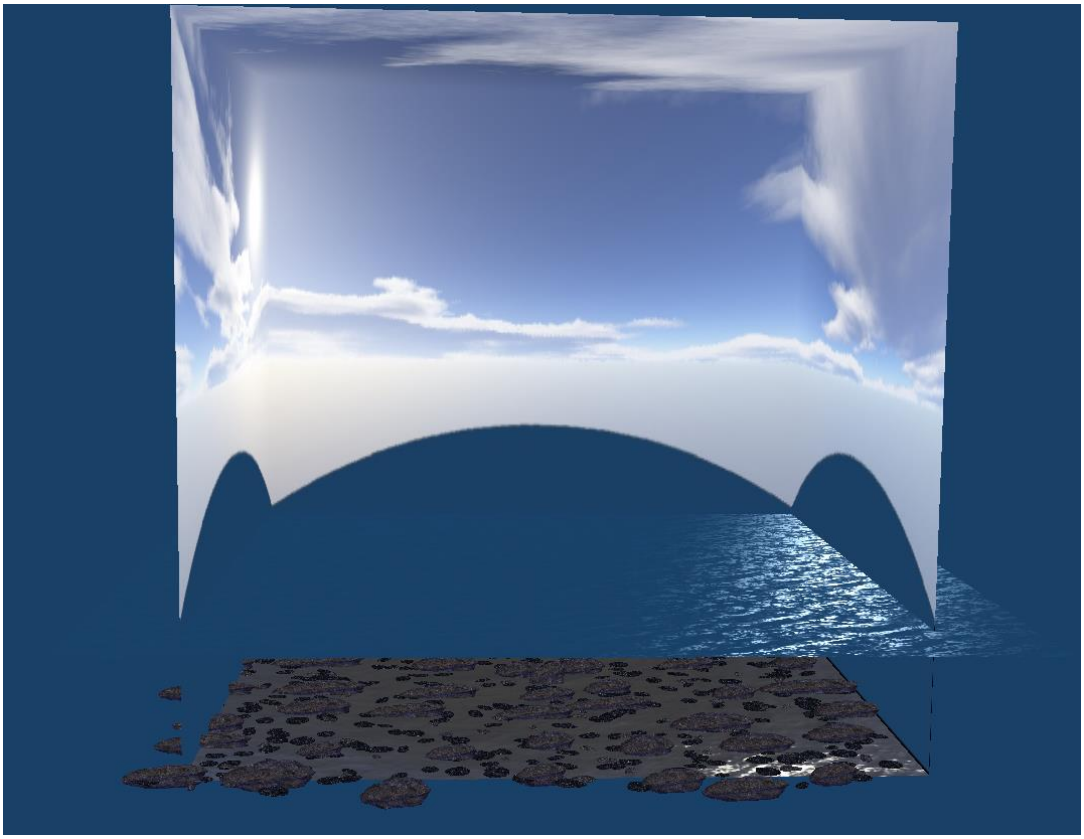


Figura 17: Resultado final de los elementos de la escena (niebla desactivada)

La próxima tarea por completar en el código seria establecer la conexión con el BlueROV2 y solicitar diferente información que se necesitara más adelante. Primeramente, se establece una conexión MAVLink utilizando el contenido en el código 'mav_link_lib' a través de un protocolo UDP en el puerto '14552'. Con la conexión ya creada se va a solicitar información en forma de mensaje con el nombre 'AHRS2' (Attitude and Heading Reference System 2) y 'SYSTEM_TIME' en intervalos regulares usando el periodo 'self.ts'. En el mensaje 'AHRS2' se incluyen datos sobre el ángulo de escora (roll), ángulo de inclinación (pitch), ángulo de deriva (yaw), latitud y longitud en coordenadas geodésicas y la altitud del vehículo.

```
# conexion
self.conexion = mavlink_lib.MAVlink_connection('udpin:localhost:14552')
# self.conexion.request_message_interval("LOCAL_POSITION_NED", self.ts)
self.conexion.request_message_interval('AHRS2', self.ts)
self.conexion.request_message_interval('SYSTEM_TIME', self.ts)
self.actual_time = self.conexion.get_info('SYSTEM_TIME', bloqueo=True).time_boot_ms*1e-3
```

Los datos que se han recopilado son necesarios para, principalmente, ubicar el vehículo en la escena en un instante inicial y posteriormente ir actualizando su posición convenientemente. Para este primera opción, se toman los datos proporcionados por

'AHRS2' relativos a la posición y se asignan a la variable 'self.actual_pos', así como los datos que representan en la orientación se guardan en la variable 'self.actual_orient'.

```
# posicion y orientacion inicial
ahrs2 = self.conexion.get_info('AHRS2', bloqueo = True)
pos = utils_lib.geodetic_to_geocentric(ahrs2.lat*1e-7, ahrs2.lng*1e-7, ahrs2.altitude)
self.actual_pos = Vec3(pos[0], pos[1], pos[2])
self.actual_orient = Vec3(ahrs2.yaw, ahrs2.pitch, ahrs2.roll)*180/pi
self.bluerov.setPos(Vec3(0, 0, ahrs2.altitude))
self.bluerov.setHpr(Vec3(0,0, 0))
```

La posición del BlueROV2 en la escena debe actualizarse constantemente para que sea una fiel representación del movimiento de este por lo que se crea una función denominada 'update' que servirá para poder reubicar al vehículo en cada periodo.

Primeramente, se obtiene el tiempo actual ('actual_time') del sistema y se calcula el tiempo que ha transcurrido desde la última actualización. Si se recibe nueva información (mensaje 'SYSTEM_TIME'), se actualiza el tiempo actual ('self.actual_time') y se calcula el intervalo entre los muestreos ('sampling_time'). Actualizado el tiempo, se recoge la información del mensaje 'AHRS2' en una variable de forma que, si hay nueva información dada por este mensaje, se actualiza la orientación del vehículo ('orient'). La información que proporciona este sensor acerca de la orientación viene dada en radianes por lo que se convierte a grados y se le pasan estos valores al BlueROV2. Para la posición del ROV, se establece la actual posición registrada como 'last_pos' y se transforman los valores obtenidos por 'ahrs2' de la latitud, longitud y altitud utilizando la función contenida en el archivo 'utils_lib'. Con los valores ya en coordenadas geocéntricas, se actualiza la posición del vehículo ('self.actual_pos').

```
# funcion para actualizar escena
def update(self, task):
    dt = globalClock.getDt()

    # Obtener el tiempo actual
    last_time = self.actual_time
    actual_time = self.conexion.get_info('SYSTEM_TIME', bloqueo=False)
    if actual_time: #si hay nuevos datos
        self.actual_time = actual_time.time_boot_ms * 1e-3
        sampling_time = self.actual_time - last_time
        if sampling_time > self.ts + 0.1:
            self.conexion.request_message_interval('AHRS2', self.ts)
            self.conexion.request_message_interval('SYSTEM_TIME', self.ts)

    # Actualizar la orientación
    ahrs2 = self.conexion.get_info('AHRS2', bloqueo=False)

    if ahrs2: #si hay nuevos datos
        orient = Vec3(-ahrs2.yaw + pi * 0.5, ahrs2.pitch + pi * 0.5, ahrs2.roll) * 180 / pi #Radianes->grados
        self.bluerov.setHpr(orient)

        # Actualizar la posición
        last_pos = self.actual_pos
        actual_pos = utils_lib.geodetic_to_geocentric(ahrs2.lat * 1e-7, ahrs2.lng * 1e-7, ahrs2.altitude) # coordenadas geodesicas a geocentricas
        self.actual_pos = Vec3(actual_pos[0], actual_pos[1], actual_pos[2])
        delta_pos = last_pos - self.actual_pos
        new_pos = self.bluerov.getPos() + delta_pos # dependiendo de la diferencia entre la ultima posicion se calcula la nueva
        new_pos.z = ahrs2.altitude
        self.bluerov.setPos(new_pos)


    # Actualizar la posición de la cámara
    self.camera.setPos(self.bluerov.getPos() + Vec3(-6, 0, 3))
    self.camera.lookAt(self.bluerov)
    return Task.cont
```

A continuación, se debe configurar la cámara de la escena para poder apreciar su movimiento en el entorno por lo que se configura la cámara para estar en una posición específica

utilizando la posición del BlueROV2. Se crea un vector en el que se situara la cámara seis unidades en el eje x alejado del vehículo y tres unidades en el eje z hacia arriba. Panda3D también ofrece la opción 'lookAt' con lo que se podrá orientar la cámara siempre hacia el ROV. La posición de la cámara debe actualizarse constantemente, ya que debe mantener una vista fija hacia el vehículo de forma que debe incluirse en la función 'update' las líneas que definen la posición de la cámara también.

```
# camara principal
self.camera.reparentTo(self.render)
self.camera.setPos(self.bluerov.getPos() + Vec3(-6, 0, 3))
self.camera.lookAt(self.bluerov)
self.camera.setHpr(180,90,0)
```

- ✓ La cámara es uno de los puntos más importantes en la escena ya que da el punto de vista de la escena y por tanto la forma en la que se perciben el entorno y los objetos dentro de el por lo que es importante realizar una configuración adecuada.

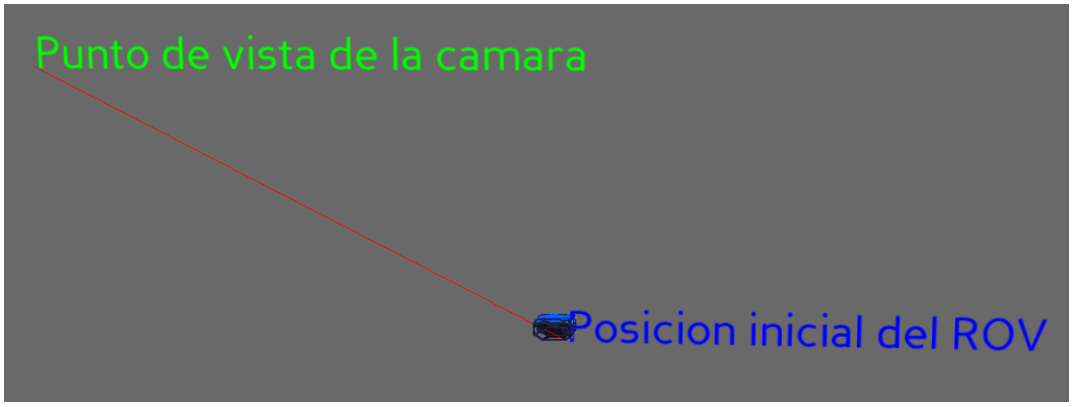


Figura 18: Representación de la visión de la cámara en la escena.

En añadido a la cámara principal, se añade una cámara secundaria que da el punto de vista del vehículo. Para configurar esta cámara se crea una nueva ventana que, gracias a la función 'setParentWindow' podrá asociarse a la ventana principal de forma que este ubicada en la parte superior izquierda de esta ventana. Si se desease separar esta visión en una ventana distinta por diversos motivos se podría eliminar la línea de código que incluye la función mencionada.

A diferencia de la cámara principal, esta cámara es fija ya que se encuentra situada en un punto especifico del BlueROV2 por lo que no debe modificarse su posición en ningún instante.

```
# camara Bluerov
wp = WindowProperties()
wp.setSize(320, 180)  # tamaño de la nueva ventana (180p)
wp.setOrigin(0, 0)
parent_window=base.win.getWindowHandle()
wp.setParentWindow(parent_window)
self.win2 = self.openWindow(props=wp)
self.last_window_size = (self.win.getXSize(), self.win.getYSize())
self.bluerov_cam = base.camList[1]
self.bluerov_cam.reparentTo(self.bluerov)
# posicion de la camara
self.cam_x=0
self.cam_y=3
self.cam_z=0
self.bluerov_cam.setPos(self.bluerov.getPos() - Vec3(self.cam_x, self.cam_y, self.cam_z))
self.bluerov_cam.setHpr(180,90,0)
```

Con todo lo anterior ya estaría el código completo que define la escena en la que se desplazara el BlueROV2 por lo que la siguiente tarea seria programar el código que se encargara de controlar el seguimiento del vehículo por la ruta definida por el usuario.

4.2.1 Seguimiento de la ruta de puntos

Para que el submarino se mueva de forma correcta en el entorno siguiendo el plan, se deben configurar una serie de parámetros. Se crea el código ‘seguimiento_kml.py’ en el que se incluye dicha configuración, así como la toma de datos del vehículo.

Al inicio del código se importan algunas librerías entre las que destacan la de ‘mavlink_lib’ y ‘utils_lib’ que se han explicado anteriormente y que se utilizarán para importar funciones específicas que serán utilizadas más adelante.

```
import sys
import mavlink_lib
import numpy as np
import utils_lib
import os
from datetime import datetime
```

La función principal del script es ‘exec’ y toma como argumento de entrada un archivo en formato ‘.kml’ que contiene los puntos de la ruta a seguir en la simulación. Se inicializa el periodo de muestreo y se establecen algunos parámetros como la distancia entre el punto (waypoint) y el vehículo a considerar para confirmar que ya ha alcanzado dicho punto, la velocidad de crucero del ROV y el máximo error que se permitirá en el yaw (ángulo de deriva) para permitir el avance del submarino. En el momento que el vehículo se encuentre a un metro o menos del siguiente waypoint se considerara como alcanzado y se comenzara el movimiento hacia el siguiente punto.

```
def exec(kml_file = None):

    # parametros simulacion y control
    ts = 1.0/60    # periodo de muestreo deseado
    sampling_time = ts   # periodo de muestreo real
    dist_to_waypoin_reached = 1.0   # distancia para considerar waypoint alcanzado
    cruise_speed = 0.5
    max_yaw_error = np.deg2rad(15)  # error de yaw para permitir avance en grados
```

La conexión se creará utilizando la librería ‘mavlink_link’ y el puerto especificado. Si la conexión falla en algún punto, la función terminará su ejecución y deberá reiniciarse el proceso.

```
# crear conexion
print('\ncreando conexion ...')
bluerov2 = mavlink_lib.MAVlink_connection('udpin:localhost:14551')
# bluerov2 = mavlink_lib.MAVlink_connection('tcp:127.0.0.1:5760')
if bluerov2.conexion == None:
    return
```

Se actualiza el periodo de muestreo para solicitar los mensajes y seguidamente se guarda en las variables ‘actual_time’ y ‘ahrs2’ los valores obtenidos en los mensajes.

```
# cambiar periodo de muestreo
bluerov2.request_message_interval('AHRS2', ts)
bluerov2.request_message_interval('SYSTEM_TIME', ts)

actual_time = bluerov2.get_info('SYSTEM_TIME', bloqueo = True).time_boot_ms*1e-3
ahrs2 = bluerov2.get_info('AHRS2', bloqueo = True)
```

Para controlar el movimiento del submarino se llama a la función del PID creada en la librería ‘utils_lib’ de forma que se aplica al ángulo de deriva, la aceleración vertical, movimiento hacia adelante y movimiento lateral del vehículo. Al llamar a esta función se especifican los parámetros del PID teniendo la ganancia proporcional (Kp) un valor de 0.5 así como la ganancia integral (Ki) y la ganancia derivativa (Kd) poseen un valor de 0.01. Estos valores se determinan tras realizar varias pruebas y ajustes con los parámetros hasta que se consigue el funcionamiento deseado.

```
# controladores PID
pid_yaw = utils_lib.PID_control(0.5, 0.01, 0.01, actual_time)
pid_lateral = utils_lib.PID_control(0.5, 0.01, 0.01, actual_time)
pid_throttle = utils_lib.PID_control(0.5, 0.01, 0.01, actual_time)
pid_fordward = utils_lib.PID_control(0.5, 0.01, 0.01, actual_time)
```

El siguiente paso es obtener los ‘waypoints’ que definen la ruta que seguirá el submarino por lo que se obtienen estos puntos del archivo en formato ‘.kml’ gracias a la función que se creó

en el código de ‘utils_lib’ llamada ‘kml2list’. Con ficha función se obtienen los puntos que se guardan en la variable ‘waypoints’, donde el primer y último punto se ajusta a la posición actual del vehículo. El plan que deberá seguir el vehículo deberá ser cerrado, es decir, que el vehículo regrese a la ubicación inicial después de visitar todos los waypoints intermedios. Con esto se consigue mantener una alta eficiencia de operación y seguridad durante la misión.

```
# obtener waypoints
if not kml_file:
    kml_file = '/home/mario/bluerov_simulation/pymavlink/prueba.kml'
waypoints = utils_lib.kml2list(kml_file)
waypoints = np.vstack((waypoints, waypoints[0]))
waypoints = np.delete(waypoints, 0, 0)
waypoints[0] = [rov_lat, rov_lon, rov_alt] #Se asigna el primer waypoint con la posicion del ROV
waypoints[-1] = waypoints[0] # idem para el ultimo waypoint
n_wp = len(waypoints)
```

Para empezar con la simulación, una vez se han obtenido los puntos que debe seguir el vehículo, se crea un bucle en el que el ROV se desplaza de un ‘waypoint’ a otro. A su vez se realizan algunas tareas de control sobre el vehículo como errores de posición, se actualizan los valores del controlador PID y se envían algunas ordenes el vehículo. Algunos de estos datos son guardados en la variable ’data_sim’ que se ha inicializado anteriormente.

```
# guardar datos
data_sim=np.vstack([ data_sim, [actual_time, ahrs2.roll, ahrs2.pitch, rov_yaw, rov_alt, rov_lat, rov_lon, error_yaw,
                                obj_xyz[0], rov_xyz[0], obj_xyz[1], rov_xyz[1], obj_xyz[2], rov_xyz[2]]])
```

Al finalizar el bucle se detienen los motores, garantizando que el ROV permanezca inmóvil. Luego se desarma el ROV y se cambia el modo de operación a 'ALT_HOLD'. En este modo el vehículo mantiene su altitud utilizando los sensores de profundidad y control sobre los motores verticales. Posteriormente se cierra la conexión y se imprime un mensaje al usuario indicando que el programa ha terminado su ejecución de forma correcta.

```
# parar motores
bluerov2.set_rc_stop()

# desarmar
bluerov2.set_arm_disarm(0)

# modo 'ALT_HOLD'
bluerov2.set_mode('ALT_HOLD')

# Cerrar conexion
bluerov2.close_connection()

# fin
print("\nPrograma finalizado\n")
```

4.3 Script de análisis en MATLAB

Una vez acabada la simulación se da la opción de exportar los datos que se han guardado a la plataforma de Matlab para mostrar diferentes valores y gráficas. Para realizar esto, se crea el script “análisis.m” en el que se programara todo lo necesario.

Los datos que se tienen en los archivos con formato ‘.csv’ se cargaran usando el comando “load”. También se especificará el número de decimales a mostrar usando el comando “format long”. Con este comando se llega a mostrar hasta dieciséis decimales mientras que si no se usa este comando únicamente se mostraran cuatro decimales. Los datos que se han guardado tienen hasta dieciséis decimales por lo que es primordial activar dicho formato.

```
%Codigo de carga y analisis de los datos
format long

data=load('data_ahrs2.csv');
waypoints=load('waypoints.csv');
```

Los datos guardados se volverán a guardar en la variable correspondiente y se realizarán algunas operaciones de transformación como pasar los valores del “hpr” de radianes a grados y transformar los waypoints a coordenadas geocéntricas.

```
%hpr (rad)
roll = data(:,2);
pitch = data(:,3);
yaw = data(:,4);
yaw_error = data(:,8);
%hpr (deg)
roll_d = rad2deg(roll);
pitch_d = rad2deg(pitch);
yaw_d = rad2deg(yaw);
yaw_error_d = rad2deg(yaw_error);

%coordenadas geodesicas
altitude = data(:,5);
lat = data(:,6);
lon = data(:,7);

%coordenadas geocentricas del ROV
rov_x = data(:,10);
rov_y = data(:,12);
rov_z = data(:,14);

%coordenadas geocentricas ruta
obj_x = data(:,9);
obj_y = data(:,11);
obj_z = data(:,13);
```

```
%coordenadas geocentricas
wgs84 = wgs84Ellipsoid('meter');
[wp_x, wp_y] = geodetic2ecef(wgs84,wp_lat, wp_lon,wp_alt);
```

Para esta última operación, debe instalarse la herramienta llamada “Aerospace Toolbox”. Para ello deberá accederse a la pestaña de aplicaciones en la parte superior del menú y buscar en la pestaña dicha herramienta.

Tras guardar los valores en las variables que le corresponden se representará en un principio una gráfica que muestre los valores “hpr”, la altitud y el error del yaw a medida que avanza el tiempo de la simulación. Se establece el tiempo de muestreo y se calcula el vector de tiempo teniendo en cuenta el número de valores de “actual_time”. Se muestra únicamente la parte del código correspondiente al valor de “Roll” ya que las demás gráficas siguen la misma estructura, solamente varía la variable a representar.

```
%ROLL
ts=1/60;
t = ((0:length(actual_time)-1) * ts)';
subplot(3,2,1)
plot(t,roll_d)
xlabel('Tiempo (s)');
ylabel('roll (grados º)');
title('Gráfico del roll en función del tiempo');
```

Se mostrará también en otra figura una representación en dos dimensiones con los valores de latitud y longitud del ROV a medida que va siguiendo la trayectoria establecida. Se usará el comando “geoplot” para dicha representación y en añadido el comando “geobasemap”, que ofrece usar como base distintos mapas en los que se representaran las coordenadas geodésicas. Para este proyecto se usará en concreto la opción “satellite” ya que se puede apreciar mejor la ubicación en la que se realiza la prueba con el vehículo.

```
%Plot de latitud y longitud
figure();
geolimits([40.60, 47],[-3.9, -3.7])
geoplot(lat,lon,"--gs","LineWidth",2, ...
    "MarkerSize",5,"MarkerEdgeColor","b", ...
    "MarkerFaceColor",[0.5 0.5 0.5])

geobasemap satellite %conexion a internet requerida
```

Por último, se representarán los valores del ROV como del objetivo a seguir en coordenadas geocéntricas para poder visualizar en 3D la ruta que sigue el vehículo, así como analizar su paso por los distintos puntos del plan. Con “Plot3” se representa un gráfico de tres dimensiones con dichos valores. Se representan los valores del vehículo, los valores de la trayectoria objetivo y los waypoints. Estos últimos serán numerados en orden de aparición en la trayectoria utilizando un bucle “for”.

```
% Distancias entre waypoints
% Crear una nueva figura
figure;
% Trazar las coordenadas en función del tiempo
plot3(rov_x, rov_y, rov_z, 'LineWidth', 2);
hold on;
plot3(obj_x, obj_y, obj_z,'r', 'LineWidth', 2);
hold on;
plot3(wp_x, wp_y, wp_alt, 'gs', 'MarkerSize', 10);
% Etiquetar waypoints
for i = 1:length(waypoints)
    text(wp_x(i), wp_y(i), wp_alt(i), sprintf('WP%d', i), 'Color', 'black','VerticalAlignment', ...
        'top','HorizontalAlignment', 'center','Fontsize', 12);

end
hold off;
title('Evolución de las coordenadas en función del tiempo');
xlabel('X');
ylabel('Y');
zlabel('Z');
legend('Ruta seguida por el vehiculo','Ruta objetivo');
grid on;
```

Una vez diseñado el código, el usuario podrá ver estas gráficas si exporta los datos utilizando la interfaz gráfica que se diseñará posteriormente.

5 APLICACIÓN

5.1 Interfaz Gráfica

Debido a que hay varios programas que deben funcionar de manera coordinada para el buen funcionamiento del sistema, se va a crear una aplicación que recoja en un solo programa la ejecución de todo el software necesario. Esta aplicación esta realizada en código Python con la ayuda del diseñador de interfaces gráficas Qt Designer, que permite configurar pestañas gráficas de forma intuitiva, sin necesidad de escribir código de forma manual. Una vez se ha diseñado la interfaz, Qt Designer ofrece una opción en la que se podrá visualizar el código Python que configura la interfaz, de manera que se podrá programar las funciones que realizan los elementos que la configuran.

Configurada la interfaz gráfica, se obtendría el resultado de la siguiente figura, en donde se tienen distintos botones que se asignaran posteriormente a funciones, así como elementos de texto relacionados con las rutas de los archivos o ejecutables que se deben seleccionar.

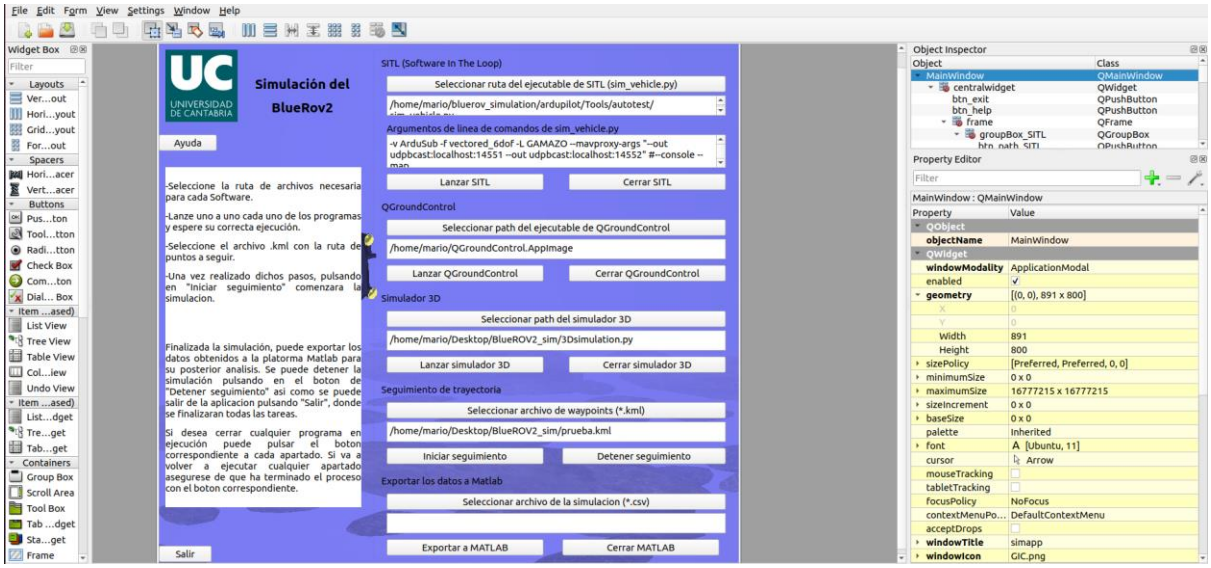


Figura 19: Interfaz gráfica diseñada vista desde Qt Designer.

Se puede observar que se divide en cinco apartados. Primeramente, se selecciona el archivo ejecutable de ardupilot, después se selecciona la ruta donde se encuentran tanto el programa QgroundControl así como la ruta del archivo de simulación en 3D, se selecciona el archivo '.kml' que contiene la ruta de puntos que seguirá el vehículo y por ultimo si el usuario lo desea puede exportar los datos de la simulación al software MATLAB.

A continuación, se deben relacionar los elementos gráficos con la función a cometer, por lo que se debe acudir ahora a el editor de código y programar de forma manual esas tareas.

5.2 Programación de los elementos en python

Para obtener el código Python que define la interfaz, se puede utilizar un comando que permite transformarlo en un archivo .py:

```
(base) mario@Ubuntu20:~$ pyuic5 app_sim.ui -o result.py
```

Si se consulta el archivo result.py aparecerá la estructura llamada UI_MainWindow conteniendo esta todos los elementos. Únicamente se usará este archivo como base para obtener así el nombre de los elementos de texto y botones que se han usado en la interfaz.

Comenzando con el código, se va a seguir el siguiente diagrama de flujo:

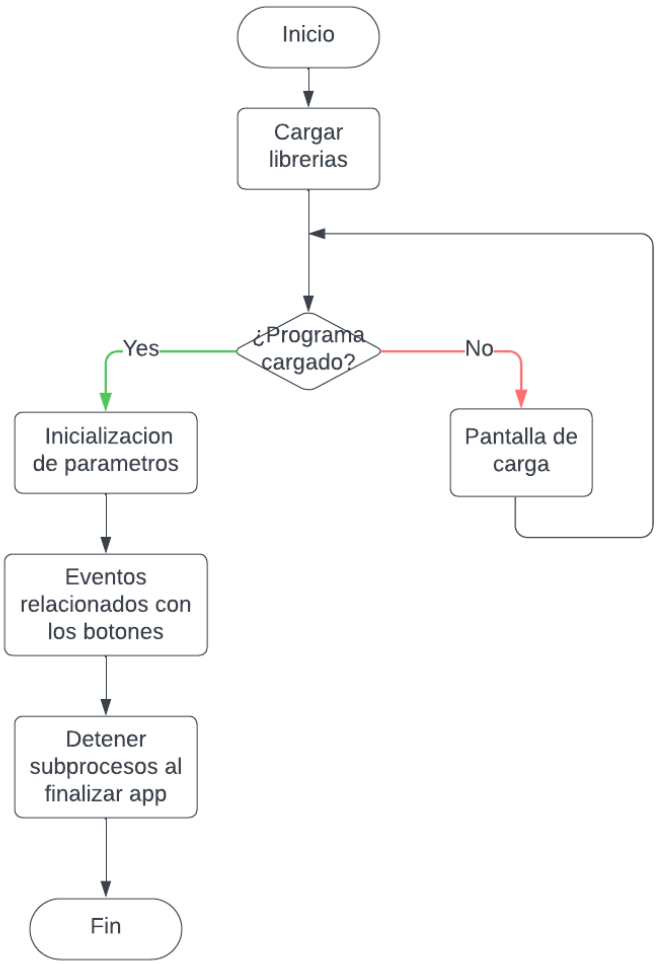


Figura 20: Diagrama de flujo de la interfaz gráfica ("app_sim").

En primera instancia se van a importar algunos módulos para cargar algunas funcionalidades específicas que serán de utilidad. Se importa ‘subprocess’ para ejecutar comandos externos desde el script. Además, se importan varios módulos de PyQt5, incluidos ‘QtCore’, ‘QtGui’, y

'QtWidgets', que son fundamentales para desarrollar interfaces gráficas de usuario (GUI) con Qt.

También se importan clases y objetos específicos de PyQt5, como 'QMainWindow', 'QApplication', y 'QMessageBox', para la gestión de ventanas, aplicaciones y diálogos interactivos. El módulo 'Qtimer' servirá de ayuda a la hora de realizar eventos temporizados. Por último, 'numpy' se usa para realizar operaciones matemáticas, así como para manipular matrices y arreglos de forma eficiente.

```
import subprocess

import sys, os, signal
from PyQt5 import QtCore, uic, QtGui
from PyQt5.QtCore import Qt ,QTimer
from PyQt5.QtCore import QObject, pyqtSignal
from PyQt5.QtGui import QPixmap
from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog, QMessageBox, QSplashScreen, QLabel
import numpy as np
import time
import psutil
```

Empezando por la estructura principal, se tiene una clase llamada "ui_app_sim" que representa la ventana principal de la aplicación. Dentro de esta se cargará la interfaz en formato "ui" que se ha mostrado anteriormente y se vincularán los botones con la función asignada a su propia pulsación, que se explicara detalladamente más adelante.

```
    # ELEMENTOS

    uic.loadUi("/home/mario/Desktop/BlueROV2_sim/app_sim.ui", self)
    self.btn_path_SITL.clicked.connect(self.btn_path_SITL_clicked)
    self.btn_start_SITL.clicked.connect(self.btn_start_SITL_clicked)
    self.btn_stop_SITL.clicked.connect(self.btn_stop_SITL_clicked)
    self.btn_path_qgc.clicked.connect(self.btn_path_qgc_clicked)
    self.btn_start_qgc.clicked.connect(self.btn_start_qgc_clicked)
    self.btn_stop_qgc.clicked.connect(self.btn_stop_qgc_clicked)
    self.btn_path_sim3d.clicked.connect(self.btn_path_sim3d_clicked)
    self.btn_start_sim3d.clicked.connect(self.btn_start_sim3d_clicked)
    self.btn_stop_sim3d.clicked.connect(self.btn_stop_sim3d_clicked)
    self.btn_path_kml.clicked.connect(self.btn_path_kml_clicked)
    self.btn_start_kml.clicked.connect(self.btn_start_kml_clicked)
    self.btn_stop_kml.clicked.connect(self.btn_stop_kml_clicked)
    self.btn_exit.clicked.connect(self.btn_exit_clicked)
    self.btn_help.clicked.connect(self.alternateAction)
    self.btn_path_data.clicked.connect(self.btn_path_data_clicked)
    self.btn_export.clicked.connect(self.export)
    self.btn_stop_matlab.clicked.connect(self.btn_stop_matlab_clicked)
    self.text_help.hide()
    self.active_status = False


    self.procesos = []
    self.proces_names = []

    self.SITL_running = False
    self.qgc_running = False
    self.sim3d_running = False
    self.kml_running = False
    self.export_running = False
```

Se oculta el texto que ofrece el botón "ayuda" de manera que no se sobrecargue la ventana con demasiada información si el usuario no lo solicita. En la aplicación se tienen varios procesos sean estos abrir, enlazar o cerrar una determinada aplicación secundaria o archivo por lo que se va a almacenar en una lista tanto la referencia del proceso creado por el sistema

como su nombre correspondiente. También se va a crear una variable en la que se indicara si un proceso en concreto está en ejecución (‘True’) o no se está ejecutando (‘False’). Con esto se podrá limitar la ejecución del mismo programa varias veces al mismo tiempo al pulsar el botón de ejecución, evitando así posibles problemas o errores futuros.

Se continua con la codificación de las funciones que obtienen la ruta del archivo(“btn_path”), las que ejecutan un archivo(“btn_start”) o las que paran su ejecución(“btn_stop”). Se va a tener una estructura similar en todas las tareas con las que interactúa el usuario por lo que se va a explicar únicamente la primera de ellas, que serían las relacionadas con el Software in the loop (SITL), utilizando el diagrama de flujo de la siguiente figura.

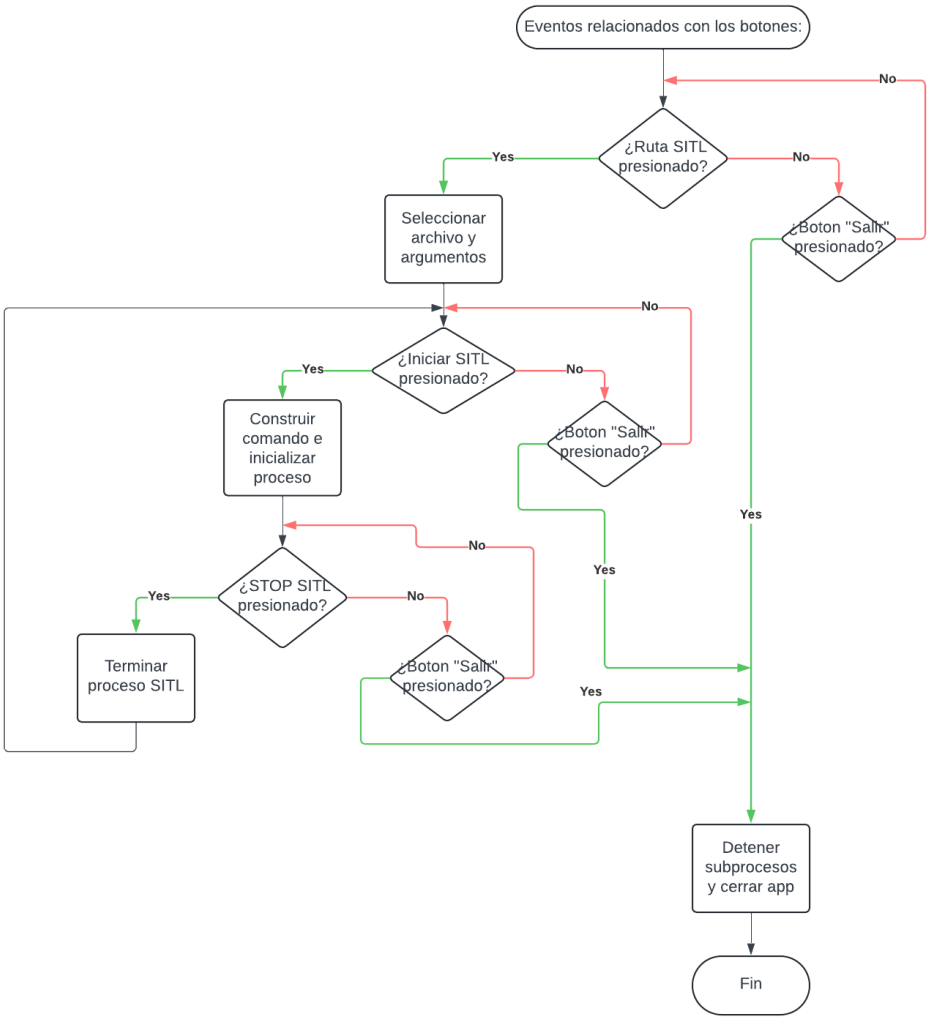


Figura 21: Diagrama de flujo relacionado con el apartado de SITL.

Comenzando con “btn_path_SITL_clicked”, en esta función se obtiene la ruta del archivo ejecutable “sim_vehicle.py” por lo que se deberá especificar en el cuadro de texto dicha ruta. Hay dos posibles opciones ya que el usuario puede pulsar en el botón de “seleccionar ruta” y

de esta forma abrir una nueva pestaña para seleccionar una nueva ruta y sustituirá la que se encontraba definida previamente en el cuadro o que el usuario no haga nada y se mantenga la ruta preestablecida.

```
# Ruta SITL
def btn_path_SITL_clicked(self):
    text = self.txt_path_SITL.toPlainText()
    options = QFileDialog.Options()
    options |= QFileDialog.DontUseNativeDialog
    fileName, _ = QFileDialog.getOpenFileName(self,"Seleccione el path del ejecutable sim_vehicle.py ","","All Files (*)", options=options)
    if not fileName:
        self.txt_path_SITL.setPlainText(text)
    else:
        self.txt_path_SITL.setPlainText(fileName)
```

En la función “btn_start_SITL_clicked” se ejecuta el archivo disponible en la ruta seleccionada anteriormente, de forma que se toma la ruta(‘filename’) y los argumentos que se especifican al ejecutable(‘args’) y se ejecuta en una terminal ‘konsole’(command).

Al realizar esta tarea, se va a crear “p” que representa el subproceso que se acaba de crear y que se va a añadir a la lista de procesos que se mencionó. Esto último será primordial para su uso en la función que para la ejecución de los subprocesos.

```
# Iniciar SITL
def btn_start_SITL_clicked(self):
    if not self.SITL_running:
        self.SITL_running = True
        filename = self.txt_path_SITL.toPlainText()
        args = self.txt_args_SITL.toPlainText()
        if not args:
            args = '-v ArduSub -f vectored_6dof -L ISLA_DE_LA_TORRE --mavproxy-args "--out udpbcast:localhost:14551" #--console --map'
            self.txt_args_SITL.setPlainText(args)

        command = 'konsole --workdir /home/mario/bluerov_simulation/ardupilot/ArduSub -e ' + filename + ' ' + args

        p = subprocess.Popen(command, shell = True, preexec_fn=os.setsid)
        self.procesos.append(p)
        self.proces_names.append('p_sitl')
    else:
        self.show_message("El SITL ya está en ejecución.")
```

Si el usuario desea finalizar con la ejecución del “SITL” deberá pulsar en el botón “cerrar SITL” de forma que entrará en juego la función “btn_stop_SITL_clicked”. Utilizando una estructura de excepciones, se busca la posición del proceso deseado en la lista de nombres llamada ‘proces_names’ asignando este valor a ‘p_ind’ y eliminando posteriormente este proceso de ambas listas de procesos. Utilizando la librería ‘psutil’ se crea un objeto pariente con el PID (‘p.pid’) de forma que se itera sobre los procesos hijos que puedan estar asociados y se terminan cada uno de ellos y a su vez terminando el proceso principal. Es más complejo detener este subproceso del SITL debido a que en su ejecución en la terminal ‘konsole’ se usan varios argumentos por lo que habría que recurrir a este método.

```
# Parar SITL
def btn_stop_SITL_clicked(self):
    try:
        p_ind = self.proces_names.index('p_sitl')
        p_name = self.proces_names.pop(p_ind)
        p = self.procesos.pop(p_ind)

        parent = psutil.Process(p.pid)
        QApplication.processEvents()
        for child in parent.children(recursive=True):
            child.terminate()
        parent.terminate()

        for proc in psutil.process_iter():
            if "xterm" in proc.name():
                proc.terminate()
        self.SITL_running = False

    except Exception as e:
        self.show_message("Error al detener el proceso.")
```

Para las tareas restantes las funciones de ‘path’ son idénticas. Sin embargo, para las que inician un proceso varían en la forma de ejecución de la terminal ‘konsole’ ya que los argumentos necesarios para su ejecución son distintos. Las funciones restantes responsables de detener los procesos son similares al ejemplo. Sin embargo, debido a que su ejecución es más sencilla, se utiliza un método diferente para acabar con dicho proceso

La aplicación incluye una opción en la que el usuario, si lo desea, puede exportar los datos que se han tomado a un entorno en MATLAB. Para realizar esto en código, simplemente se va a tomar la ruta de la carpeta que se ha creado para la última simulación como se explicó anteriormente y a su vez se va a tomar la ruta del archivo ‘.m’ que trata los datos.

```
# Exportar datos
def export(self):
    if not self.export_running:
        self.export_running = True
        data_file = self.txt_path_matlab.toPlainText()
        path_data = os.path.dirname(data_file)
        print("",path_data)
        script = "analysis.m"
        ruta_script = "/home/mario/Desktop/BlueROV2_sim"
        command = f'konsole -e /usr/local/MATLAB/R2024a/bin/matlab -r "cd(\'{ruta_script}\'); addpath(\'{path_data}\'); run(\'{script}\')"'
        p = subprocess.Popen(command, shell= True)
        self.procesos.append(p)
        self.proces_names.append('p_matlab')
    else:
        self.show_message("El proceso ya está en ejecución.")
```

El archivo con formato ‘.m’ tomará la carpeta donde se encuentra ubicado el archivo con formato ‘.csv’ con los datos y ejecutará el programa mostrándole al usuario algunas gráficas con los datos más significativos para tener en cuenta para el posterior análisis del funcionamiento general del sistema.

Cuando el usuario haya acabado con el análisis, si desea cerrar únicamente la plataforma y seguir realizando simulaciones con distintos planes tendrá dicha opción si usa el botón ‘Cerrar MATLAB’ en el que a diferencia de otros procesos al estar asociados varios argumentos en

el momento de la ejecución del comando en ‘konsole’, se debe recurrir a la librería ‘psutil’ para terminar completamente con el proceso, como ya se hizo anteriormente con la ejecución del SITL. El identificador del proceso padre se guarda en una variable de forma que primero acabará con los procesos secundarios relacionados al principal para terminar finalmente la ejecución del proceso ‘parent’.

- ✓ Es muy importante tener en cuenta todos los aspectos de la aplicación ya que se debe facilitar en mayor medida al usuario el uso de esta por lo que se debe priorizar la sencillez e intuición. Si se utilizase otro método con estos subprocesos podría no acabar con él y entorpecer la experiencia del usuario.

```
# Detener matlab
def btn_stop_matlab_clicked(self):
    try:
        p_ind = self.proces_names.index('p_matlab')
        p_name = self.proces_names.pop(p_ind)
        p = self.procesos.pop(p_ind)

        # Usar psutil para matar el proceso hijo
        parent = psutil.Process(p.pid)
        for child in parent.children(recursive=True):
            child.terminate()
        parent.terminate()
        self.export_running = False
    except Exception as e:
        self.show_message("Error al detener el proceso.")
```

Para abandonar la aplicación, se pulsará el botón ‘Salir’ de manera que aparecerá en pantalla una nueva pestaña(“QMessageBox”) en la que se preguntará al usuario si desea abandonar la aplicación. Si se selecciona la opción ‘Si’ se finalizan todos los procesos, de igual forma que con los botones específicos de las tareas, y se cerrara la pestaña principal de la aplicación. Otra forma de cerrar la aplicación es usando la opción de cerrar que ofrece la barra de título de la ventana en la que, de igual forma que usando el botón ‘Salir’, se finalizará la ejecución de los subprocesos y la aplicación general.

```python
def btn_exit_clicked(self):
    # Mostrar un cuadro de diálogo para confirmar el cierre
    title = 'Aviso'
    text = '¿Estás seguro de que deseas cerrar la aplicación?'

    # Crear y configurar el cuadro de diálogo
    msg = QMessageBox()
    msg.setIcon(QMessageBox.Question)
    msg.setText(text)
    msg.setWindowTitle(title)

    # Cambiar el texto del botón estándar de "Sí" a "Confirmar"
    msg.setStandardButtons(QMessageBox.No | QMessageBox.Yes)
    confirm_button = msg.button(QMessageBox.Yes)
    confirm_button.setText('Sí')

    msg.setDefaultButton(QMessageBox.No)

    # Ejecutar el cuadro de diálogo y verificar la respuesta
    res= msg.exec_()

    if res == QMessageBox.Yes:
        # Cerrar todos los procesos antes de salir
        for p in self.procesos:
            os.kill(p.pid, signal.SIGTERM)
        # Cerrar la ventana y salir de la aplicación
        self.close()
```

Se incluyen algunas características o funciones más básicas que no afectan al desarrollo del proceso de simulación buscado como la creación de la función que muestra el texto de ayuda, así como la de una “SplashScreen” como pantalla de carga de la aplicación.

Figura 22: Pantalla de carga de la aplicación.

6 SIMULACIÓN

Una vez que está diseñado el entorno 3D, se podrá realizar una simulación vinculando el SITL de ArduSub con el software de QgroundControl, y de esta forma vincular el plan de ruta que se va a llevar a cabo con el simulador 3D.

Para ello, se debe acceder a la aplicación diseñada llamada “simapp” que se encuentra en el escritorio para facilitar su acceso, como se muestra en la figura. Se hará clic en el icono y empezará su ejecución.

Figura 23: Icono de la interfaz gráfica "simapp".

Una vez iniciada la aplicación, aparecerá la interfaz que se observa en la siguiente figura donde hay diferentes apartados. Para un funcionamiento correcto del sistema se deberá ejecutar el SITL y esperar a que este se cargue correctamente para proceder. Después de que este correctamente inicializado, se abrirá la aplicación QGroundControl y el simulador 3D. Con lo anterior, se podrá seleccionar el plan deseado para la simulación y se podrá ejecutar el código relacionado con el seguimiento. Es muy importante ejecutar los apartados de forma individual y esperar a que hayan cargado todos sus parámetros de forma correcta, si no es así podrían ocurrir fallos durante la simulación y, por tanto, que esta no sea válida.

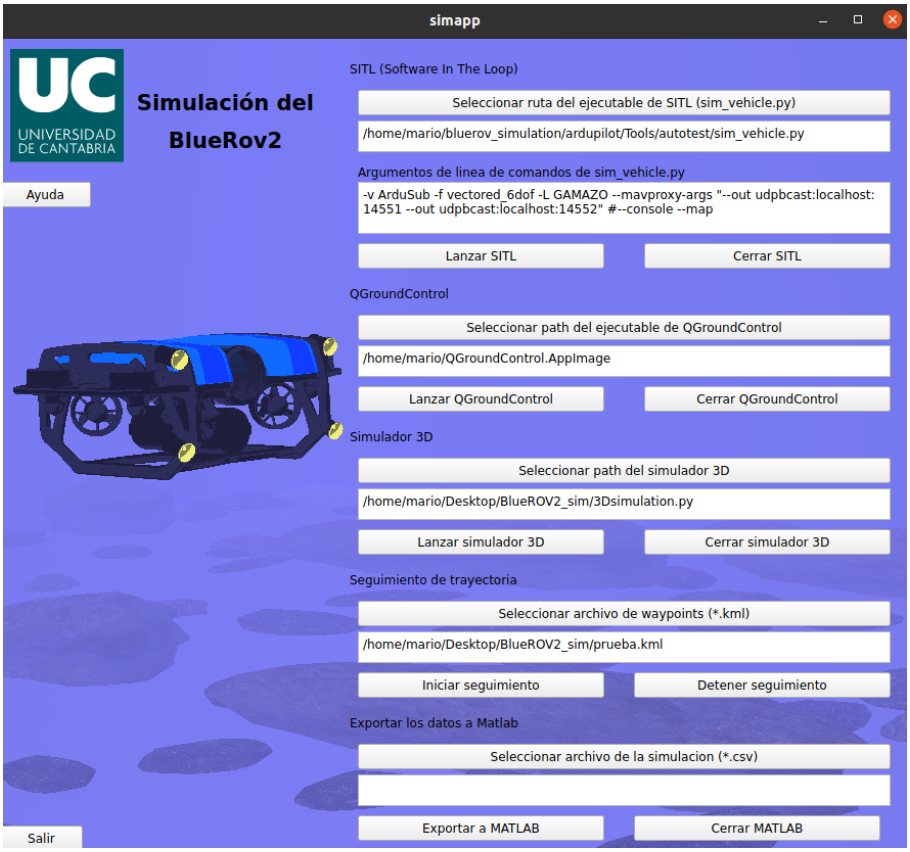


Figura 24: Vista principal de la aplicación diseñada "simapp".

6.1 Configuración del SITL

Lo primero será cargar propiamente dicho el SITL. Para ello, se accede a los archivos que contiene la herramienta de Ardusub, y se busca un archivo de Python llamado ‘sim_vehicle.py’ que se encuentra en la carpeta “tools”. Este permite cargar un modelo determinado de vehículo con las físicas simuladas correspondientes para seguidamente iniciar la simulación del vehículo mediante la herramienta ArduSub.

Al poder cargar los modelos y físicas necesarias, este archivo permite configurar un conjunto de parámetros que son esenciales para la correcta configuración de la simulación. En la ventana de comandos, a la vez que se llama al archivo, se le pueden indicar diferentes parámetros, entre los que destacan la ubicación de partida del vehículo ROV, la configuración del vehículo submarino que se está utilizando y demás.

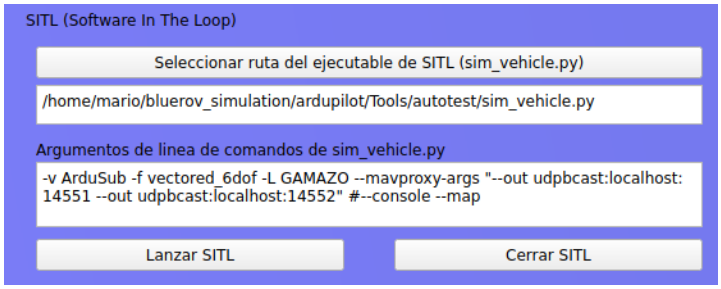


Figura 25: Apartado del SITL en la interfaz.

Se debe indicar con qué tipo de vehículo se está trabajando. Sin embargo, al utilizar la herramienta ArduSub que únicamente se centra en vehículos submarinos no tiene importancia. Seguidamente, se indica la configuración del vehículo ROV, el cual posee una configuración compuesta de vectores de empuje en seis grados de libertad, pudiendo controlar la profundidad y dirección del vehículo.

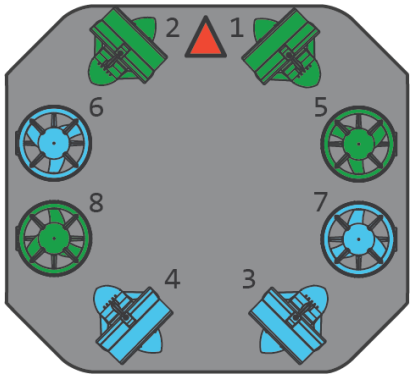


Figura 26: "Frame" del BlueROV2 heavy

La ubicación seleccionada para la simulación será la del dique de Gamazo, pudiendo observar su latitud y longitud exacta en la siguiente figura.

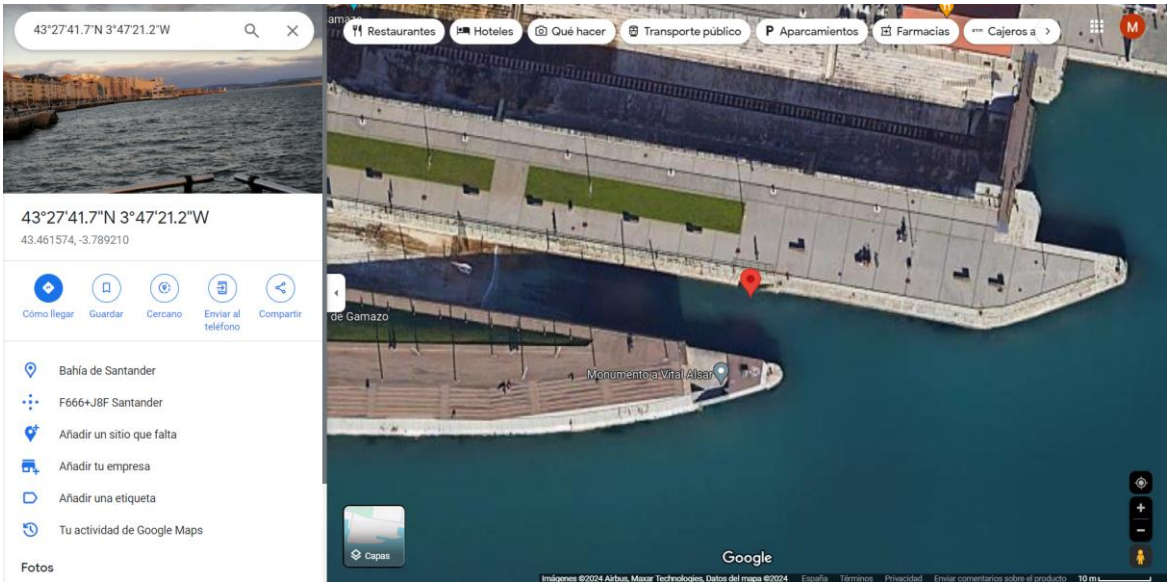


Figura 27: Ubicación inicial del ROV en Google Maps.

Para cargar esta ubicación directamente en el software, se debe acceder a el archivo 'locations.txt', que se encuentra en la misma carpeta que el archivo ejecutable, y se deberá añadir las coordenadas correspondientes a esa ubicación con un nombre asignado. De esta forma se podrá cargar dicha ubicación directamente desde los argumentos del ejecutable.

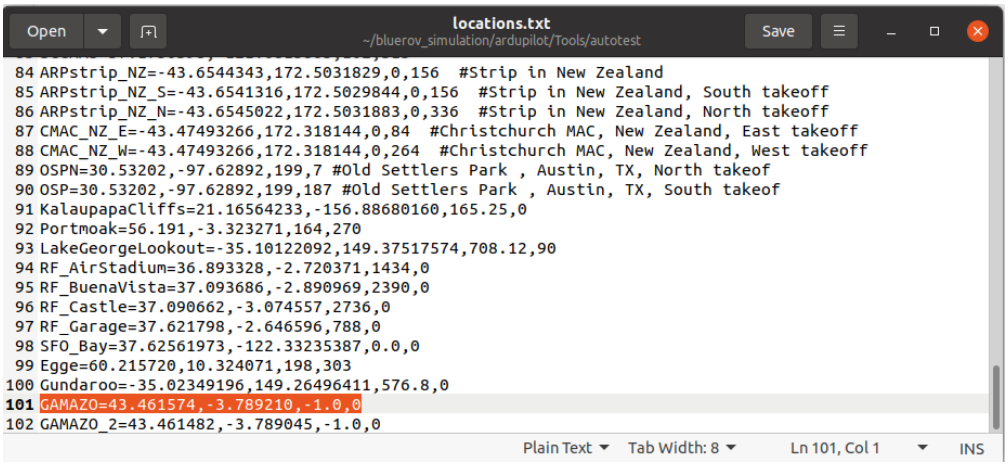


Figura 28: Ubicación "GAMAZO" añadida al archivo 'locations'.

Se realiza la conexión entre el vehículo y la estación en la superficie (ordenador) para su propia comunicación. En los argumentos de –mavproxy-args se configura la forma en la que se envían los datos al propio MAVProxy. Se establecen dos puertos de comunicación, el 14551 y 14552, utilizando el protocolo UDP de forma que se podrá trasmitir los datos por difusión, pudiendo recibir varios destinatarios el mismo mensaje.

Con todo lo anterior, se ejecuta en la aplicación el SITL de forma que se cargaran los paquetes y ArduSub mostrará la pestaña vista en la figura.

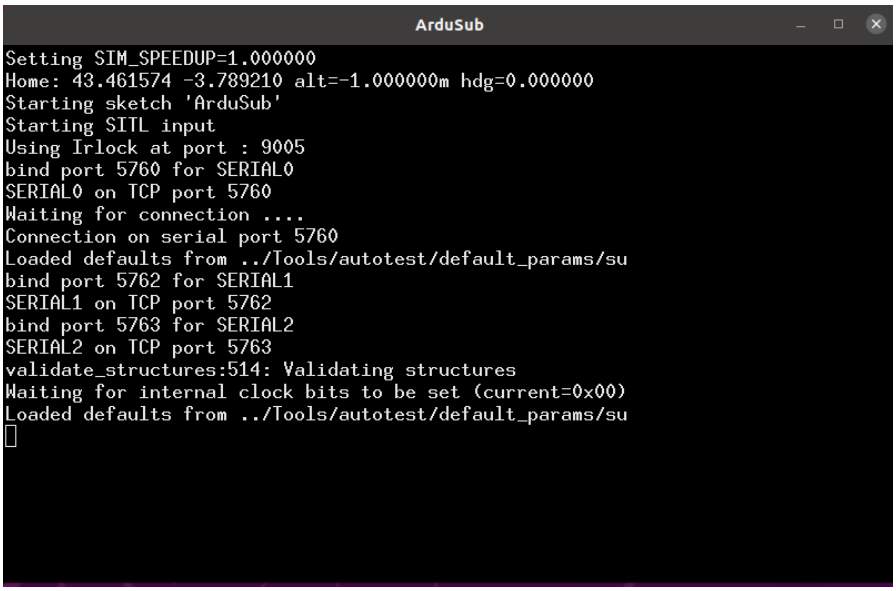


Figura 29: Parámetros principales al ejecutar el archivo.

A su vez, se abre otra pestaña en la que muestra cómo se cargan y comprueban los parámetros necesarios para la carga del SITL.

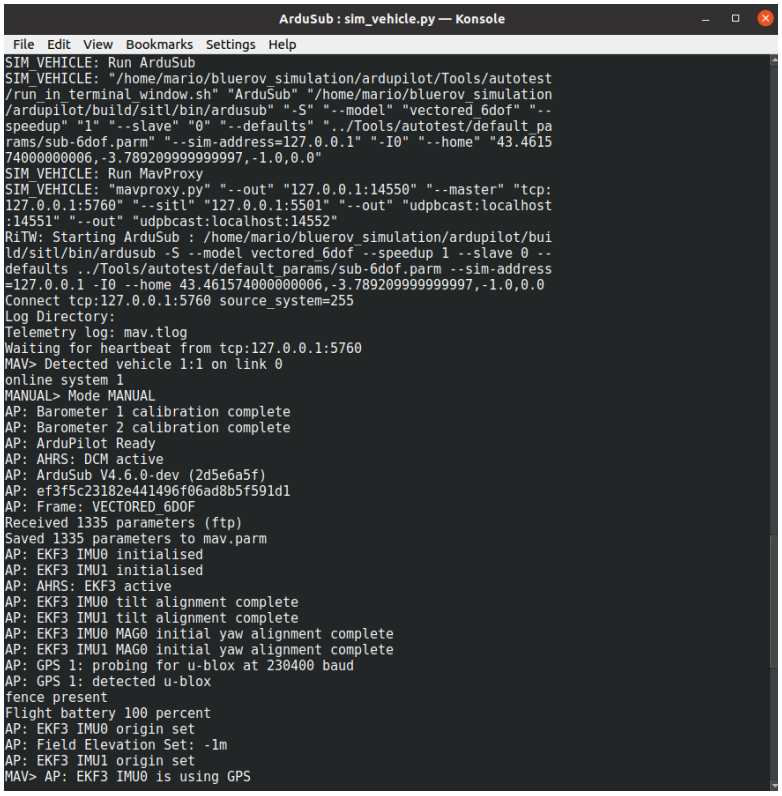


Figura 30: Consola en la que se ejecuta "sim_vehicle" y donde se inicializan los parámetros.

Se comprueba la calibración correcta de los sensores propios del vehículo y demás aspectos físicos, así como la carga de los parámetros explícitos que se le ha dado para la simulación.

6.2 Conexión con QGroundControl

Con el SITL ya ejecutado, se debe lanzar la aplicación QgroundControl de forma que estará ya vinculado el vehículo a esta aplicación y se debería ver la ubicación de partida seleccionada en los parámetros de ejecución del SITL. Al ser una aplicación desarrollada como base para vehículos aéreos, no se podrá más que comprobar el seguimiento de la ruta planteada en 2D y a su vez comprobar los datos básicos de velocidad y altitud correspondientes al vehículo autónomo.

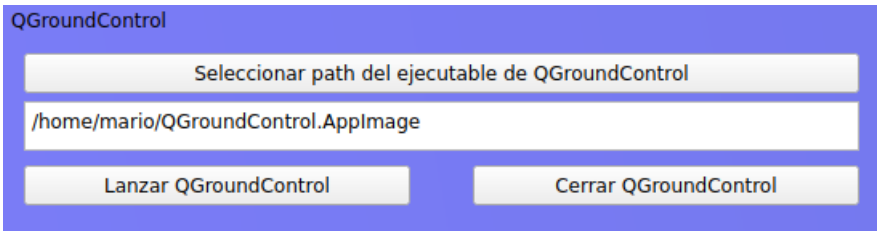


Figura 31: Apartado de ejecución de QGrounControl.

En la siguiente figura se pude observar el aspecto del programa previo a la ejecución del plan de ruta. El BlueROV2 se encuentra ya conectado y listo para empezar la tarea.

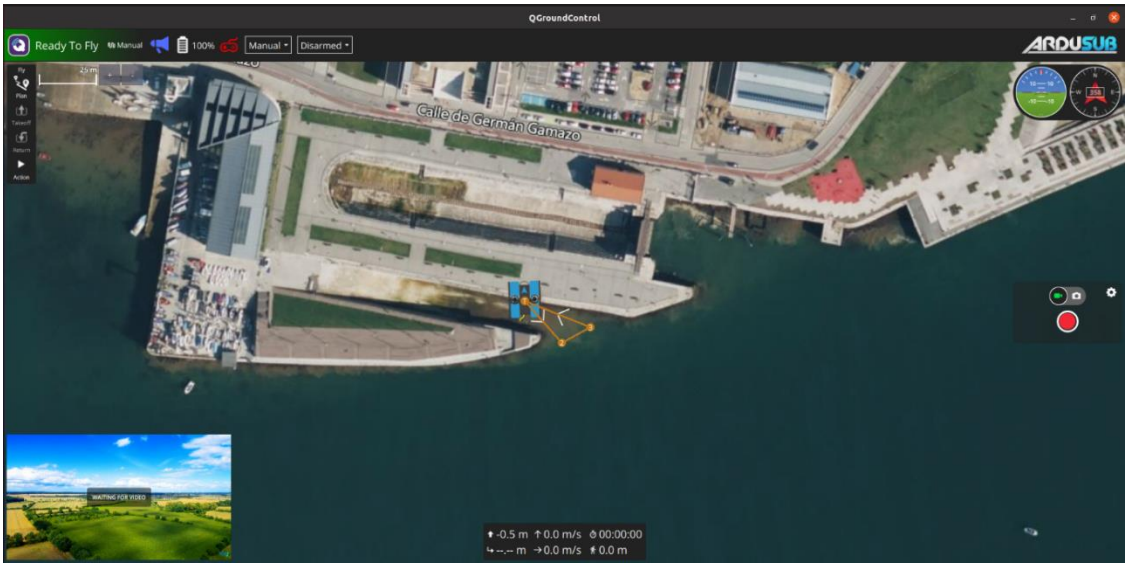


Figura 32: Vista previa a la simulación del seguimiento en QgroundControl

Cuando se inicie el seguimiento, se podrá observar en la aplicación como el vehículo va siguiendo la ruta de puntos que se muestra en color naranja.

6.3 Simulador 3D

Para ejecutar la escena virtual, se debe seleccionar la ruta del ejecutable que la incluye y después presionar en ‘’Lanzar simulador 3D”.

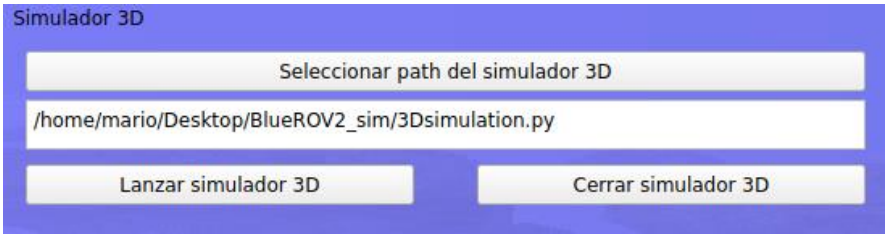


Figura 33: Apartado del simulador 3D dentro de la interfaz.

A continuación, aparecerá la terminal donde se ha ejecutado el script de Python y tras comprobar que todo este correcto cargara la escena 3D. En la terminal se indicará el estado de la conexión con el vehículo. Si el 'SITL' no se ha ejecutado correctamente, se intentará establecer la conexión hasta diez veces, con un intento cada segundo. Si después de estos intentos la conexión no se ha logrado, se detendrá la ejecución del script de la escena.

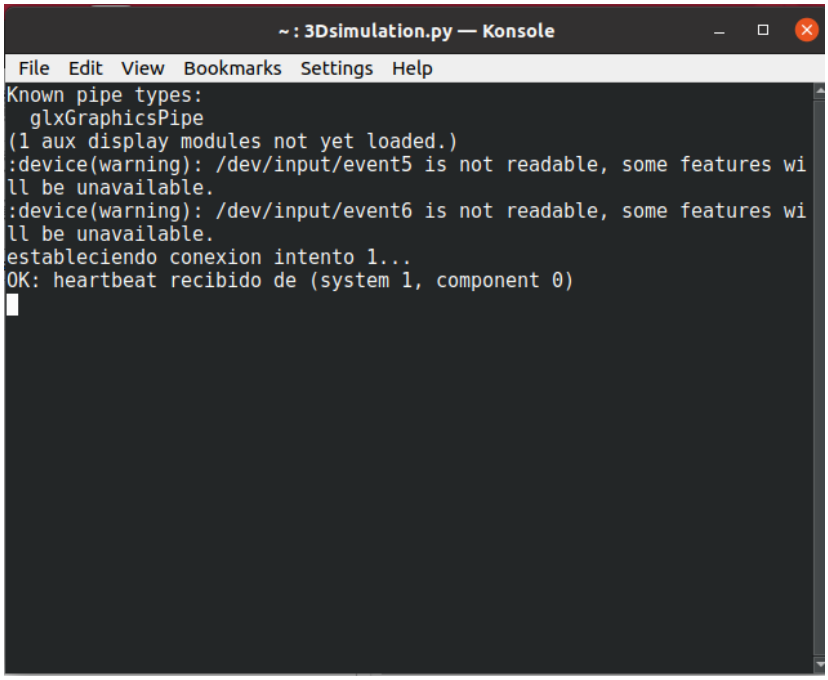


Figura 34: Consola donde se ejecuta el simulador.

En primera instancia el submarino se encuentra estático esperando a que se inicie el seguimiento y así comenzar su movimiento hacia los “waypoints”. La tasa de fotogramas por segundo se estableció con un valor límite de 60 FPS. Sin embargo, se podrían producir alguna bajada repentina de dicho valor debido al uso de las texturas y efectos del programa.

Figura 35: Inicialización del simulador.

6.4 Ruta de puntos

El objetivo de la aplicación es observar cómo el ROV sigue correctamente la ruta de puntos. Para comprobar esto, se debe seleccionar el plan que incluye los distintos puntos en formato

'.kml'. Este archivo puede generarse en la aplicación de QgroundControl en el que se podrá establecer un punto de inicio, punto de fin, el ángulo de yaw para cada waypoint, así como la altitud. La trayectoria utilizada en esta simulación constara de cuatro puntos, incluyendo el punto de inicio("Takeoff") y de retorno. En la parte izquierda del programa existe una barra que incluye diferentes herramientas para crear y modificar el plan, donde se puede establecer el punto "home" así como los diferentes waypoints que configuran la ruta.

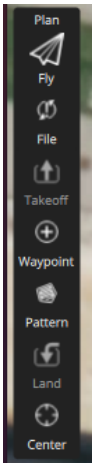


Figura 36: Menú de herramientas para la edición del plan.

Al ir configurando este plan, se podrá ver en la parte superior del programa la herramienta "plan toolbar", donde se puede seleccionar cada uno de los puntos de la trayectoria y observar la distancia que existe con el waypoint anterior, la diferencia de altitud entre dos puntos y el valor de yaw con el que alcanza ese punto (predefinido por QGroundControl).

Figura 37: Plan definido para la simulación.

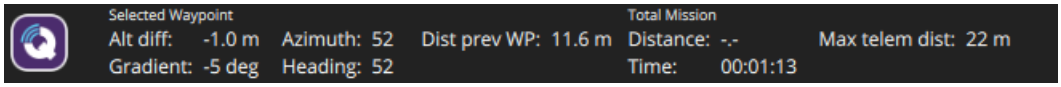


Figura 38: Datos del tercer waypoint.

El archivo puede guardarse después con formato ‘. plan’ o formato ‘.kml’. El primer formato se usa para planificar y ejecutar directamente en QgroundControl misiones detalladas en las que se usa un vehículo no tripulado (UAV o ROV) mientras que el formato ‘.kml’ se utiliza para representar datos geoespaciales en distintas aplicaciones siendo una de ellas Google Earth. Para esta tarea, es más útil el último formato ya que este apartado de la simulación únicamente se centra en la visualización y análisis de la ruta.

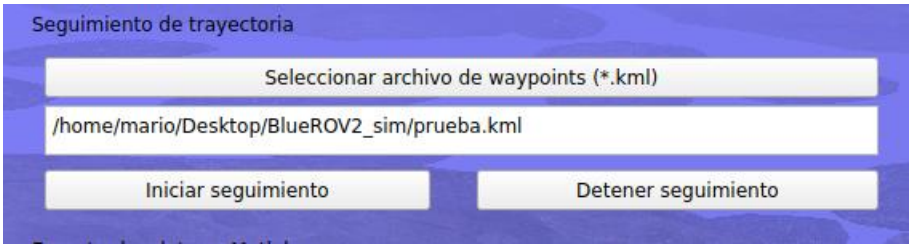


Figura 39: Apartado del seguimiento en la interfaz.

Tras pulsar el botón de “Iniciar seguimiento”, se abrirá otra terminal konsole en la que, al igual que en el simulador 3D, establecerá la conexión y si no es posible se finalizará la ejecución del programa. Si se crea la conexión de forma correcta, aparecerán los puntos que debe seguir el vehículo indicando la latitud, longitud y altitud. Después aparece un mensaje que indica el comienzo de las maniobras, así como el armado del ROV.

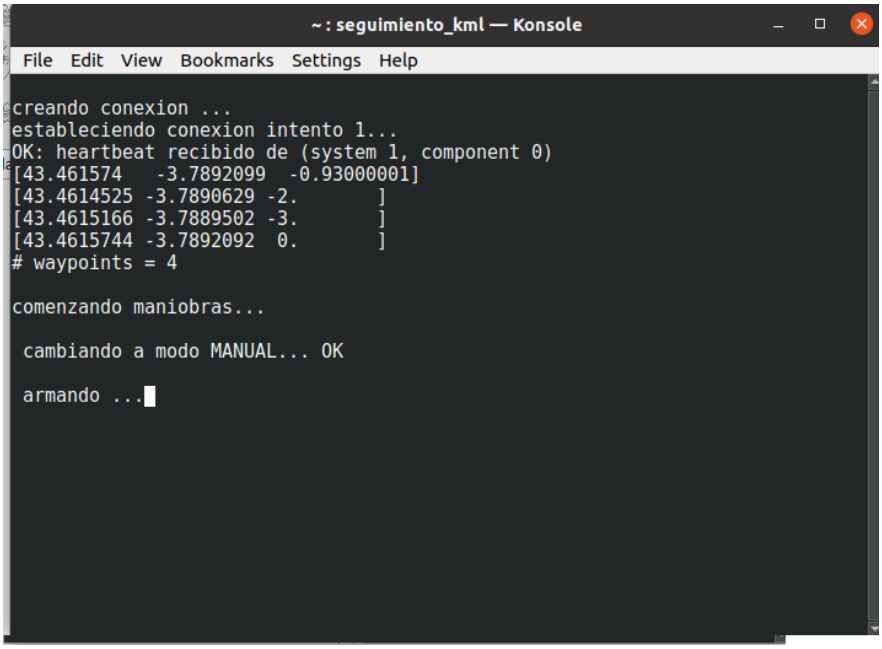


Figura 40: Inicialización del seguimiento en la consola.

Con el ROV armado empezara el seguimiento comenzando por el primer punto. En este ejemplo se tienen dos “waypoints” diferentes ya que el primer y cuarto punto que se ve en la figura representan la posición inicial y posición final, debido a que el plan utilizado usa un circuito cerrado. Entre punto y punto se muestra diferente información valiosa tanto del

vehículo como información del seguimiento en coordenadas cartesianas. Entre esta información destaca la que actualiza la distancia al siguiente “waypoint” y el error del yaw. Con dicha información se podrá comprobar que se alcanza adecuadamente el punto y que llegue con la orientación correcta.

Una vez llegue al “waypoint” final, el programa indicara en un mensaje que ha finalizado su ejecución correctamente y se cerrara la terminal. A partir de este momento, el usuario puede cerrar los programas para realizar una nueva simulación del seguimiento, dado que la posición del vehículo ha variado. Debe volver a ejecutarse el SITL para reestablecer la posición en ‘GAMAZO’ e inicializar de nuevo los valores.

Figura 41: Vehículo desplazándose hacia el primer waypoint.

Figura 42: Vehículo desplazándose hacia el ultimo waypoint.

6.5 Transferir los datos a la plataforma MATLAB

El usuario puede exportar los datos de la simulación a MATLAB directamente desde la aplicación si lo desea. Durante el seguimiento del vehículo se ha generado una carpeta en la ruta “simulaciones” que contiene un archivo con formato “.csv” con algunos datos de la

simulación. Este formato permite al usuario exportar los datos a cualquier lenguaje de programación o incluso manipularlo mediante software como Microsoft Excel o LibreOffice.

La aplicación ofrece por defecto la opción de exportar los datos a MATLAB para su análisis. El usuario puede elegir la carpeta que contiene el archivo de la última simulación (opción preestablecida) o seleccionar una carpeta de una simulación anterior.

En la interfaz, se selecciona el archivo de la última simulación dentro de la carpeta y luego se hace clic en el botón "Exportar a MATLAB". Esto abrirá una ventana de la plataforma donde se ejecutará el script que generará diversas gráficas.

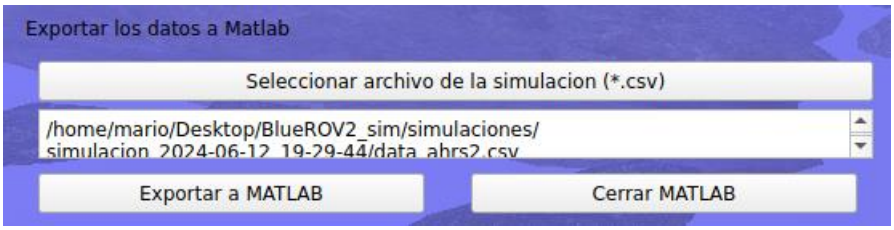


Figura 43: Apartado de carga de datos en MATLAB dentro de la interfaz.

Una vez abierto MATLAB se ejecutará el script “análisis.m” en el que se cargaran los datos de la simulación recogidos en “data_ahrs2.csv” y también los waypoints en otro archivo con formato ‘.csv’. Primeramente, se guardarán los datos en las variables para después representar gráficas de los mismos. En la siguiente figura se puede observar unas gráficas que representan los valores de la orientación del vehículo y de la altitud a lo largo de la simulación.

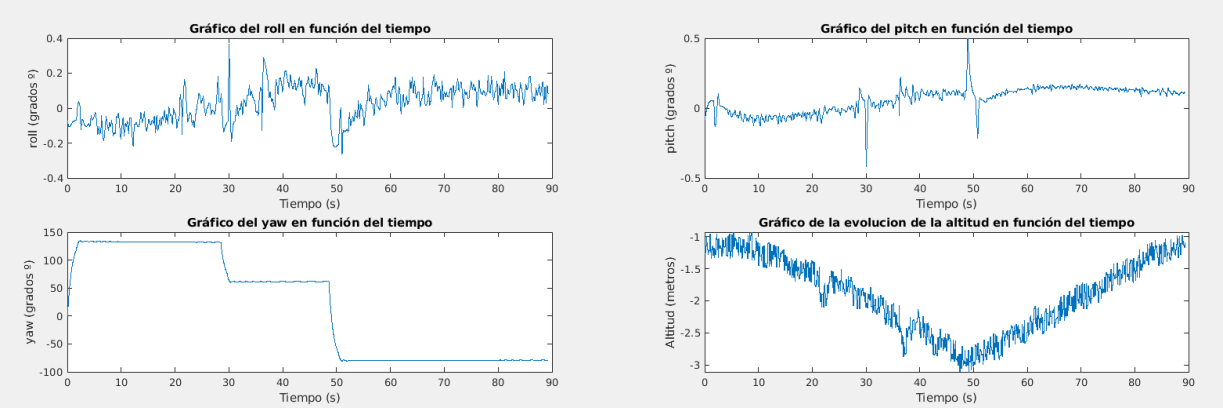


Figura 44: Valores de la orientación(º) y de la altitud(m) obtenidos de la simulacion.

A su vez se muestra otra gráfica en la misma figura mostrando el error del yaw. Se observa claramente como este valor vario levemente en las reorientaciones que se producen al alcanzar los diferentes puntos de la ruta y como en los demás instantes el valor se mantiene cercano a cero.

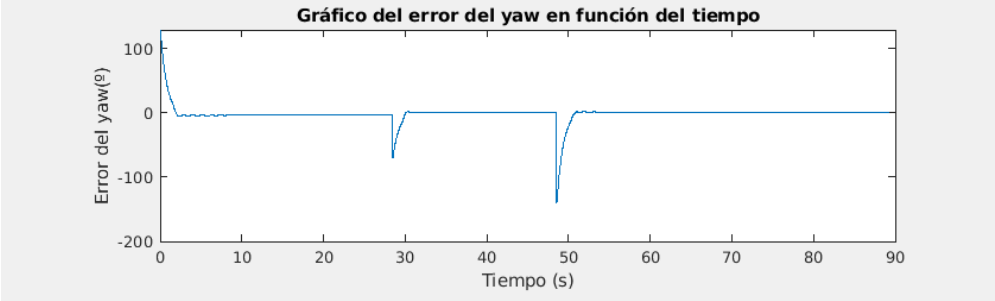


Figura 45: Error del yaw.

Teniendo los valores de la latitud y longitud del BlueROV2 recogidos en la simulación, se puede mostrar el camino que ha seguido el vehículo sobre la imagen satélite real de la ubicación.

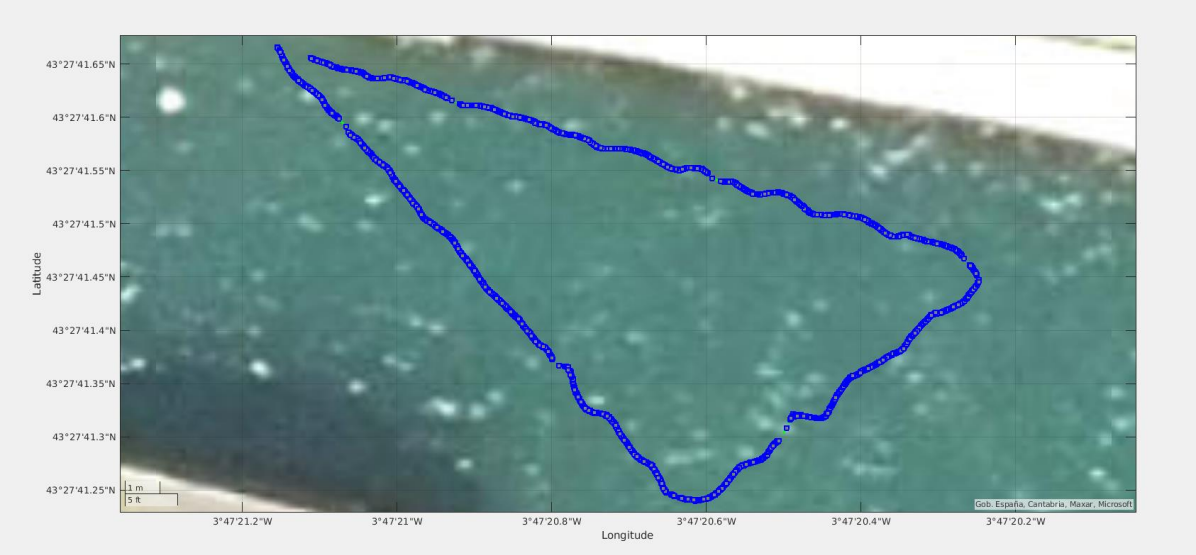


Figura 46: Gráfico 2D de la ruta seguida utilizando los valores de latitud y longitud.

En la figura anterior únicamente se puede apreciar la ruta en 2D. Sin embargo, durante el seguimiento se producen variaciones en la altitud dependiendo del punto a alcanzar por lo que debe mostrarse la gráfica representada con “plot3” que tiene en cuenta también la altitud.

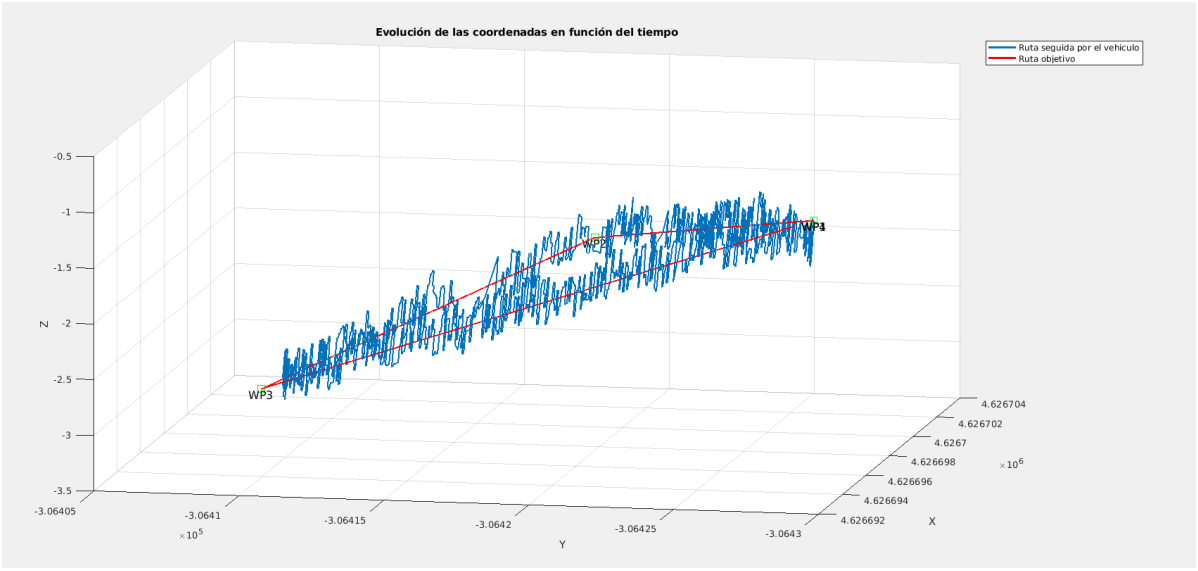


Figura 47: Gráfica de las coordenadas geocéntricas de la ruta deseada (rojo) y la ruta seguida (azul).

En esta representación 3D se muestra en color azul la ruta que sigue el vehículo, con su correspondiente nivel de ruido en la altitud, y en color rojo se muestra la ruta que debería seguir. También se marcan los waypoints con un cuadrado verde estando identificados de forma numérica.

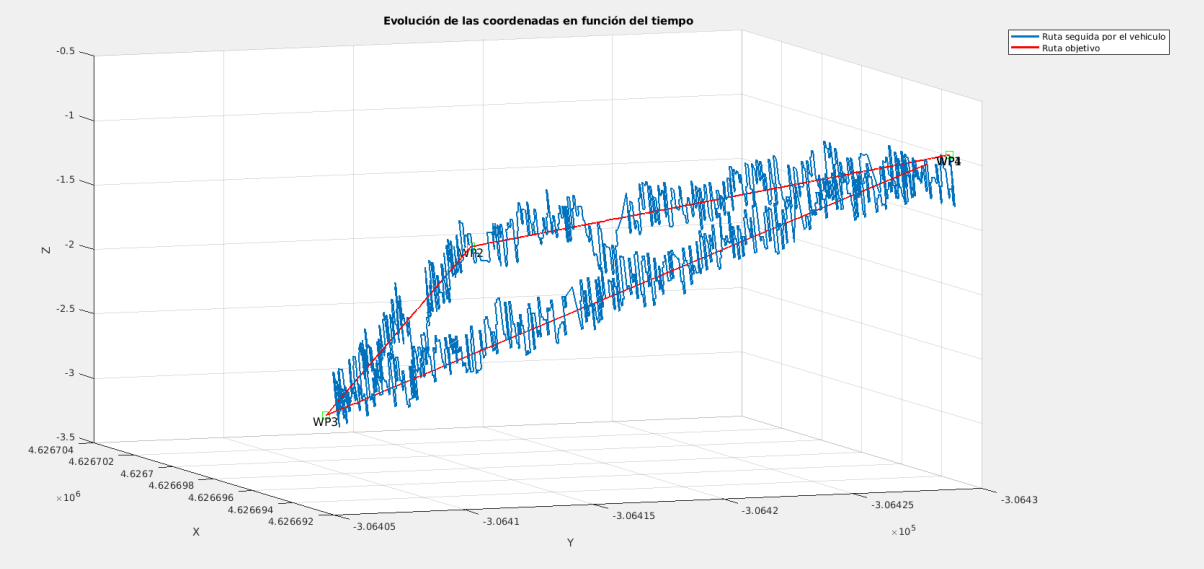


Figura 48: Gráfica de las coordenadas geocéntricas de la ruta deseada (rojo) y la ruta seguida (azul) desde otro punto de vista.

En esta otra perspectiva se ve como el vehículo va siguiendo aproximadamente la trayectoria y como alcanza los waypoints establecidos en el plan.En el siguiente apartado se realizará un análisis de los resultados de la simulación.

6.6 Análisis de los resultados

En este último apartado se van a estudiar los resultados obtenidos de la simulación con el BlueROV2 al realizar un seguimiento sobre una ruta de puntos.

En cuanto a la recopilación y procesamiento de los datos, se debe valorar si los valores que dan los sensores del vehículo durante la prueba son almacenados de forma correcta y que no hay ningún valor erróneo o extraño. La trayectoria o plan definido debe ser claro y se deben alcanzar las coordenadas de los waypoints en un determinado orden. Todo esto anterior se cumple analizando los valores recopilados. Ahora se debe comprobar dentro de estos la eficiencia del vehículo, es decir, la desviación que existe entre la trayectoria definida por el usuario y la trayectoria que sigue el ROV.

Para medir dicha desviación entre la ruta objetivo y la ruta que es seguida por el BlueROV2 se puede calcular la distancia euclidiana entre los puntos correspondientes de ambas trayectorias en cada instante de tiempo. De esta forma se podrá observar que tan lejos está el vehículo de la trayectoria deseada en cada punto.

La desviación euclídea es una medida de la separación entre dos conjuntos que se encuentran en un espacio euclídeo. En este caso, se considera la escena de simulación un espacio euclídeo, donde se utiliza un sistema de coordenadas cartesianas para describir la posición del vehículo. Se calcula como la raíz cuadrada de la suma de los cuadrados de las diferencias en cada punto correspondiente en los conjuntos de puntos comparados [15].

$$desviacion(P1, P2) = \sqrt{(rov_x - obj_x)^2 + (rov_y - obj_y)^2 + (rov_z - obj_z)^2}$$

Ecuación 6: Ecuación para el cálculo de la desviación euclídea utilizada.

Para realizar esto con los valores tomados, se desarrolla unos comandos adicionales en MATLAB de forma que se obtiene la siguiente gráfica mostrando la desviación entre la ruta deseada y la que sigue el ROV para cada instante de tiempo.

Figura 49: Desviación entre la ruta deseada y la ruta seguida por el BlueROV2.

Se observa como la máxima desviación con valor de un metro se produce aproximadamente a los 30 y 50 segundos, momentos en los que se alcanzan los waypoints intermedios de la ruta. Esta desviación de un metro se produce al haber establecido como alcanzados los puntos al estar a una distancia menor o igual de un metro.

Otro factor que considerar es la orientación del vehículo, ya que en el plan establecido el vehículo requiere un yaw especifico en cada waypoint. Si se analiza la gráfica que muestra los valores de la orientación del ROV, se observa como el valor del yaw va variando dependiendo del waypoint siguiente, de forma que mantiene su valor correctamente hasta alcanzar dicho punto.

La altitud del vehículo tiene un ruido significativo que depende directamente de los valores que genera el SITL, por lo que no se podría en principio corregirlo. Aun teniendo este ruido se puede apreciar en la gráfica como el vehículo se va adaptando y va variando el valor de la altitud dependiendo de la que se ha requerido en la ruta.

7 CONCLUSIONES

La creación de un entorno de simulación para un vehículo operado de forma remota presentaba numerosos desafíos, pero también ofrece grandes oportunidades para el aprendizaje y la investigación. A pesar de las dificultades de la creación de una escena precisa se ha conseguido desarrollarla consiguiendo una simulación aceptable y funcional.

El desarrollo de este proyecto ha propiciado el avance de conocimientos sobre diferentes herramientas como máquinas virtuales, pruebas SITL, lenguaje de programación Python y otros recursos que son esenciales para la programación y simulación del vehículo. A través de diferentes pruebas, el sistema ha demostrado ser una herramienta valiosa a la hora de la experimentación, facilitando la implementación o prueba de diferentes parámetros sobre el ROV antes de realizarlo sobre una prueba física, así como analizar los datos obtenidos.

En un primer momento, el proyecto presentó retos significativos, como el aprendizaje del uso correcto de programación en Python, la configuración del SITL, y el diseño de interfaces de usuario. Sin embargo, el esfuerzo y dedicación invertidos han permitido alcanzar dichos objetivos.

El entorno de simulación diseñado permite a cualquier usuario realizar pruebas con el ROV, ya sea por su propia necesidad o por aprendizaje e interés sobre el campo de control y la programación. Este proyecto no solo cumple con sus metas educativas, sino que también promueve el avance constante en el desarrollo y control de ROVs y sistemas subacuáticos similares.

8 BIBLIOGRAFÍA

[1] Wikipedia, la enciclopedia libre. «Vehículo sumergible operado remotamente», Accedido 29 de mayo de 2024. https://es.wikipedia.org/w/index.php?title=Veh%C3%ADculo_sumergible_operado_remotamente&oldid=159240968.

[2] Armada Española. (2023). «RGM Junio 2023 Parte 05». Accedido 29 de mayo de 2024. https://armada.defensa.gob.es/archivo/rgm/2023/06/RGMJunio2023Parte05.pdf

[3] Blue Robotics. «BlueROV2 Buyer's Guide by Options». Accedido 18 de marzo de 2024. https://bluerobotics.com/learn/bluerov2-buyers-guide-by-options/.

[4] Wikipedia, la enciclopedia libre. «Ángulos de navegación». Accedido 19 de mayo de 2024. https://es.wikipedia.org/w/index.php?title=%C3%81ngulos_de_navegaci%C3%B3n&oldid=131672420.

[5] L. Walker y F. Giorgio Serchi. (2023). «Disturbance Preview for Non-Linear Model Predictive Trajectory Tracking of Underwater Vehicles in Wave Dominated Environments». Accedido 30 de junio de 2024.https://www.researchgate.net/publication/376495324_Disturbance_Preview_for_Non-Linear_Model_Predictive_Trajectory_Tracking_of_Underwater_Vehicles_in_Wave_Dominated_Environments

[6] Rydill, Louis J., y Roy Burcher. «Submarine hydrostatics». Concepts in Submarine Design (blog), 1994. Accedido 12 de junio de 2024. https://doi.org/10.1017/CBO9781107050211.004.

[7] Zayas Gato, F. (2020). «Diseño de controladores PID», Accedido 13 de junio de 2024. https://ruc.udc.es/dspace/bitstream/handle/2183/25824/Zayas_Gato_2020_Dise%C3%B1o_de_controladores_PID.pdf?sequence=2&isAllowed=y

[8] Christ, R. D., & Wernli Sr, R. L. (2007). «The ROV Manual: A User Guide for Observation-Class Remotely Operated Vehicles», Accedido 30 de mayo de 2024. https://ntcontest.ru/upload/iblock/52d/52d49cf6584942a6f5479d0402a35bf9.pdf

[9] Wikipedia, la enciclopedia libre. «Máquina virtual». Accedido 19 de marzo de 2024. https://es.wikipedia.org/w/index.php?title=M%C3%A1quina_virtual&oldid=155514070.

[10] «Ubuntu 20.04.6 LTS (Focal Fossa)». Accedido 19 de marzo de 2024. https://releases.ubuntu.com/focal/.

[11] «Overview · GitBook». Accedido 19 de marzo de 2024. https://www.ardusub.com/.

[12] Python.org. «What Is Python? Executive Summary». Accedido 21 de marzo de 2024. https://www.python.org/doc/essays/blurb/.

[13] «Free CAD Designs, Files & 3D Models | The GrabCAD Community Library». Accedido 17 de mayo de 2024. https://grabcad.com/library/bluerobotics-bluerov2-heavy-1/details?folder_id=4856422.

[14] GitHub. «Water wave generation concept». Accedido 17 de mayo de 2024. https://github.com/wezu/p3d_wave.

[15] Cuadras, C. (1989). «Distancias Estadisticas». Accedido 8 de junio de 2024. https://halweb.uc3m.es/esp/personal/personas/dpena/publications/castellano/1989EE_cuadras_coment.pdf

DOCUMENTO 2: ANEXOS

INDICE DE CONTENIDO

1 CODIGO DE PYTHON

1.1 Interfaz gráfica

```
#!/home/mario/anaconda3/bin/python3

# -*- coding: utf-8 -*-

#Mario Charlon Soldevilla


import subprocess


import sys, os, signal

from PyQt5 import QtCore, uic, QtGui

from PyQt5.QtCore import Qt ,QTimer

from PyQt5.QtCore import QObject, pyqtSignal

from PyQt5.QtGui import QPixmap

from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog, QMessageBox,
QSplashScreen, QLabel

import numpy as np

import time

import psutil


# BARRA DE PROGRESO


class ProgressBarUpdater(QObject):

    progressUpdated = pyqtSignal(int)
```

```
    def progreso(self):
        for i in range(100):
            # Tiempo entre pasos
            import time
            time.sleep(0.015)

            self.progressUpdated.emit(i + 1)  # Emite i + 1 para valores de 1 a 100

# PANTALLA DE CARGA

class SplashScreen(QSplashScreen):
    def __init__(self):
        super(QSplashScreen, self).__init__()
        uic.loadUi("/home/mario/Desktop/BlueROV2_sim/load_screen.ui",self)
        self.setWindowFlag(Qt.FramelessWindowHint)
        pixmap=QPixmap("/home/mario/Desktop/BlueROV2_sim/bg.png")
        self.setPixmap(pixmap)


        self.progressUpdater = ProgressBarUpdater()
        self.progressUpdater.progressUpdated.connect(self.updateProgressBar)


        from threading import Thread
        Thread(target=self.progressUpdater.progreso).start()

    def updateProgressBar(self, value):
```

```
        self.progressBar.setValue(value)


# APLICACION


class ui_app_sim(QMainWindow):
    def __init__(self):
        super().__init__()



        # ELEMENTOS


        uic.loadUi("/home/mario/Desktop/BlueROV2_sim/app_sim.ui", self)
        self.btn_path_SITL.clicked.connect(self.btn_path_SITL_clicked)
        self.btn_start_SITL.clicked.connect(self.btn_start_SITL_clicked)
        self.btn_stop_SITL.clicked.connect(self.btn_stop_SITL_clicked)
        self.btn_path_qgc.clicked.connect(self.btn_path_qgc_clicked)
        self.btn_start_qgc.clicked.connect(self.btn_start_qgc_clicked)
        self.btn_stop_qgc.clicked.connect(self.btn_stop_qgc_clicked)
        self.btn_path_sim3d.clicked.connect(self.btn_path_sim3d_clicked)
        self.btn_start_sim3d.clicked.connect(self.btn_start_sim3d_clicked)
        self.btn_stop_sim3d.clicked.connect(self.btn_stop_sim3d_clicked)
        self.btn_path_kml.clicked.connect(self.btn_path_kml_clicked)
        self.btn_start_kml.clicked.connect(self.btn_start_kml_clicked)
        self.btn_stop_kml.clicked.connect(self.btn_stop_kml_clicked)
        self.btn_exit.clicked.connect(self.btn_exit_clicked)
        self.btn_help.clicked.connect(self.alternateAction)
        self.btn_path_data.clicked.connect(self.btn_path_data_clicked)
```

```
        self.btn_export.clicked.connect(self.export)
        self.btn_stop_matlab.clicked.connect(self.btn_stop_matlab_clicked)
        self.text_help.hide()
        self.active_status = False



        self.procesos = []
        self.proces_names = []

        self.SITL_running = False
        self.sim3d_running = False
        self.kml_running = False
        self.export_running = False



    # Ruta SITL
    def btn_path_SITL_clicked(self):
        text = self.txt_path_SITL.toPlainText()
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        fileName, _ = QFileDialog.getOpenFileName(self,"Seleccione el path del ejecutable
sim_vehicle.py ","","All Files (*)", options=options)
        if not fileName:
            self.txt_path_SITL.setPlainText(text)
        else:
            self.txt_path_SITL.setPlainText(fileName)


    # Iniciar SITL
```

```
    def btn_start_SITL_clicked(self):
        if not self.SITL_running:
            self.SITL_running = True
            filename = self.txt_path_SITL.toPlainText()
            args = self.txt_args_SITL.toPlainText()
            if not args:
                args = '-v ArduSub -f vectored_6dof -L ISLA_DE_LA_TORRE --mavproxy-args "--
out udpbcast:localhost:14551" #--console --map'
                self.txt_args_SITL.setPlainText(args)

            command = 'konsole --workdir /home/mario/bluerov_simulation/ardupilot/ArduSub -e
' + filename + ' ' + args

            p = subprocess.Popen(command, shell = True, preexec_fn=os.setsid)
            self.procesos.append(p)
            self.proces_names.append('p_sitl')
        else:
            self.show_message("El SITL ya está en ejecución.")

    # Parar SITL
    def btn_stop_SITL_clicked(self):
        try:
            p_ind = self.proces_names.index('p_sitl')
            p_name = self.proces_names.pop(p_ind)
            p = self.procesos.pop(p_ind)

            parent = psutil.Process(p.pid)
```

```
            QApplication.processEvents()
            for child in parent.children(recursive=True):
                child.terminate()
            parent.terminate()

            for proc in psutil.process_iter():
                if "xterm" in proc.name():
                    proc.terminate()
            self.SITL_running = False

        except Exception as e:
            self.show_message("Error al detener el proceso.")

    # Ruta QgroundControl
    def btn_path_qgc_clicked(self):
        text = self.txt_path_qgroundcontrol.toPlainText()
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        fileName, _ = QFileDialog.getOpenFileName(self,"Seleccionar ejecutable de
QGroundControl","","All Files (*)", options=options)
        if not fileName:
            self.txt_path_qgroundcontrol.setPlainText(text)
        else:
            self.txt_path_qgroundcontrol.setPlainText(fileName)

    # Iniciar Qgc
    def btn_start_qgc_clicked(self):
        ruta=self.txt_path_qgroundcontrol.toPlainText()
```

```
        try:
            command = f'konsole -e  "{ruta}"'
            p = subprocess.Popen(command.split())
            self.procesos.append(p)
            self.proces_names.append('p_qgc')



        except Exception as e:
            print(f"Error al iniciar el proceso: {e}")

    # Parar Qgc
    def btn_stop_qgc_clicked(self):
        try:
            p_ind = self.proces_names.index('p_qgc')
            p_name = self.proces_names.pop(p_ind)
            p = self.procesos.pop(p_ind)
            QApplication.processEvents()
            os.kill(p.pid, signal.SIGTERM)
            p.wait()
        except Exception as e:
            self.show_message("Error al detener el proceso.")

    # Ruta Simulador
    def btn_path_sim3d_clicked(self):
        text = self.txt_path_sim3d.toPlainText()
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
```

```
        fileName, _ = QFileDialog.getOpenFileName(self,"Seleccionar ejecutable del simulador 3D","","All Files (*)", options=options)
        if not fileName:
            self.txt_path_sim3d.setPlainText(text)
        else:
            self.txt_path_sim3d.setPlainText(fileName)

    # Iniciar simulador
    def btn_start_sim3d_clicked(self):
        if not self.sim3d_running:
            self.sim3d_running = True

            command = 'konsole -e ' + self.txt_path_sim3d.toPlainText()
            p = subprocess.Popen(command.split())
            self.procesos.append(p)
            self.proces_names.append('p_sim3d')
        else:
            self.show_message("El simulador ya está en ejecución.")

    # Detener simulador
    def btn_stop_sim3d_clicked(self):
        try:
            p_ind = self.proces_names.index('p_sim3d')
            p_name = self.proces_names.pop(p_ind)
            p = self.procesos.pop(p_ind)
            QApplication.processEvents()
            os.kill(p.pid, signal.SIGTERM)
            self.sim3d_running = False
```

```
        except Exception as e:
            self.show_message("Error al detener el proceso.")


    # Ruta plan
    def btn_path_kml_clicked(self):
        text = self.txt_path_kml.toPlainText()
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        fileName, _ = QFileDialog.getOpenFileName(self,"Seleccionar archivo kml","","kml
Files (*.kml)", options=options)
        if not fileName:
            self.txt_path_kml.setPlainText(text)
        else:
            self.txt_path_kml.setPlainText(fileName)


    # Iniciar plan
    def btn_start_kml_clicked(self):
        if not self.kml_running:
            self.kml_running = True
            plan = self.txt_path_kml.toPlainText()
            # seguimiento_kml.exec(plan)
            command = 'konsole -e /home/mario/Desktop/BlueROV2_sim/seguimiento_kml.py ' +
plan
            p = subprocess.Popen(command.split())
            self.procesos.append(p)
            self.proces_names.append('p_kml')
        else:
```

```
            self.show_message("El seguimiento ya esta en ejecución.")


    # Detener plan
    def btn_stop_kml_clicked(self):
        try:
            p_ind = self.proces_names.index('p_kml')
            p_name = self.proces_names.pop(p_ind)
            p = self.procesos.pop(p_ind)


            QApplication.processEvents()
            os.kill(p.pid, signal.SIGTERM)
            self.kml_running = False
        except Exception as e:
            self.show_message("Error al detener el proceso.")


    # Ruta datos
    def btn_path_data_clicked(self):
        ruta_carpeta = "/home/mario/Desktop/BlueROV2_sim/simulaciones"
        ult_carpeta = self.obtener_ultima_carpeta(ruta_carpeta)
        archivo = "data_ahrs2.csv"
        if ult_carpeta:
            ruta_sin_archivo = os.path.join(ruta_carpeta, ult_carpeta)
            ruta_completa = os.path.join(ruta_sin_archivo, archivo)
            self.txt_path_matlab.setPlainText(ruta_completa)
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        fileName, _ = QFileDialog.getOpenFileName(self,"Seleccionar datos de
simulacion","","All Files (*)",  options=options)
```

```
        if not fileName:


            self.txt_path_matlab.setPlainText(ruta_completa)


        else:
            self.txt_path_matlab.setPlainText(fileName)


    # Exportar datos
    def export(self):
        if not self.export_running:
            self.export_running = True
            data_file = self.txt_path_matlab.toPlainText()
            path_data = os.path.dirname(data_file)
            print("",path_data)
            script = "analysis.m"
            ruta_script = "/home/mario/Desktop/BlueROV2_sim"
            command = f'konsole -e /usr/local/MATLAB/R2024a/bin/matlab -r
"cd(\'{ruta_script}\'); addpath(\'{path_data}\'); run(\'{script}\')"'
            p = subprocess.Popen(command, shell= True)
            self.procesos.append(p)
            self.proces_names.append('p_matlab')
        else:
            self.show_message("El proceso ya está en ejecución.")


    # Detener matlab
    def btn_stop_matlab_clicked(self):
        try:
```

```
            p_ind = self.proces_names.index('p_matlab')
            p_name = self.proces_names.pop(p_ind)
            p = self.procesos.pop(p_ind)

            # Usar psutil para matar el proceso hijo
            parent = psutil.Process(p.pid)
            for child in parent.children(recursive=True):
                child.terminate()
            parent.terminate()
            self.export_running = False
        except Exception as e:
            self.show_message("Error al detener el proceso.")


    # Desplegar texto ayuda
    def alternateAction(self):
        self.active_status = not self.active_status

        if self.active_status:
            text = self.text_help.text()
            self.text_help.setText(text)
            self.text_help.show()
        else:
            self.text_help.hide()

    # Obtener ultima carpeta
    def obtener_ultima_carpeta(self,ruta):
```

```
        carpetas = [nombre for nombre in os.listdir(ruta) if os.path.isdir(os.path.join(ruta,
nombre))]
        carpetas.sort(key=lambda carpeta: os.path.getctime(os.path.join(ruta, carpeta)))
        ultima_carpeta = carpetas[-1] if carpetas else None

        return ultima_carpeta

    # Abandonar aplicacion
    def btn_exit_clicked(self):
        title = 'Aviso'
        text = '¿Estás seguro de que deseas cerrar la aplicación?'
        msg = QMessageBox()
        msg.setIcon(QMessageBox.Question)
        msg.setText(text)
        msg.setWindowTitle(title)

        msg.setStandardButtons(QMessageBox.No | QMessageBox.Yes)
        confirm_button = msg.button(QMessageBox.Yes)
        confirm_button.setText('Sí')

        msg.setDefaultButton(QMessageBox.No)
        res= msg.exec_()

        if res == QMessageBox.Yes:
            for p in self.procesos:
                os.kill(p.pid, signal.SIGTERM)
            self.close()
```

```
    # Avisos al ejecutar archivos varias veces
    def show_message(self, message):
        msg = QMessageBox()
        msg.setIcon(QMessageBox.Information)
        msg.setText(message)
        msg.setWindowTitle("Aviso")
        msg.setStandardButtons(QMessageBox.Ok)
        msg.exec_()

    # Detener subprocesos al cerrar app (sin utilizar "salir")
    def closeEvent(self, event):
        for p in self.procesos:
            pid = p.pid

            parent = psutil.Process(pid)

            children = parent.children(recursive=True)
            for child in children:
                child.terminate()
            parent.terminate()

            for proc in psutil.process_iter():
                if "xterm" in proc.name():
                    proc.terminate()

        event.accept()
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)

    splash= SplashScreen()
    splash.show()

    window=ui_app_sim()

    QTimer.singleShot(1900, window.show)
    QTimer.singleShot(1899, splash.close)

    app.exec_()
    app.quit()
```

1.2 Simulador 3D

```
#!/home/mario/anaconda3/bin/python3
# -*- coding: utf-8 -*-
#Mario Charlon Soldevilla
# from panda3d.core import loadPrcFile
# loadPrcFile("config/conf.prc")

from panda3d.core import loadPrcFileData
confVars = """
win-size 640 480
window-title Bluerov2 Simulator
show-frame-rate-meter True
sync-video False
clock-frame-rate 60
"""
loadPrcFileData("", confVars)



# from panda3d.core import ConfigVariableManager
# ConfigVariableManager.getGlobalPtr().listVariables



from math import pi, sin, cos
```

```
from direct.showbase.ShowBase import ShowBase


from direct.task import Task

from panda3d.core import PointLight, AmbientLight, DirectionalLight, Fog, TextNode,
TransparencyAttrib

from direct.gui.OnscreenImage import OnscreenImage

from panda3d.core import CardMaker, TransparencyAttrib

from panda3d.core import NodePath, GraphicsOutput

from panda3d.core import TextureStage, Vec3,GeoMipTerrain, WindowProperties

from panda3d.core import *

from direct.gui.OnscreenText import OnscreenText

from direct.interval.IntervalGlobal import *



from lightmanager import LightManager

import random, cv2, socket

import numpy as np

import pandas as pd


import mavlink_lib

import utils_lib



class MyApp(ShowBase):

    def __init__(self):

        ShowBase.__init__(self)

        random.seed()
```

```
self.ts = 1.0/60
xy_scale = 1 # factor de escala xy (escala 1 = 100 metros)
water_depth = 28

base.disableMouse()

color_water=(0.1, 0.25,0.4,1)
color_luz = (0.5, 0.5, 1, 1)

base.setBackgroundColor(color_water)

#setup
size=512 # tamaño del buffer del agua
self.update_speed=1.0/60.0 #wave rate (1/60=60fps)

# niebla
niebla = Fog("niebla")
niebla.setColor(color_luz)
#niebla.setExpDensity(0.01)
niebla.setExpDensity(0.04)
#Aplicar niebla
self.render.setFog(niebla)

# crear iluminacion
# luz direccional
dlight = DirectionalLight("luz direccional")
dlight.setColor(color_luz)
```

```
dlight.setDirection((0, 45, -45))
self.dlnp = self.render.attachNewNode(dlight)
# luz ambiental
alight = AmbientLight("luz ambiental")
alight.setColor(color_luz)
self.alnp = self.render.attachNewNode(alight)



# cargar fondo
fondo = self.loader.loadModel("assets/fondo.glb")
fondo.clearModelNodes()
fondo.flattenStrong()
fondo.reparentTo(self.render)
fondo.setPos(0, 0, -water_depth +18)
fondo.setHpr(0, 0, 0)
fondo.setScale(xy_scale)
fondo.hide()



# cargar rocas
rock1 = self.loader.load_model("assets/roca1.glb")
rock1.reparentTo(fondo)
rock1.clearModelNodes()
rock1.flattenStrong()
rock1.setPos(0, 0, 0)
rock1.setScale(1, 1, 0.5)
```

```
rock1.setLight(self.dlnp)
rock1.setLight(self.alnp)
rock1.showThrough()
for x in range(-50*xy_scale, 50*xy_scale, 5):
    for y in range(-50*xy_scale, 50*xy_scale, 5):
        nueva_roca_nodo = fondo.attachNewNode("new-block-placeholder")
        escala = (1+2*random.random(), 1+2*random.random(), 0.5)
        nueva_roca_nodo.setScale(escala)
        posx = x+10*(random.random()-0.5)
        posy = y+10*(random.random()-0.5)
        posz = 0
        nueva_roca_nodo.setPos(posx, posy, posz)
        nueva_roca_nodo.setHpr(random.random()*360, 0, 0)
        nueva_roca_nodo.setLight(self.dlnp)
        nueva_roca_nodo.setLight(self.alnp)
        rock1.instanceTo(nueva_roca_nodo)


rock2 = self.loader.load_model("assets/roca2.glb")
rock2.reparentTo(fondo)
rock2.clearModelNodes()
rock2.flattenStrong()
rock2.setPos(0, 0, 0)
rock2.setScale(0.05, 0.05, 0.05)
rock2.setLight(self.dlnp)
rock2.setLight(self.alnp)
rock2.showThrough()
for x in range(-50*xy_scale, 50*xy_scale, 10):
```

```
        for y in range(-50*xy_scale, 50*xy_scale, 10):
            nueva_roca_nodo = fondo.attachNewNode("new-block-placeholder")
            escala = (1+2*random.random(), 1+2*random.random(), 0.5)
            nueva_roca_nodo.setScale(escala)
            posx = x+10*(random.random()-0.5)
            posy = y+10*(random.random()-0.5)
            posz = 0
            nueva_roca_nodo.setPos(posx, posy, posz)
            nueva_roca_nodo.setHpr(random.random()*360, 0, 0)
            nueva_roca_nodo.setLight(self.dlnp)
            nueva_roca_nodo.setLight(self.alnp)
            rock2.instanceTo(nueva_roca_nodo)

    rock5 = self.loader.load_model("assets/roca5.glb")
    rock5.reparentTo(fondo)
    rock5.clearModelNodes()
    rock5.flattenStrong()
    rock5.setPos(0, 0, 0)
    rock5.setScale(0.05)
    rock5.setLight(self.dlnp)
    rock5.setLight(self.alnp)
    rock5.showThrough()
    for x in range(-50*xy_scale, 50*xy_scale, 15):
        for y in range(-50*xy_scale, 50*xy_scale, 15):
            nueva_roca_nodo = fondo.attachNewNode("new-block-placeholder")
            escala = (1+2*random.random(), 1+2*random.random(), 0.5)
            nueva_roca_nodo.setScale(escala)
```

```
        posx = x+10*(random.random()-0.5)
        posy = y+10*(random.random()-0.5)
        posz = 0
        nueva_roca_nodo.setPos(posx, posy, posz)
        #nueva_roca_nodo.setHpr(random.random()*360, 0, 0)
        nueva_roca_nodo.setLight(self.dlnp)
        nueva_roca_nodo.setLight(self.alnp)
        rock5.instanceTo(nueva_roca_nodo)

fondo.clearModelNodes()
fondo.flattenStrong()

# cargar bluerov
self.bluerov = self.loader.loadModel("assets/bluerov23.glb")
self.bluerov.clearModelNodes()  # simplificar mesh
self.bluerov.flattenStrong()  # simplificar mesh
self.bluerov.reparentTo(self.render)

self.bluerov.setHpr(90, 90, 0)
self.bluerov.setPos(0, 0, 29)
self.bluerov.setScale(1)
self.bluerov.setLight(self.dlnp)
self.bluerov.setLight(self.alnp)

#skybox
skybox=loader.loadModel('skybox/skybox')
skybox.reparentTo(self.render)
skybox.setPos(0, 0, 40)
```

```
        skybox.setScale(50*xy_scale)



        # conexion
        self.conexion = mavlink_lib.MAVlink_connection('udpin:localhost:14552')
        # self.conexion.request_message_interval("LOCAL_POSITION_NED", self.ts)
        self.conexion.request_message_interval('AHRS2', self.ts)
        self.conexion.request_message_interval('SYSTEM_TIME', self.ts)
        self.actual_time = self.conexion.get_info('SYSTEM_TIME',
bloqueo=True).time_boot_ms*1e-3

        # posicion y orientacion inicial
        ahrs2 = self.conexion.get_info('AHRS2', bloqueo = True)
        pos = utils_lib.geodetic_to_geocentric(ahrs2.lat*1e-7, ahrs2.lng*1e-7, ahrs2.altitude)
        self.actual_pos = Vec3(pos[0], pos[1], pos[2])
        self.actual_orient = Vec3(ahrs2.yaw, ahrs2.pitch, ahrs2.roll)*180/pi
        self.bluerov.setPos(Vec3(0, 0, ahrs2.altitude))
        self.bluerov.setHpr(Vec3(0,0, 0))

        # camara principal
        self.camera.reparentTo(self.render)
        self.camera.setPos(self.bluerov.getPos() + Vec3(-6, 0, 3))
        self.camera.lookAt(self.bluerov)
        self.camera.setHpr(180,90,0)

        # camara Bluerov
        wp = WindowProperties()
```

```
        wp.setSize(320, 180)  # tamaño de la nueva ventana
        wp.setOrigin(0, 0)
        parent_window=base.win.getWindowHandle()
        wp.setParentWindow(parent_window)
        self.win2 = self.openWindow(props=wp)
        self.win2.setClearColorActive(True)
        self.win2.setClearColor((color_water))

        self.win2.getDisplayRegion(0).setClearColor(color_water)
        self.win2.getDisplayRegion(0).setClearColorActive(True)
        self.last_window_size = (self.win.getXSize(), self.win.getYSize())
        self.bluerov_cam = base.camList[1]

        self.bluerov_cam.reparentTo(self.bluerov)

        # posicion de la camara
        self.cam_x=0
        self.cam_y=3
        self.cam_z=0
        self.bluerov_cam.setPos(self.bluerov.getPos() - Vec3(self.cam_x, self.cam_y,
self.cam_z))
        self.bluerov_cam.setHpr(180,90,0)



        self.altitude_text = OnscreenText(
            text="",
            pos=(0.8, -0.90),  # Posición en la ventana
            scale=0.07,  # Escala del texto
```

```
        align=TextNode.ARight,  # Alinear el texto a la Dcha
        fg=(1, 1, 1, 1),  # Color del texto
        bg=(0, 0, 0, 0),  # Color de fondo transparente
        shadow=(0, 0, 0, 1),  # Sombra del texto

    )
    self.roll_text = OnscreenText(
        text="",
        pos=(1.2, -0.82),
        scale=0.07,
        align=TextNode.ARight,
        fg=(1, 1, 1, 1),
        bg=(0, 0, 0, 0),
        shadow=(0, 0, 0, 1),
    )
    self.pitch_text = OnscreenText(
        text="",
        pos=(1.2, -0.90),
        scale=0.07,  #
        align=TextNode.ARight,
        fg=(1, 1, 1, 1),
        bg=(0, 0, 0, 0),
        shadow=(0, 0, 0, 1),

    )

    self.yaw_text = OnscreenText(
```

```
        text="",
        pos=(1.2, -0.98),
        scale=0.07,
        align=TextNode.ARight,
        fg=(1, 1, 1, 1),
        bg=(0, 0, 0, 0),
        shadow=(0, 0, 0, 1),
    )



    # mover el bluerov
    self.data_sim = []
    self.taskMgr.add(self.update, "update")
    # Añadir una tarea para verificar cambios en el tamaño de la ventana principal
    self.taskMgr.add(self.checkWindowSize, "checkWindowSize")

    # Superficie del agua
    #wave source buffer
    self.wave_source=self.makeBuffer(size=256, texFilter=Texture.FTNearest)
    self.wave_source['quad'].setColor(0,0,0, 0)

    self.wave_source['quad'].setZ(-10)



    #make the ping-pong buffer to draw the water waves
    #shader 2 makes no use of the wave_source texture, appart from that they are identical
```

```
shader1=Shader.load(Shader.SL_GLSL,'v.glsl', 'make_wave_f.glsl')
shader2=Shader.load(Shader.SL_GLSL,'v.glsl', 'make_wave2_f.glsl')
self.ping=self.makeBuffer(shader1, size)
self.pong=self.makeBuffer(shader2, size)
self.ping['quad'].setShaderInput("size",float(size))
self.pong['quad'].setShaderInput("size",float(size))
self.ping['quad'].setTexture(self.pong['tex'])
self.pong['quad'].setTexture(self.ping['tex'])
self.ping['quad'].setShaderInput('startmap', self.wave_source['tex'])
self.ping['buff'].setActive(True)
self.pong['buff'].setActive(False)

#some vars to track the state of things
self.time=0
self.state=0

#set lights
#reusing some of my old stuff, no mind the light manager
l=LightManager()
self.sun=l.addLight(pos=(128.0, 300.0, 50.0), color=(0.9, 0.9, 0.9), radius=500.0)



#preview
cm = CardMaker("plane")
cm.setFrame(0, 128, 0, 128)
self.water_plane=self.render.attachNewNode(cm.generate())
self.water_plane.lookAt(0, 0, -1)
```

```
        self.water_plane.setShader(Shader.load(Shader.SL_GLSL,'water_v.glsl','water_f.glsl'))
        self.water_plane.setShaderInput("size",float(size))

self.water_plane.setShaderInput("normal_map",loader.loadTexture("textures/waves.jpg"))
        self.water_plane.setTexture(self.pong['tex'])
        self.water_plane.hide(BitMask32.bit(1))
        self.water_plane.setScale(xy_scale)
        self.water_plane.setTransparency(TransparencyAttrib.MAlpha)
        self.water_plane.reparentTo(self.render)
        self.water_plane.setPos(-64, 64, water_depth-18)
        self.water_plane.setHpr(0, 90, 0)
        self.water_plane.setTwoSided(True)

        self.makeWaterBuffer()
        self.water_plane.flattenStrong()



    def makeWaterBuffer(self):
        self.water_buffer = base.win.makeTextureBuffer("water", 512, 512)
        self.water_buffer.setClearColor(base.win.getClearColor())
        self.water_buffer.setSort(-1)
        self.water_camera = base.makeCamera(self.water_buffer)
        self.water_camera.reparentTo(self.render)
        self.water_camera.node().setLens(base.camLens)
        self.water_camera.node().setCameraMask(BitMask32.bit(1))
        #Create this texture and apply settings
        reflect_tex = self.water_buffer.getTexture()
```

```
        reflect_tex.setWrapU(Texture.WMClamp)
        reflect_tex.setWrapV(Texture.WMClamp)

        #Create plane for clipping and for reflection matrix
        self.clip_plane = Plane(Vec3(0, 0, 1), Point3(0, 0,0.5)) # a bit off, but that's how it
should be
        clip_plane_node = self.render.attachNewNode(PlaneNode("water", self.clip_plane))
        tmp_node = NodePath("StateInitializer")
        tmp_node.setClipPlane(clip_plane_node)
        tmp_node.setAttrib(CullFaceAttrib.makeReverse())
        self.water_camera.node().setInitialState(tmp_node.getState())
        self.water_plane.setShaderInput('camera',self.water_camera)
        self.water_plane.setShaderInput("reflection",reflect_tex)


    def makeBuffer(self, shader=None, size=256, texFilter=Texture.FTLinearMipmapLinear):
        root=NodePath("bufferRoot")
        tex=Texture()
        tex.setWrapU(Texture.WMClamp)
        tex.setWrapV(Texture.WMClamp)
        tex.setMagfilter(texFilter)
        tex.setMinfilter(texFilter)
        props = FrameBufferProperties()
        props.setRgbaBits(16, 16, 0, 0)
        props.setSrgbColor(False)
        props.setFloatColor(True)
        buff=base.win.makeTextureBuffer("buff", size, size, tex, fbp=props)
```

```
        #the camera for the buffer
        cam=base.makeCamera(win=buff)
        #cam.reparentTo(root)
        #cam.setPos(size/2,size/2,100)
        cam.setP(-90)
        lens = OrthographicLens()
        lens.setFilmSize(size, size)
        cam.node().setLens(lens)
        #plane with the texture
        cm = CardMaker("plane")
        cm.setFrame(0, size, 0, size)
        quad=root.attachNewNode(cm.generate())
        quad.lookAt(0, 0, -1)
        if shader:
            ShaderAttrib.make(shader)
            quad.setAttrib(ShaderAttrib.make(shader))
        #return all the data in a dict
        return{'root':root, 'tex':tex, 'buff':buff, "cam":cam, 'quad':quad}


    def setWaveCaster(self, node):
        #the pahnatom node needs to be clipped at the water level
        clip_plane = Plane(Vec3(0, 0, -1), Point3(0, 0, 1.0))
        plane_node = PlaneNode("clip")
        plane_node.setPlane(clip_plane)
        plane_path = self.render.attachNewNode(plane_node)
        node.setClipPlane(plane_path)
        #we also remove the texture, and paint it red
        node.setTextureOff(1)
```

```
        node.setColor(1.0, 0.0, 0.0, 0.0)
        node.setLightOff()
        #finaly draw it inside out
        node.setAttrib(CullFaceAttrib.make(CullFaceAttrib.MCullCounterClockwise))



    # actualiza la segunda ventana al maximizar o minimizar la ventana principal
    def checkWindowSize(self, task):
            current_size = (self.win.getXSize(), self.win.getYSize())
            is_fullscreen = self.win.getProperties().getFullscreen()

            if current_size != self.last_window_size and not is_fullscreen:
                main_width = self.win.getXSize()
                main_height = self.win.getYSize()
                new_width = int(main_width * 0.25)
                new_height = int(main_height * 0.25)

                wp = WindowProperties()
                wp.setSize(new_width, new_height)
                self.win2.requestProperties(wp)

                self.last_window_size = current_size

            return task.cont

    # funcion para actualizar escena
```

```
    def update(self, task):
        dt = globalClock.getDt()

        # Obtener el tiempo actual
        last_time = self.actual_time
        actual_time = self.conexion.get_info('SYSTEM_TIME', bloqueo=False)
        if actual_time: #si hay nuevos datos
            self.actual_time = actual_time.time_boot_ms * 1e-3
            sampling_time = self.actual_time - last_time
            if sampling_time > self.ts + 0.1:
                self.conexion.request_message_interval('AHRS2', self.ts)
                self.conexion.request_message_interval('SYSTEM_TIME', self.ts)

        # Actualizar la orientación
        ahrs2 = self.conexion.get_info('AHRS2', bloqueo=False)

        if ahrs2: #si hay nuevos datos
            orient = Vec3(-ahrs2.yaw + pi * 0.5, ahrs2.pitch + pi * 0.5, ahrs2.roll) * 180 / pi
#Radianes->grados
            self.bluerov.setHpr(orient)

            # Actualizar la posición
            last_pos = self.actual_pos
            actual_pos = utils_lib.geodetic_to_geocentric(ahrs2.lat * 1e-7, ahrs2.lng * 1e-7,
ahrs2.altitude) # coordenadas geodesicas a geocentricas
            self.actual_pos = Vec3(actual_pos[0], actual_pos[1], actual_pos[2])
            delta_pos = last_pos - self.actual_pos
            new_pos = self.bluerov.getPos() + delta_pos # dependiendo de la diferencia entre la
ultima posicion se calcula la nueva
```

```
        new_pos.z = ahrs2.altitude
        self.bluerov.setPos(new_pos)

        # Actualiza el texto en pantalla con el valor de la altitud
        formatted_data = "Altitud:{:.2f}".format(ahrs2.altitude)
        self.altitude_text.setText(formatted_data)

        # Actualiza el texto en pantalla con el valor de roll
        formatted_data1 = "Roll:{:.2f}".format(ahrs2.roll*180/pi)
        self.roll_text.setText(formatted_data1)


        # Actualizar el texto en pantalla con el valor de pitch
        formatted_data2 = "Pitch:{:.2f}".format(ahrs2.pitch*180/pi)
        self.pitch_text.setText(formatted_data2)


        # Actualizar el texto en pantalla con el valor de yaw
        formatted_data3 = "Yaw:{:.2f}".format(ahrs2.yaw*180/pi)
        self.yaw_text.setText(formatted_data3)


    # Actualizar la posición de la cámara
    self.camera.setPos(self.bluerov.getPos() + Vec3(-6, 0, 3))
    self.camera.lookAt(self.bluerov)
```

```
        return Task.cont



if __name__ == "__main__":
    app = MyApp()
    app.run()
```

1.3 Seguimiento

```
#!/home/mario/anaconda3/bin/python3
# -*- coding: utf-8 -*-
#Mario Charlon Sodlevilla

import sys
import mavlink_lib
import numpy as np
import utils_lib
import os
import csv
from datetime import datetime

def exec(kml_file = None):

    # parametros simulacion y control
```

```
    ts = 1.0/60    # periodo de muestreo deseado
    sampling_time = ts   # periodo de muestreo real
    dist_to_waypoin_reached = 1.0   # distancia para considerar waypoint alcanzado
    cruise_speed = 0.5
    max_yaw_error = np.deg2rad(15)  # error de yaw para permitir avance en grados

    # crear conexion
    print('\ncreando conexion ...')
    bluerov2 = mavlink_lib.MAVlink_connection('udpin:localhost:14551')
    # bluerov2 = mavlink_lib.MAVlink_connection('tcp:127.0.0.1:5760')
    if bluerov2.conexion == None:
        return

    # cambiar periodo de muestreo
    bluerov2.request_message_interval('AHRS2', ts)
    bluerov2.request_message_interval('SYSTEM_TIME', ts)

    # tiempo actual
    fecha_actual = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    # Crear el nombre de la carpeta de la simulación
    nombre_carpeta = f"simulacion_{fecha_actual}"
    # Crear la ruta completa para la carpeta de la simulación

    ruta_carpeta =
os.path.join("/home/mario/Desktop/Prueba_panda3d/simulaciones",nombre_carpeta)
    # Crear la carpeta de la simulación
    os.makedirs(ruta_carpeta)
```

```
ruta_archivo = os.path.join(ruta_carpeta, 'data_ahrs2.csv')
# Crear el archivo antes del bucle
with open(ruta_archivo, 'w+') as archivo:
    pass  # Solo crear archivo
ruta_archivo_wp = os.path.join(ruta_carpeta, 'waypoints.csv')

actual_time = bluerov2.get_info('SYSTEM_TIME', bloqueo = True).time_boot_ms*1e-3
ahrs2 = bluerov2.get_info('AHRS2', bloqueo = True)




# rov_roll = ahrs2.roll
# rov_pitch = ahrs2.pitch
rov_yaw = ahrs2.yaw
rov_lat = ahrs2.lat*1e-7
rov_lon = ahrs2.lng*1e-7
rov_alt = ahrs2.altitude

# controladores PID
pid_yaw = utils_lib.PID_control(0.5, 0.01, 0.01, actual_time)
pid_lateral = utils_lib.PID_control(0.5, 0.01, 0.01, actual_time)
pid_throttle = utils_lib.PID_control(0.5, 0.01, 0.01, actual_time)
pid_fordward = utils_lib.PID_control(0.5, 0.01, 0.01, actual_time)

# obtener waypoints
if not kml_file:
```

```
        kml_file = '/home/mario/bluerov_simulation/pymavlink/prueba.kml'
    waypoints = utils_lib.kml2list(kml_file)
    waypoints = np.vstack((waypoints, waypoints[0]))
    waypoints = np.delete(waypoints, 0, 0)
    waypoints[0] = [rov_lat, rov_lon, rov_alt] #Se asigna el primer waypoint con la posicion del
ROV
    waypoints[-1] = waypoints[0] # idem para el ultimo waypoint
    n_wp = len(waypoints)



    # Guardar waypoints en archivo CSV
    with open(ruta_archivo_wp, mode='w', newline='') as file:
        writer = csv.writer(file)
        for waypoint in waypoints:
            writer.writerow(waypoint)

    for wp in waypoints:
        print(wp)
    print('# waypoints =', len(waypoints))


    print('\ncomenzando maniobras...')


    # cambiar a modo MANUAL
    bluerov2.set_mode('MANUAL')


    # armar
    bluerov2.set_arm_disarm(1)
```

```
# arrays para guardar datos
#time, obj_yaw, rov_yaw, obj_x, rov_x, obj_y, rov_y, obj_z, rov_z

data_sim = np.zeros([0, 14])
# para cada waypoint
for n in range(1,n_wp):
    # waypoint anterior
    last_wp = waypoints[n-1]
    last_lat = last_wp[0]
    last_lon = last_wp[1]
    last_alt = last_wp[2]
    last_wp_xyz = utils_lib.geodetic_to_geocentric(last_lat, last_lon, last_alt)

    # waypoint siguiente
    next_wp = waypoints[n]
    next_lat = next_wp[0]
    next_lon = next_wp[1]
    next_alt = next_wp[2]
    next_wp_xyz = utils_lib.geodetic_to_geocentric(next_lat, next_lon, next_alt)

    # velocidad y posicion del objetivo movil
    obj_velocity = next_wp_xyz - last_wp_xyz
    dist_obj_to_wp = np.linalg.norm(obj_velocity)
    obj_velocity /= (dist_obj_to_wp+1e-9)
    obj_speed = 0.0
    obj_xyz = last_wp_xyz
```

```
# obj_lat = last_lat
# obj_lon = last_lon
# obj_alt = last_alt

# yaw del objetivo
obj_yaw = np.arctan2(obj_velocity[1], -obj_velocity[0])

# distancia del rov al siguiente waypoint
dist_rov_to_next_wp = dist_obj_to_wp

# reset de los pid
pid_yaw.reset(actual_time)
pid_lateral.reset(actual_time)
pid_throttle.reset(actual_time)
pid_fordward.reset(actual_time)

# while True:
while dist_rov_to_next_wp > dist_to_waypoin_reached:

    print('\n\nwaypoint', n, '/', n_wp-1)

    # print(bluerov2.get_info('GLOBAL_POSITION_INT', bloqueo = True))
    # print(bluerov2.get_info('LOCAL_POSITION_NED', bloqueo = True))
    # print(bluerov2.get_info('AHRS', bloqueo = True))
    # print(bluerov2.get_info('AHRS2', bloqueo = True))
    # print(bluerov2.get_info('ATTITUDE', bloqueo = True))
```

```
        # tiempo actual
        last_time = actual_time
        actual_time = bluerov2.get_info('SYSTEM_TIME', bloqueo =
True).time_boot_ms*1e-3
        sampling_time = actual_time - last_time

        # periodo de muestreo
        if sampling_time > ts + 0.1:
            bluerov2.request_message_interval('AHRS2', ts)
            bluerov2.request_message_interval('SYSTEM_TIME', ts)
            bluerov2.request_message_interval('VFR_HUD', ts)

        # posicion del objetivo
        obj_xyz += obj_speed*obj_velocity*sampling_time
        # distancia del objetivo al waypoint siguiente
        dist_obj_to_wp = np.linalg.norm(next_wp_xyz - obj_xyz)

        # posicion y orientacion actuales del rov
        ahrs2 = bluerov2.get_info('AHRS2', bloqueo = True)
        # rov_roll = ahrs2.roll
        # rov_pitch = ahrs2.pitch
        rov_yaw = ahrs2.yaw
        rov_lat = ahrs2.lat*1e-7
        rov_lon = ahrs2.lng*1e-7
        rov_alt = ahrs2.altitude
        rov_xyz = utils_lib.geodetic_to_geocentric(rov_lat, rov_lon, rov_alt)

        # error de coordenadas geograficas del rov
```

```
# error_lat = next_lat - rov_lat
# error_lon = next_lon - rov_lon
# error_alt = next_alt - rov_alt

# distancia de la posicion actual del rov al waypoints siguiente
vector_rov_to_next_wp_xyz = next_wp_xyz - rov_xyz
dist_rov_to_next_wp = np.linalg.norm(vector_rov_to_next_wp_xyz)

# error de yaw entre el objetivo y el rov
error_yaw = obj_yaw - rov_yaw
if error_yaw > np.pi:
    error_yaw -= 2*np.pi
elif error_yaw < -np.pi:
    error_yaw += 2*np.pi

# distancia del rov al objetivo
# vector_rov_to_obj_xyz = obj_xyz - rov_xyz
# dist_rov_to_obj = np.linalg.norm(vector_rov_to_obj_xyz)
# velocidad del objetivo
if dist_obj_to_wp < cruise_speed:
    obj_speed = dist_obj_to_wp*np.cos(error_yaw)
elif obj_speed < cruise_speed:
    obj_speed += 0.1*np.cos(error_yaw)**4


# orientacion del rov respecto del waypoint siguiente
```

```
        ang_rov_to_next_wp = np.arctan2(vector_rov_to_next_wp_xyz[1], -
vector_rov_to_next_wp_xyz[0])
        # error de orientacion del rov respecto del yaw del objetivo
        angular_offset = ang_rov_to_next_wp - obj_yaw
        lateral_offset = np.linalg.norm(vector_rov_to_next_wp_xyz[:2])*np.sin(angular_offset)
        if lateral_offset > np.pi:
            lateral_offset = np.pi
        elif lateral_offset < -np.pi:
            lateral_offset = -np.pi

        # canales: pitch = 0, roll = 1, throttle = 2, yaw = 3, fordward = 4, lateral = 5
        error_throttle = obj_xyz[2] - rov_xyz[2]
        pid_throttle.update(error_throttle, actual_time)
        pwm_throttle = 1500 + int(pid_throttle.out*400)*(np.abs(error_yaw) <
max_yaw_error)

        pid_yaw.update(error_yaw, actual_time)
        pwm_yaw = 1500 + int(pid_yaw.out*400)

        pid_lateral.update(lateral_offset, actual_time)
        pwm_lateral = 1500 + int(pid_lateral.out*400)*(np.abs(error_yaw) < max_yaw_error)

        error_fordward = np.linalg.norm(obj_xyz[:2] - rov_xyz[:2])
        if error_fordward > 0.1:
            pid_fordward.update(error_fordward, actual_time)
            pwm_fordward = 1500 + int(pid_fordward.out*400)*(np.abs(error_yaw) <
max_yaw_error)
        else:
            pwm_fordward = 1500
```

```
        # canales: pitch = 0, roll = 1, throttle = 2, yaw = 3, fordward = 4, lateral = 5
        channels_pwm = [65535]*18
        channels_pwm[2:6] = [pwm_throttle, pwm_yaw, pwm_fordward, pwm_lateral]
        bluerov2.set_rc_channel_list(channels_pwm)

        # guardar datos
        data_sim=np.vstack([ data_sim, [actual_time, ahrs2.roll, ahrs2.pitch, rov_yaw,
rov_alt, rov_lat, rov_lon, error_yaw,
                                obj_xyz[0], rov_xyz[0], obj_xyz[1], rov_xyz[1], obj_xyz[2],
rov_xyz[2]]])

        error_yaw_deg = np.rad2deg(error_yaw)
        print(f'sampling_time (s): {sampling_time:7.3f}')
        print(f'error_yaw (º): {error_yaw_deg:7.3f}')
        print(f'lateral_offset (m): {lateral_offset:7.3f}')
        print(f'dist_rov_to_next_wp (m): {dist_rov_to_next_wp:7.3f}')
        # print(f'dist_to_target: {dist_to_target:7.3f} metros')
        print(f'xyz_error (m): {vector_rov_to_next_wp_xyz[0]:7.3f}
{vector_rov_to_next_wp_xyz[1]:7.3f}  {vector_rov_to_next_wp_xyz[2]:7.3f}')



    print('\n\n')

    # expotar datos a fichero
    for actual_time, roll, pitch, yaw, altitude, lat, lng, error_yaw, next_wp_x, rov_x, next_wp_y,
rov_y, next_wp_z, rov_z in data_sim:
```

```
        with open(ruta_archivo, 'a+') as archivo:
            archivo.write(f"{actual_time} {roll} {pitch} {yaw} {altitude} {lat} {lng} {error_yaw} {next_wp_x} {rov_x} {next_wp_y} {rov_y} {next_wp_z} {rov_z}\n")



    # parar motores
    bluerov2.set_rc_stop()

    # desarmar
    bluerov2.set_arm_disarm(0)

    # modo 'ALT_HOLD'
    bluerov2.set_mode('ALT_HOLD')

    # Cerrar conexion
    bluerov2.close_connection()

    # fin
    print("\nPrograma finalizado\n")



if __name__ == "__main__":
    if len(sys.argv) == 1:
        exec()
    else:
        exec(sys.argv[1])
```

1.4 MAVlink y utilidades

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#Mario Charlon Soldevilla
import time
import sys
from pymavlink import mavutil


class MAVlink_connection:
    def __init__(self, ip_dir):
        """
        - Crea una conexion MAVlink con el BlueRov2, con el protocolo, direccion y puerto especificados,
        y le pone en modo 'STABILIZE', con los pwm de los propulsores a 1500,
        y desarmado

        Parameters
        ----------
        ip : string
            protocolo, direccion ip y puerto de la conexion
```

```
        Returns
        -------
        objeto de la clase MAVlink_connection

        ej: conexion = MAVlink_connection('udpin:localhost:14551')

        """

        # crear conexion
        self.conexion = self.create_connection(ip_dir)

    def create_connection(self, ip):
        """
        Inicia una conexion mavutil.mavlink_connection con el protocolo, direccion y puerto
especificados

        Parameters
        ----------
        ip : string
            protocolo, direccion ip y puerto de la conexion. ej: 'udpin:localhost:14551'

        Returns
        -------
        conexion : mavutil.mavlink_connection
            conexion con el protocolo, direccion y puerto especificados

        """
        for n in range(10):
```

```
            print("estableciendo conexion intento " + str(n+1) + "...")
            # establecer conexion
            conexion = mavutil.mavlink_connection(ip)
            # esperar latido
            conexion.wait_heartbeat(blocking = True, timeout = 1)
            if conexion.target_system == 1 and conexion.target_component == 0:
                print("OK: heartbeat recibido de (system %u, component %u)" %
(conexion.target_system, conexion.target_component))
                break
            else:
                conexion = None
        if conexion == None:
            print('¡ IMPOSIBLE CREAR CONEXION !')
        return conexion

    def set_arm_disarm(self, arm_disarm):
        """
        Arma/desarma el BlureRov2

        Parameters
        ----------
        arm_disarm : bool
            0 = desarmar, 1 = armar.

        Returns
        -------
        None.
```

```
        """
        # armar/desarmar
        self.conexion.mav.command_long_send(
            self.conexion.target_system,
            self.conexion.target_component,
            mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM,
            0, arm_disarm, 0, 0, 0, 0, 0, 0)

        if arm_disarm:
            print("\n armando ...", end = " ")
            self.conexion.motors_armed_wait()
        else:
            print("\n desarmando ...", end = " ")
            self.conexion.motors_disarmed_wait()

        print('OK')

    def set_mode(self, modo):
        """
        Establece el modo de funcionamiento del BlureRov2. Los modos posibles son:
            'ALT_HOLD', 'POSHOLD', 'STABILIZE', 'MANUAL', 'AUTO', 'GUIDED',
            'CIRCLE', 'SURFACE', 'ACRO'

        Parameters
        ----------
        modo : string
            modo de funcionamiento.
```

```
    Returns
    -------
    None.

    """
    # Check if mode is available
    if modo not in self.conexion.mode_mapping():
        print('\n modo desconocido: {}'.format(modo))
        print('\n Try:', list(self.conexion.mode_mapping().keys()))
        sys.exit(1)

    print('\n cambiando a modo ' + modo + '...', end = " ")
    # Get mode ID
    mode_id = self.conexion.mode_mapping()[modo]

    self.conexion.mav.set_mode_send(
        self.conexion.target_system,
        mavutil.mavlink.MAV_MODE_FLAG_CUSTOM_MODE_ENABLED,
        mode_id)

    while True:
        msg = self.conexion.recv_match(type = 'HEARTBEAT', blocking = False)
        if msg:
            new_mode = mavutil.mode_string_v10(msg)
            if new_mode == modo:
                print('OK')
```

```
            break

    def set_rc_channel_list(self, channels_pwm):
        """
        Establece el canal de entrada RC al valor pwm sin modificar el resto de canales

        Parameters
        ----------
        channel_id : int
            numero de canal     1 <= channel_id <= 18.
                0 = PITCH
                1 = ROLL
                2 = THROTTLE
                3 = YAW
                4 = FORDWARD
                5 = LATERAL
                6 = CAMERA_PAN
                7 = CAMERA_TILT
                8 = LIGHTS_1
                9 = LIGHTS_2
                10 = VIDEOSWITHC
                11..17 = canales auxiliares
        pwm : int, optional
            valor en us del pwm 1100 <= pwm <= 1900. The default is 1500.

        Returns
        -------
        None.
```

```
        """
        self.conexion.mav.rc_channels_override_send(
            self.conexion.target_system,                # target_system
            self.conexion.target_component,             # target_component
            *channels_pwm)                  # RC channel list, in microseconds.
        print('pwm:    PITCH    ROLL    THROTTLE    YAW    FORDWARD    LATERAL')
        print('-----------------------------------------------------------------')
        print(f'     {channels_pwm[0]:10} {channels_pwm[1]:10} {channels_pwm[2]:10}
{channels_pwm[3]:10} {channels_pwm[4]:10} {channels_pwm[5]:10}')




    def set_rc_channel_pwm(self, channel_id, pwm=1500):
        """
        Establece el canal de entrada RC al valor pwm sin modificar el resto de canales


        Parameters
        ----------
        channel_id : int
            numero de canal     1 <= channel_id <= 18.
                0 = PITCH
                1 = ROLL
                2 = THROTTLE
                3 = YAW
                4 = FORDWARD
                5 = LATERAL
```

```
        6 = CAMERA_PAN
        7 = CAMERA_TILT
        8 = LIGHTS_1
        9 = LIGHTS_2
        10 = VIDEOSWITHC
        11..17 = canales auxiliares
    pwm : int, optional
        valor en us del pwm 1100 <= pwm <= 1900. The default is 1500.

    Returns
    -------
    None.

    """
    if channel_id < 0 or channel_id > 17:
        print("\n ¡ Error: no existe el canal !")
        return

    channel_list = ('PITCH', 'ROLL', 'THROTTLE', 'YAW', 'FORDWARD', 'LATERAL',
                'CAMERA_PAN', 'CAMERA_TILT', 'LIGHTS_1', 'LIGHTS_2', 'VIDEO_SWITCH')
    print('\n poniendo', channel_list[channel_id], 'a', pwm, '...', end = " ")

    rc_channel_values = [65535]*18
    rc_channel_values[channel_id] = pwm
    self.conexion.mav.rc_channels_override_send(
        self.conexion.target_system,                # target_system
        self.conexion.target_component,             # target_component
        *rc_channel_values)                 # RC channel list, in microseconds.
```

```
        print("OK")

    def set_rc_stop(self):
        """
        Pone todos los pwm de los propulsores a 1500 us sin modificar el resto de servos

        Returns
        -------
        None.

        """
        print("\n parando propulsores ...", end = " ")
        rc_channel_values = [65535]*18
        rc_channel_values[:6] = [1500]*6
        self.conexion.mav.rc_channels_override_send(
            self.conexion.target_system,                # target_system
            self.conexion.target_component,             # target_component
            *rc_channel_values)                 # RC channel list, in microseconds.
        print("OK")

    def get_info(self, tipo = None, bloqueo = True):
        """
        Solicita un mensaje de informacion al BlureRov2

        Parameters
        ----------
        tipo : string, optional
```

tipo de la informacion deseada. Puede ser:

None (retorna toda la informacion)

'ATTITUDE'

'GLOBAL_POSITION_INT'

'SYS_STATUS'

'POWER_STATUS'

'MEMINFO'

'NAV_CONTROLLER_OUTPUT'

'MISSION_CURRENT'

'VFR_HUD'

'SERVO_OUTPUT_RAW'

'RC_CHANNELS'

'RAW_IMU'

'SCALED_IMU2'

'SCALED_IMU3'

'SCALED_PRESSURE'

'SCALED_PRESSURE2'

'SCALED_PRESSURE3'

'GPS_RAW_INT'

'SYSTEM_TIME'

'AHRS'

'SIMSTATE'

'AHRS2'

'EKF_STATUS_REPORT'

'LOCAL_POSITION_NED'

'VIBRATION'

'BATTERY_STATUS'

'NAMED_VALUE_FLOAT' (8 posibilidades)

```
        Returns
        -------
        string
            mensaje con la informacion

        """
        msg = self.conexion.recv_match(type = tipo, blocking = bloqueo)
        return msg

    def request_message_interval(self, message_id, time_seconds: float):
        """
        Request MAVLink message in a desired frequency,
        documentation for SET_MESSAGE_INTERVAL:

https://mavlink.io/en/messages/common.html#MAV_CMD_SET_MESSAGE_INTERVAL

        Args:
            message_id (str): MAVLink message ID
            time_seconds (float): Desired time inverval in seconds
        """
        if message_id == 'AHRS2':
            msg =  mavutil.mavlink.MAVLINK_MSG_ID_AHRS2
        elif message_id == 'SYSTEM_TIME':
            msg =  mavutil.mavlink.MAVLINK_MSG_ID_SYSTEM_TIME
        elif message_id == 'LOCAL_POSITION_NED':
            msg =  mavutil.mavlink.MAVLINK_MSG_ID_LOCAL_POSITION_NED
```

```
        elif message_id == 'ATTITUDE':
            msg =  mavutil.mavlink.MAVLINK_MSG_ID_ATTITUDE
        elif message_id == 'VFR_HUD':
            msg =  mavutil.mavlink.MAVLINK_MSG_ID_VFR_HUD

        self.conexion.mav.command_long_send(
            self.conexion.target_system, self.conexion.target_component,
            mavutil.mavlink.MAV_CMD_SET_MESSAGE_INTERVAL, 0,
            msg, # The MAVLink message ID
            1e6 * time_seconds, # The interval between two messages in microseconds. Set to -1 to disable and 0 to request default rate.
            0, 0, 0, 0, # Unused parameters
            0, # Target address of message stream (if message has target address fields). 0: Flight-stack default (recommended), 1: address of requestor, 2: broadcast.
        )

    def close_connection(self):
        """
        Cierra la conexion con el BlureRov2

        Returns
        -------
        None.

        """
        print('\n cerrando conexion ...', end = " ")
        self.conexion.close()
        print('OK\n')
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#Mario Charlon Soldevilla
import numpy as np
from matplotlib import pylab as pl
from pykml import parser

class PID_control():
    def __init__(self, kp, ki, kd, current_time):
        self.time = current_time
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.out = 0.0
        self.error = 0.0
        self.deriv_error = 0.0
        self.int_error = 0.0

    def update(self, error, current_time):
        sample_time = current_time - self.time
        self.time = current_time
        self.deriv_error = (error-self.error)/sample_time
        if np.abs(self.out) < 1.0:
            self.int_error += (error+self.error)*0.5*sample_time
        self.out = self.kp*error + self.ki*self.int_error + self.kd*self.deriv_error
```

```
        if self.out > 1.0:
            self.out = 1.0
        elif self.out < -1.0:
            self.out = -1.0
        self.error = error

    def reset(self, current_time):
        self.time = current_time
        self.out = 0.0
        self.error = 0.0
        self.deriv_error = 0.0
        self.int_error = 0.0

# Opcional

def plotdata(datafile):
    # pl.ion()
    pl.close("all")
    pl.pause(0.1)
    #time, obj_yaw, rov_yaw, obj_x, rov_x, obj_y, rov_y, obj_z, rov_z
    data = np.loadtxt(datafile)
    # data = data[1:, :]
    data[:,0] -= data[0,0]
    data[:,3:5] -= data[0, 4]
    data[:,3:5] *= -1
    data[:,5:7] -= data[0, 6]
    # data[:,7:] -= data[0, -1]
```

```
fig, ax = pl.subplots(nrows=2, ncols=2)

ax[0,0].plot(data[:,0], data[:,1]*180/np.pi, label='target')
ax[0,0].plot(data[:,0], data[:,2]*180/np.pi, label='actual')
ax[0,0].set_xlabel('t(s)')
ax[0,0].set_ylabel('YAW(º)')
ax[0,0].legend(loc='upper right')
ax[0,0].grid(visible=True)

ax[0,1].plot(data[:,0], data[:,7], label='target')
ax[0,1].plot(data[:,0], data[:,8], label='actual')
ax[0,1].set_xlabel('t(s)')
ax[0,1].set_ylabel('Z(m)')
ax[0,1].legend(loc='upper right')
ax[0,1].grid(visible=True)

ax[1,0].plot(data[:,0], data[:,3], label='target')
ax[1,0].plot(data[:,0], data[:,4], label='actual')
ax[1,0].set_xlabel('t(s)')
ax[1,0].set_ylabel('X(m)')
ax[1,0].legend(loc='upper right')
ax[1,0].grid(visible=True)

ax[1,1].plot(data[:,0], data[:,5], label='target')
ax[1,1].plot(data[:,0], data[:,6], label='actual')
ax[1,1].set_xlabel('t(s)')
ax[1,1].set_ylabel('Y(m)')
```

```
    ax[1,1].legend(loc='upper right')
    ax[1,1].grid(visible=True)

    pl.tight_layout()
    pl.show()

def kml2list(kml_file):
    root = parser.parse(kml_file).getroot()
    line = root.Document.Placemark.LineString.coordinates
    points = line.text.split()
    coords = [p.split(',') for p in points]
    waypoints = [[float(n) for n in c] for c in coords]
    # lat, lon, alt
    return [[wp[1], wp[0], wp[2]] for wp in waypoints]

def kml2list2(kml_file):
    root = parser.parse(kml_file).getroot()
    points = root.findall('.//{http://www.opengis.net/kml/2.2}Point')
    return [[float(coord) for coord in p.coordinates.text.split(',')] for p in points]

def geodetic_to_cartesian(latitude, longitude, altitude):
    # Radio de la Tierra en metros
    R = 6371000.0

    # Convertir las coordenadas de grados a radianes
    lat_rad = np.deg2rad(latitude)
    lon_rad = np.deg2rad(longitude)
```

```
    # Calcular las coordenadas cartesianas
    x = (R + altitude) * np.cos(lat_rad) * np.cos(lon_rad)
    y = (R + altitude) * np.cos(lat_rad) * np.sin(lon_rad)
    # z = (R + altitude) * np.sin(lat_rad)
    z = altitude

    return np.array([x, y, z])

def geodetic_to_geocentric(latitude, longitude, height):
    """Return geocentric (Cartesian) Coordinates x, y, z corresponding to
    the geodetic coordinates given by latitude and longitude (in
    degrees) and height above ellipsoid. The ellipsoid must be
    specified by a pair (semi-major axis, reciprocal flattening).

    """
# Ellipsoid parameters: semi major axis in metres, reciprocal flattening.
    # GRS80 = 6378137, 298.257222100882711
    WGS84 = 6378137, 298.257223563
    ellipsoid = WGS84
    phi = np.deg2rad(latitude)
    lamb = np.deg2rad(longitude)
    sin_phi = np.sin(phi)
    a, rf = ellipsoid          # semi-major axis, reciprocal flattening
    e2 = 1 - (1 - 1 / rf) ** 2  # eccentricity squared
    n = a / np.sqrt(1 - e2 * sin_phi ** 2) # prime vertical radius
    r = (n + height) * np.cos(phi)   # perpendicular distance from z axis
    x = r * np.cos(lamb)
```

```
    y = r * np.sin(lamb)
    # z = (n * (1 - e2) + height) * sin_phi
    z = height
    return np.array([x, y, z])


def vector_between_points(lat1, lon1, alt1, lat2, lon2, alt2):
    # Obtener las coordenadas cartesianas de los dos puntos
    x1, y1, z1 = geodetic_to_cartesian(lat1, lon1, alt1)
    x2, y2, z2 = geodetic_to_cartesian(lat2, lon2, alt2)

    # Calcular el vector entre los dos puntos
    vector_x = x2 - x1
    vector_y = y2 - y1
    vector_z = z2 - z1

    return vector_x, vector_y, vector_z

if __name__ == "__main__":
    plotdata('data.csv')
```

2 CODIGO DE MATLAB

```
%Codigo de carga y analisis de los datos
%Mario Charlon Soldevilla
format long

data=load('data_ahrs2.csv');
waypoints=load('waypoints.csv');
```

```
actual_time = data(:,1);

%Waypoints
wp_lat = waypoints(:, 1);
wp_lon = waypoints(:, 2);
wp_alt = waypoints(:, 3);

%coordenadas geocentricas
wgs84 = wgs84Ellipsoid('meter');
[wp_x, wp_y] = geodetic2ecef(wgs84,wp_lat, wp_lon,wp_alt);

%hpr (rad)
roll = data(:,2);
pitch = data(:,3);
yaw = data(:,4);
yaw_error = data(:,8);
%hpr (deg)
roll_d = rad2deg(roll);
pitch_d = rad2deg(pitch);
yaw_d = rad2deg(yaw);
yaw_error_d = rad2deg(yaw_error);

%coordenadas geodesicas
altitude = data(:,5);
lat = data(:,6);
lon = data(:,7);

%coordenadas geocentricas del ROV
rov_x = data(:,10);
rov_y = data(:,12);
rov_z = data(:,14);

%coordenadas geocentricas ruta
obj_x = data(:,9);
obj_y = data(:,11);
obj_z = data(:,13);


%plot de los datos hpr

%ROLL
ts=1/60;
t = ((0:length(actual_time)-1) * ts)';
subplot(3,2,1)
plot(t,roll_d)
xlabel('Tiempo (s)');
ylabel('roll (grados º)');
title('Gráfico del roll en función del tiempo');

%PITCH
subplot(3,2,2)
plot(t,pitch_d)
xlabel('Tiempo (s)');
ylabel('pitch (grados º)');
title('Gráfico del pitch en función del tiempo');

%YAW
```

```
subplot(3,2,3)
plot(t,yaw_d)
xlabel('Tiempo (s)');
ylabel('yaw (grados º)');
title('Gráfico del yaw en función del tiempo');

%ALTITUD

subplot(3,2,4)
plot(t,altitude)
xlabel('Tiempo (s)');
ylabel('Altitud (metros)');
title('Gráfico de la evolucion de la altitud en función del tiempo');


%ERROR DEL YAW
subplot(3,2,5)
plot(t,yaw_error_d)
xlabel('Tiempo (s)');
ylabel('Error del yaw(º)');
title('Gráfico del error del yaw en función del tiempo');



%Plot de latitud y longitud
figure();
geolimits([40.60, 47],[-3.9, -3.7])
geoplot(lat,lon,"--gs","LineWidth",2, ...
    "MarkerSize",5,"MarkerEdgeColor","b", ...
    "MarkerFaceColor",[0.5 0.5 0.5])

geobasemap satellite %conexion a internet requerida


% Distancias entre waypoints
% Crear una nueva figura
figure;
% Trazar las coordenadas en función del tiempo
plot3(rov_x, rov_y, rov_z, 'LineWidth', 2);
hold on;
plot3(obj_x, obj_y, obj_z,'r', 'LineWidth', 2);
hold on;
plot3(wp_x, wp_y, wp_alt, 'gs', 'MarkerSize', 10);
% Etiquetar waypoints
for i = 1:length(waypoints)
    text(wp_x(i), wp_y(i), wp_alt(i), sprintf('WP%d', i), 'Color',
'black','VerticalAlignment', ...
        'top','HorizontalAlignment', 'center','Fontsize', 12);

end
hold off;
title('Evolución de las coordenadas en función del tiempo');
xlabel('X');
ylabel('Y');
zlabel('Z');
legend('Ruta seguida por el vehiculo','Ruta objetivo');
grid on;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ANALISIS POSTERIOR
% desviación entre la ruta objetivo y la ruta seguida por el ROV
% num_points = length(obj_x);
% deviation = zeros(num_points, 1);
%
% for i = 1:num_points
%     deviation(i) = sqrt((rov_x(i) - obj_x(i))^2 + (rov_y(i) - obj_y(i))^2 +
(rov_z(i) - obj_z(i))^2);
% end
%
% % Graficar la desviación a lo largo del tiempo
% figure;
% plot(t, deviation, 'LineWidth', 2);
% xlabel('Tiempo (s)');
% ylabel('Desviación (metros)');
% title('Desviación entre la ruta seguida y la ruta objetivo');
% grid on;
```

3 ARCHIVOS Y MODELOS 3D UTILIZADOS

En este último apartado de los anexos se incluye un enlace en el que se encuentran diferentes archivos secundarios y modelos 3D que se han utilizado en el proyecto:

https://unicancloud-my.sharepoint.com/:u:/g/personal/mcs991_alumnos_unican_es/Eeh-RuyTKdhLq3vJhkM-0R8BRvuJUPFUQXvjkKWIU01veg?e=REBbX1