

MÁSTER EN INGENIERÍA INFORMÁTICA
Facultad de informática
Universidad Complutense de Madrid

Autopiloto aeronáutico LibelulaX

LibelulaX aeronautical autopilot

[FreeRTOS HITL]

Septiembre de 2022



UNIVERSIDAD
COMPLUTENSE
MADRID

Autor:
Antonio Vázquez Pérez
Directores:
Guillermo Botella Juan
Alberto Antonio del Barrio García

CALIFICACIÓN FINAL: 9

Índice

1. Resumen	6
2. Abstract	6
3. Introducción	7
3.1. Historia	7
3.2. Aeromodelismo	8
3.3. Objetivo	9
4. Estado del arte	9
4.1. Soluciones comerciales	9
4.1.1. Radiolink PixHawk	10
4.1.2. PixHawk 4	11
4.1.3. Akbird 2.0	12
4.2. ArduPilot	13
4.3. Conclusiones	13
4.3.1. Tabla resumen	13
5. Especificaciones	15
5.1. Funcionalidad	15
5.2. Hardware	15
5.3. Software	19
5.3.1. Visual studio code	19
5.3.2. ESP-IDF Software development kit	19
5.3.3. FlightGear	19
5.3.4. Programación	19
5.3.5. Suite LibreOffice	20
5.4. Planificación	20
5.4.1. Desglose de tareas	22
6. Entorno de desarrollo	23
6.1. Repositorio	23
7. Desarrollo	23
7.1. ESP-IDF - Proyecto básico	23
7.2. Conexión HITL	25
7.3. Control de la aeronave	25
7.3.1. Superficies de control	25
7.4. Creación del modelo PID	26
7.4.1. Introducción	26
7.4.2. Implementación	28
7.4.3. Pruebas de control	28
7.5. Creación de una consola por puerto serie	31

7.6.	Elección y control del simulador	32
7.6.1.	Valoración de alternativas	33
7.7.	Protocolo de interconexión	34
7.7.1.	VARIABLES DE ENTRADA	34
7.7.2.	VARIABLES DE SALIDA	34
7.7.3.	LibelulaXProtocol	35
7.7.4.	Inconvenientes del protocolo ⇒ script de comunicación	38
7.8.	Brújula y guiado GPS	38
7.8.1.	Posicionamiento GPS	39
7.8.2.	Distancia entre dos puntos	40
7.8.3.	Rumbo - brújula entre dos puntos	41
7.8.4.	Elaboración y visualización de rutas	43
7.9.	Elementos PID y de control	46
7.9.1.	Cabeceo - PID	46
7.9.2.	Alabeo - PID	47
7.9.3.	Guinada	48
7.9.4.	Velocidad	48
7.9.5.	Altitud - PID secundario	48
7.9.6.	Rumbo de brújula - PID	50
7.10.	Modos de vuelo	50
7.10.1.	Control manual	50
7.10.2.	Control semi-automático	51
7.10.3.	Control automático	53
7.10.4.	Máquina de estados	53
7.11.	Sistema operativo	55
7.11.1.	Características de FreeRTOS	55
7.11.2.	Programación	56
7.11.3.	Clase aircraft	56
7.11.4.	Clase PID	57
7.11.5.	Clase GPS	58
7.11.6.	Clase autopilot	60
7.11.7.	Componente cmdConsole	62
7.11.8.	Diagrama de tareas	63
8.	Banco de pruebas	63
8.1.	Objetivo	63
8.2.	Prueba 1: Precisión de ruta	63
8.2.1.	Escenario	64
8.2.2.	Ejecución y resultados	64
8.2.3.	Análisis de resultados	65
8.3.	Prueba 2: Ruido giroscópico	65
8.3.1.	Escenario	65
8.3.2.	Ejecución y resultados	65
8.3.3.	Análisis de resultados	66

9. Compilación y ejecución de la demostración	66
9.0.1. Compilación	66
9.0.2. Ejecución	67
10. Conclusiones	67
10.1. Cuestiones	68
10.1.1. ¿Se ha cumplido el objetivo de este proyecto?	68
10.1.2. ¿Cómo de resistente y fiable es el modo automático?	68
10.1.3. ¿Qué planes futuros esperamos de este proyecto?	68
10.2. Questions	69
10.2.1. Have the objective of this project been achieved?	69
10.2.2. How resilient and reliable is the automatic mode?	69
10.2.3. Which future plans we expect from this project?	69
10.3. Ampliaciones y consideraciones	70

Glosario de términos y abreviaturas

- **HITL:** Hardware In The Loop.
- **UAV:** Unmanned Aerial Vehicle.
- **PID:** Proportional Integral Derivative.
- **GPS:** Global Positioning System.
- **MPU:** MicroProcessor Unit.
- **IDE:** Integrated Development Environment.
- **SDK:** Software Development Kit.

Índice de figuras

1.	<i>El Kettering Bug siendo preparado para el despegue. (Prisacariu y col., 2017)</i>	7
2.	<i>Tecnologías actuales (Prisacariu y col., 2017)</i>	8
3.	<i>Avión entrenador comercial FMS ranger.</i>	9
4.	<i>Pixhawk RadioLink</i>	10
5.	<i>Pixhawk 4</i>	11
6.	<i>Akbird 2.0</i>	12
7.	<i>Microcontrolador ESP32 mini</i>	16
8.	<i>Módulo MPU</i>	16
9.	<i>Módulo GPS</i>	17
10.	<i>Emisora y receptor</i>	17
12.	<i>Página principal del proyecto en la plataforma github.</i>	23
13.	<i>Interfaz de configuración del proyecto(menuconfig)</i>	24
14.	<i>Página de documentación y ayuda de ESP-IDF.</i>	24
15.	<i>Flujo de hardware in the loop.</i>	25
16.	<i>Ejes de movimiento de una aeronave. («Blog aerodeporte», 2022)</i>	26
17.	<i>Esquema de flujo de un PID</i>	27
18.	<i>Esquema del cabeceo de la aeronave.</i>	28
19.	<i>Caso de prueba 0 PID</i>	29
20.	<i>Caso de prueba 1 PID</i>	30
21.	<i>Caso de prueba 2 PID</i>	30
22.	<i>Salida del comando help en el terminal serie. Tenemos comandos de utilidad para poder ver la memoria restante, la versión del sdk o reiniciar el dispositivo.</i>	32
23.	<i>Especificación del protocolo en sentido hacia el microcontrolador.</i>	36
24.	<i>Especificación del protocolo en sentido desde el microcontrolador.</i>	37
25.	<i>Esquema de funcionamiento de script serialParser.py</i>	38
26.	<i>Imagen google earth, satélites GPS.</i>	39
27.	<i>Distancia entre A y B.</i>	40
28.	<i>Código de cálculo de distancias</i>	41

29.	<i>Por ejemplo, el camino seguido para ir desde Buenos Aires a Pekín por la distancia más corta.</i>	42
30.	<i>Bagdad - Osaka</i>	42
31.	<i>Código de cálculo de rumbo</i>	43
32.	<i>Software mission planner. («Mission Planner software», 2022)</i>	44
33.	<i>Página de elaboración de rutas Geoplaner.</i>	45
34.	<i>Interfaz del simulador FlightGear con historial de una ruta realizada por este mismo proyecto.</i>	46
35.	<i>Figura que detalla el cabeceo del avión («Blog aerodeporte», 2022)</i>	46
36.	<i>Figura que detalla el alabeo del avión («Blog aerodeporte», 2022)</i>	47
37.	<i>Figura que detalla la guiñada del avión («Blog aerodeporte», 2022)</i>	48
38.	<i>Relación entre presión atmosférica y altitud (Soares, 2015)</i>	49
39.	<i>Esquema de conexión del modo de vuelo manual.</i>	51
40.	<i>Esquema de conexión del modo de vuelo semi-automático.</i>	52
41.	<i>Esquema de ángulo inseguro y recuperación.</i>	52
42.	<i>Esquema de conexión del modo de vuelo automático.</i>	53
43.	<i>Entorno de programación ESP-IDF («ESP-IDF documentation», 2022)</i>	56
44.	<i>Trazada real de la aeronave sobre un punto de ruta.</i>	64
45.	<i>Trazada real de la aeronave sobre la ruta completa.</i>	66

1. Resumen

El presente trabajo de fin de máster tiene el objetivo de crear una solución de piloto automático por microcontrolador efectiva y de bajo costo para el **control completamente automático de aeronaves de tipo UAV** o vehículo aéreo no tripulado.

Dentro del mercado actual se pueden encontrar soluciones que utilizan un procesador de tipo SoC, complejos y de un costo elevado. Por ello, este proyecto busca **implementar una solución propia** dentro de un microcontrolador de tipo **ESP32** con doble núcleo y capacidad de conectividad bluetooth y wifi, mucho más barato pero sin perder posibilidades, con la fiabilidad de un sistema operativo como **FreeRTOS**.

El sistema final poseerá una interfaz serie la cual proveerá un **terminal de comandos** tipo linux por el que poder enviar multitud de comandos y permitirá la comunicación para una simulación en bucle mediante el **programa simulador FlightGear**, de código abierto, utilizando el hardware real ESP32 vía USB.

Finalmente, obtendremos un avión de tipo radiocontrol a pequeña escala pudiendo recorrer una ruta gps y volver de nuevo al punto al que fué lanzado a la espera de retomar el control manual para su aterrizaje.

Todo ello siendo **computado** en tiempo real por el **hardware** sobre un **entorno simulado**.

2. Abstract

This end of master project aims to create a microcontroller-based autopilot as an effective and low-cost solution for the **fully automatic control of UAV-type aircraft** or unmanned aerial vehicles.

Within the current market, solutions can be found that use a complex and expensive SoC-type processor. Thus, this project seeks to **implement its own solution** using a **ESP32**-type microcontroller with dual core, Bluetooth and Wi-Fi connectivity, much cheaper but without losing possibilities, with the reliability of an operating system like as **FreeRTOS**. The final system will have an interface providing a linux styled **command terminal** for communicating multiple commands and allowing communication for a looped simulation using the open source **FlightGear simulator program**, communicating with real hardware via USB.

Finally, we will obtain a small-scale radio-controlled aircraft, being able to follow a GPS route and return again to the point to which it was launched, waiting to resume manual control for landing.

Everything being **computed** at real-time by **hardware** and running on a **simulated environment**.

3. Introducción

3.1. Historia

La idea de poder utilizar aviones los cuales no requieran de una persona para poder tripularlos lleva en el pensamiento de los investigadores desde la **primera guerra mundial**. La primeras incursiones en este campo surgieron dentro del campo militar con la esperanza de poder **amortiguar los costes** de la pérdida de pilotos y aeronaves las cuales fueron producidas en masa durante la primera guerra mundial.

El primer prototipo de una máquina de este tipo estuvo vinculado al estadounidense **Elmer Sperry**, quien creó una aeronave controlada por piloto automático (*Prisacariu y col., 2017*). Los profesionales militares vieron un gran potencial en el UAV y cedieron siete aviones de tipo Curtis N-9 para montar con este sistema de **piloto automático** y utilizarlos como aviones bombarderos. Los primeros vuelos de prueba fueron realizados en 1917 con un piloto en la cabina. Este piloto fue el responsable de los despegues y aterrizajes, sin embargo, las demás fases del vuelo fueron guiadas por el piloto automático. Después de volar 48 km las bombas fueron expulsadas pero no pudo acertar al objetivo a menos de 3 km.

Otro bombardero, el **Kettering Bug** [Figura 1] se completó en noviembre de 1917. Fue pedido por las Fuerzas Armadas de los EE. UU. y construido por Charles Kettering. El fuselaje fue creado por Orville Wright y el sistema de control y navegación fue desarrollado por el experto en el tema, Elmer Sperry, quien creó el avión automático que se mencionó anteriormente. Después del despegue, el pequeño biplano fue **guiado por el automatismo** en dirección al objetivo, donde cuando pasó el tiempo preestablecido, el motor se detuvo, las alas cayeron y el cuerpo en forma de torpedo lleno de **explosivo de 80 kg** dio en el blanco y explotó. El motor Ford de 4 pistones y 40 caballos podría acelerar la aeronave a 100 km/h. Aunque la construcción fue exitosa, no participó en la guerra, porque cuando el ejército de los EE. UU. lo puso en servicio, **la guerra había terminado**.

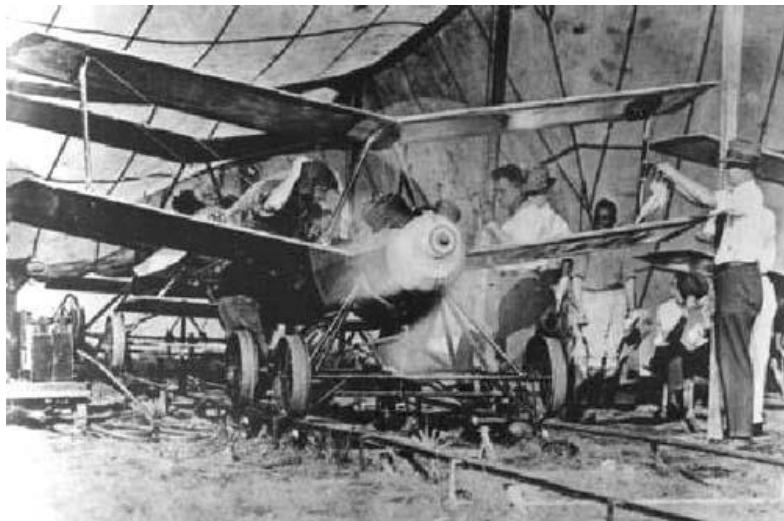
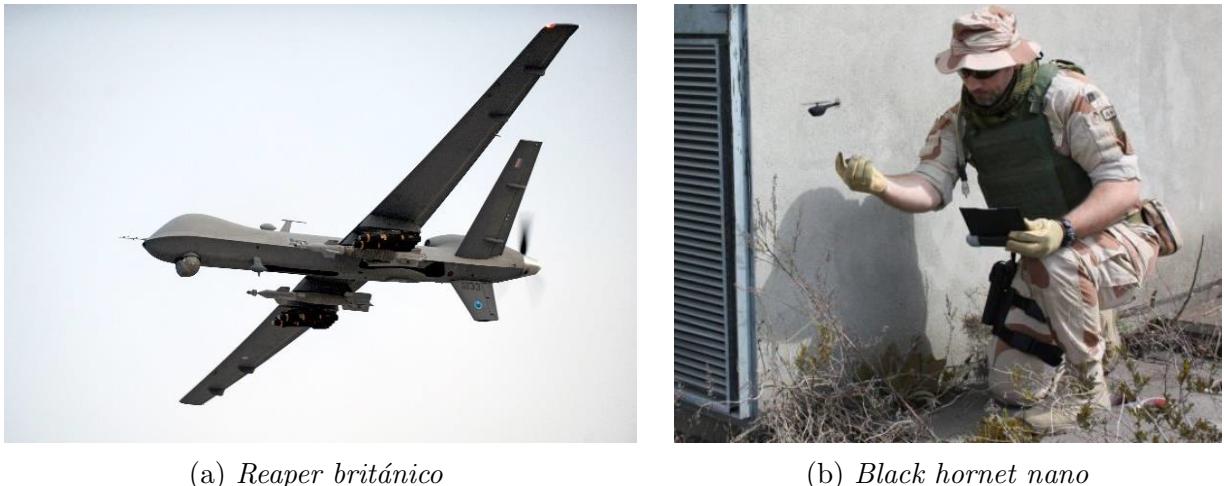


Figura 1: *El Kettering Bug siendo preparado para el despegue. (Prisacariu y col., 2017)*

Tras esta introducción histórica, en su futuro, el devenir de este tipo de dispositivos se seguiría viendo fuertemente influenciado por la industria militar. Los avances en este campo habrán pasado desde la asistencia al vuelo, misiles guiados, aeronaves con control por radio hasta lo que tenemos hoy en día como aviones UAV completamente automáticos para misiones de reconocimiento o micro uav.



(a) *Reaper británico*

(b) *Black hornet nano*

Figura 2: *Tecnologías actuales (Prisacariu y col., 2017)*

3.2. Aeromodelismo

Para el caso particular de este proyecto nos centraremos en el **mundo del aeromodelismo**, donde miles de aficionados construyen sus maquetas a **pequeña escala** y, efectivamente, no cabe una persona dentro de la cabina del avión a diferencia de sus homólogos a copiar. Para solucionar este problema, los aficionados utilizan el control por radio para así, desde tierra, poder manejar en el aire el aeromodelo.

Los aviones a escala también se llevan utilizando mucho tiempo en la industria aeronáutica para probar sus propiedades aerodinámicas, como por ejemplo probándolos dentro de un túnel de viento antes de llevarlos a producción.

En nuestro caso vamos a realizar el piloto automático para un modelo, como es llamado dentro del aeromodelismo, de tipo **entrenador**. Tiene la morfología de lo que podríamos conocer como una avioneta.



Figura 3: Avión entrenador comercial FMS ranger.

3.3. Objetivo

Como hemos podido ver, el control a **distanzia o automático** de modelos aeronáuticos es algo que ha despertado en la historia mucho interés. Este proyecto servirá para poder desarrollar una electrónica de control de dirección, estabilización y guiado gps totalmente automatizada para ser integrada en aviones radiocontrol.

El **reto** de este proyecto es encontrar la forma de lograr dicho control mientras se intenta **abaratizar y simplificar** todo lo posible para lograr un funcionamiento **efectivo** y a **tiempo real** en un microcontrolador de bajo coste. Además de demostrar las capacidades adquiridas durante el máster.

4. Estado del arte

Hemos de tener en cuenta que el concepto de piloto automático ya existe y que no solamente se encuentra en aviones, sino también en otros formatos como cuadricópteros. Para centrarnos un poco más en el contexto de este trabajo solamente evaluaremos los pilotos automáticos que aplican al nivel manejo de avión radiocontrol.

4.1. Soluciones comerciales

Para poder evaluar las necesidades reales de este trabajo y el enfoque de obtener un producto de menor coste e iguales o mejores prestaciones observaremos los productos comerciales ya existentes en el mercado.

4.1.1. Radiolink PixHawk



Figura 4: *Pixhawk RadioLink*

Este controlador de vuelo es uno de los más utilizados dentro de la afición del aeromodelismo e incorpora un microcontrolador ARM Cortex M4 de 32bits, acelerómetro/giróscopo de 3 ejes, magnetómetro y barómetro.

Según la página web de robotshop:

Basado en el 3DR PIXhawk, el Piloto Automático Avanzado Radiolink PixHawk se desarrolló con un diseño de circuito optimizado y un sistema de control de calidad de software de automatización de diseño creativo. Tiene menos interferencia de componentes internos, menos ruido, una brújula más precisa y seguridad durante el vuelo, tiene un diseño humanizado.

(«Robotshop Pixhawk distributor website», 2022)

Precio al consumidor (+21 %IVA): **144,52€**
+ GPS: **179,46€**

4.1.2. PixHawk 4



Figura 5: *Pixhawk 4*

Este modelo es más avanzado que el mostrado anteriormente e incorpora el GPS. Posee además mayor capacidad de conexión con buses externos y posibilidad de realizar vuelo asistido.

(«Mouser Pixhawk4 distributor website», 2022)

Precio al consumidor (+21 %IVA): **242,52€**

4.1.3. Akbird 2.0



Figura 6: *Akbird 2.0*

Realmente este producto se encuentra actualmente descontinuado pero lo mostraremos como ejemplo de las prestaciones/precio de los competidores.

Especificaciones:

- Sensores GPS, barómetro y altitud para estabilizar y controlar el modelo.
- Integrada funcionalidad HD OSD.
- GPS con función "volver a casa".
- Vuelo waypoint GPS.
- Vuelo de crucero, planeo recto y nivelado.
- Auto-nivelación.
- Registro de datos de vuelo.

(«*Hobbyking Akbird2.0 distributor website*», 2022)

Precio al consumidor (+21 %IVA): **190,05\$**

4.2. ArduPilot



Esta plataforma consiste en un firmware abierto que, realizando unos ajustes y configuración previos, sería capaz de llevar un control avanzado de rutas GPS, vuelta a casa y gestión 100 % automática de multitud de dispositivos hardware.

Este es un sistema de piloto automático confiable, versátil y de código abierto que admite muchos tipos de vehículos. El código fuente está desarrollado por una gran comunidad.

Su desventaja es que debido a las características del proyecto no se encuentra optimizado para microcontroladores de características modestas, encareciendo el precio del hardware para proyectos en cantidad o no aprovechando todas las prestaciones del hardware.

Como ventaja tenemos que a este software le acompaña otras herramientas como *mission planner* que permite planificar todas las rutas de forma sencilla.

4.3. Conclusiones

Con estos ejemplos habrá sido posible observar que tanto las soluciones comerciales como las de bibliotecas abiertas pueden requerir bien de un alto desembolso o bien de un esfuerzo de integración junto a un hardware no tan barato. Por las razones mencionadas se procederá a elaborar un software compatible con los microcontroladores ESP32.

Se trata de un microcontrolador ampliamente extendido sobretodo en el mundo del IoT, bien documentado y es de los más conocidos por los entusiastas de los microcontroladores o automatismos.

4.3.1. Tabla resumen

A continuación tenemos una tabla de resumen en la que poder observar las características de todos los productos en comparación a este proyecto.

ALTERNATIVAS			
Nombre	Desempeño	Facilidad	Precio
RadioLink PixHawk	Alto	Alta	Alto
PixHawk 4	Muy alto	Alta	Muy alto
AkBird 2.0	Alto	Alta	Alto
Ardupilot	Medio - alto	Baja	Medio
LibelulaX	Medio	Media	Muy bajo

Nuestra solución encajaría dentro de unas **características generales medias** pero con **muy bajo precio**.

5. Especificaciones

5.1. Funcionalidad

A continuación habremos de saber a grandes rasgos qué especificaciones queremos que cumpla este sistema. Por lo tanto, se van a definir una serie de requisitos los cuales se procederán a detallar más en profundidad posteriormente:

- **Control automático**

Capacidad de seguir rutas GPS: el sistema ha de ser capaz de comprender el sistema de posicionamiento global, y ha de poder seguir una ruta previamente especificada de forma precisa.

Función de retorno a casa: cuando la aeronave termine la ruta u ocurra un problema de gravedad ha de tener la capacidad de intentar volver al punto global desde el que despegó.

Resiliencia a ráfagas de viento: el modelo de control ha de poder soportar ruido de sensores o actuaciones externas como ráfagas de viento.

Posibilidad de probar el sistema en un entorno simulado: es de vital importancia para este proyecto que el sistema pueda ser capaz de conectarse a un ordenador y controlar un avión radiocontrol virtual. Esto como forma de prueba o para visualizar la ruta que posteriormente seguirá el avión.

5.2. Hardware

A fin de poder demostrar la **asequibilidad del proyecto** se ha montado una configuración basada en componentes reales los cuales conformarían **hipotéticamente** la solución completa.

Comentaremos a continuación los elementos que hemos seleccionado dentro del mercado actual y las razones de estas decisiones.

Los componentes cuyos nombres se encuentran resaltados en azul serán los que estemos utilizando para desarrollar el piloto automático de este proyecto. La electrónica seleccionada para la configuración es la siguiente:

- **Miccontrolador ESP32:** Dual core a un precio muy asequible. Costo estimado: 6,5€.



Figura 7: *Microcontrolador ESP32 mini*

En contraposición a los procesadores de las alternativas mencionadas en el apartado de *estado del arte*, este, en relación precio-prestaciones, es mucho mejor, ya que contamos con dos núcleos de procesamiento suficientemente potentes y además tiene integrado FreeRTOS en su entorno de programación proveído por el fabricante.

La facilidad de configuración y accesibilidad en precio al público general lo haría un gran candidato para, en caso de publicar el código, hacer este autopiloto altamente asequible para aficionados (repositorio en apartado 6.1).

Cualquier procesador de esta gama ESP32 nos podría ser válido para este proyecto.

- **Sensor múltiple MPU-9250:** Incluye: giroscopio de tres ejes + acelerómetro triaxial + campo magnético triaxial + barómetro. Costo estimado: 10€.



Figura 8: *Módulo MPU*

Este sensor tan completo es capaz de ofrecer mucho por muy poco, y además sin sacrificar en prestaciones, al incorporar dentro de sí 4 sensores nada desdeñables en una sola placa. Al hacer uso de I2C como protocolo de comunicación permitiría la opción de incorporar todavía más sensores al bus con la facilidad de uso de este protocolo tan bien conocido.

Este mismo dispositivo nos permitirá obtener las mediciones de altitud, orientación y aceleración del dispositivo, además de ángulos de brújula.

- **Módulo GPS GY-NEO6MV2:** Permitirá obtener coordenadas GPS en tiempo real. Costo estimado: 5€.



Figura 9: *Módulo GPS*

Este es un módulo realmente imprescindible para la navegación a lo largo del globo terrestre. En este caso disponemos de una placa que incluye todo lo necesario para la comunicación e incorpora además una antena. La comunicación con el microcontrolador será utilizando el puerto serie, algo que nos hubiera gustado que fuera por otro medio pero que nos es asumible ya que solamente disponemos de un dispositivo que utilice esta interfaz.

Nos encontramos con que dispone de una tasa máxima de muestreo de 1Hz. Suficientemente rápido para reconocer en tiempo real nuestra ubicación

- **Emisor - receptor RC** Emisora de radio FlySky FS i6, Receptor FS iA10B



Figura 10: *Emisora y receptor*

La emisora seleccionada podría controlar nuestro avión de forma manual hasta una distancia de 1km, además, incluye una función para poder recibir y visualizar datos de telemetría en el propio mando y así poder examinar datos como la batería u otras opciones personalizadas.

El transceptor que se colocaría en el avión implementa un bus propietario del fabricante al cual se pueden conectar otros dispositivos para que sean capaces de transmitir información empleando radiofrecuencia.

- **Avión RC:** En esta sección se incluyen todos los componentes ya incluídos dentro de un avión entrenador RC. Tales como servos, motores, batería, el fuselaje etc.

5.3. Software

5.3.1. Visual studio code

Como **editor** de código y para crear las rutinas de compilación y programación de la placa se ha utilizado visual studio code.

Es un software con el que ya nos encontramos familiarizados y entre su catálogo de extensiones se encuentra la de ESP-IDF, el SDK del fabricante del ESP32.



5.3.2. ESP-IDF Software development kit

El SDK(Software Development Kit) oficial para programar el microcontrolador ESP-32 es el **ESP-IDF**, el cual está basado en **FreeRTOS** y permite el manejo de elementos típicos de un sistema operativo como las tareas o el planificador.



5.3.3. FlightGear

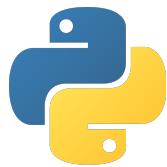
Simulador aeronáutico de código abierto de gran fidelidad. Empleado para evaluar el comportamiento y rendimiento de los sistemas de control. Encontraremos la explicación en detalle de las capacidades de este simulador y el por qué de su elección en el apartado 7.6.1.



5.3.4. Programación

Para programar la lógica del microcontrolador se ha utilizado el lenguaje **C++** junto con **C**. Para otros elementos de utilidad de comunicación o programación se ha empleado **Python**. Todo ello compilado/ejecutado en un sistema **Debian**.

El lenguaje C se encuentra principalmente determinado porque es el que se utiliza para desarrollar software en **ESP-IDF**.



5.3.5. Suite LibreOffice

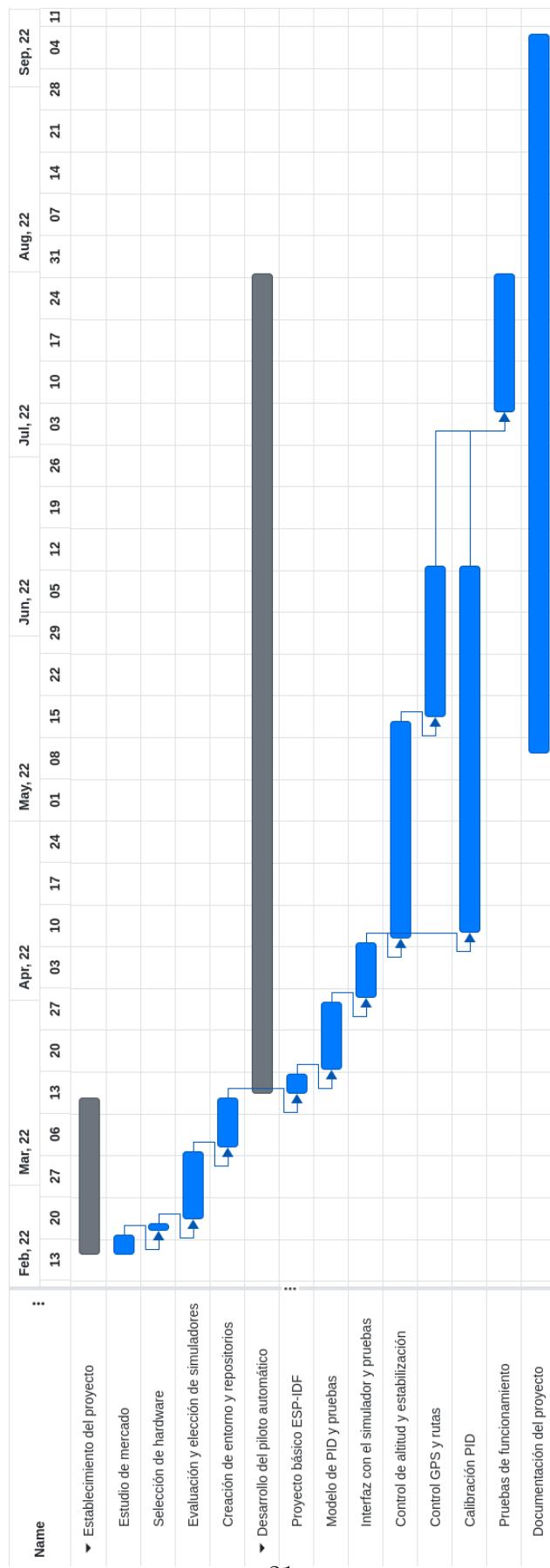
A la hora de elaborar documentación hemos empleado la suite LibreOffice, de código abierto.



5.4. Planificación

Con el siguiente *diagrama de Gantt* tenemos como objetivo representar a grandes rasgos todas las tareas llevadas a cabo durante este proyecto. Las fechas que en la tabla aparecen serán orientativas. El régimen de tiempo dedicado al proyecto es variable en función de los períodos de disponibilidad.

*Las flechas que aparecen enlazando las distintas tareas se corresponden a dependencias, es decir, se ha de terminar la primera antes de poder continuar a la siguiente.



5.4.1. Desglose de tareas

- **Estudio de mercado:** análisis sobre si las necesidades que se plantean resolver con el proyecto son reales y efectivamente supone una ventaja con respecto a los demás.
- **Selección de hardware:** investigación para poder evaluar y discernir de entre todos los posibles microcontroladores y sensores aquellos que se ajustan a las necesidades del proyecto.
- **Evaluación y elección de simuladores:** seleccionar de entre todos los posibles simuladores un candidato válido que cumpla con todas las necesidades tanto a nivel de simulación como de comunicación con la placa.
- **Creación de entorno y repositorios:** creación de una infraestructura que mejore la redundancia y estabilidad del proceso de desarrollo.
- **Proyecto básico ESP-IDF:** documentación, comprensión y arranque inicial de la placa con un código de ejemplo para probar su funcionamiento y el correcto entendimiento del SDK ESP-IDF.
- **Modelo de PID y pruebas:** elaboración de un modelo PID versátil y pruebas que verifiquen su funcionamiento.
- **Interfaz con el simulador y pruebas:** creación de un método de comunicación con el simulador por puerto serie para poder transferir los datos de simulación o actuación de forma fiable.
- **Control de altitud y estabilización:** dotación al piloto automático de un control sobre los actuadores principales para poder despegar y mantener estable la aeronave. También realizar la lectura de los sensores para poder reaccionar en consecuencia del entorno.
- **Control GPS y rutas:** dotación al piloto automático de la capacidad de almacenar rutas GPS y controlar la aeronave para realizar su seguimiento.
- **Calibración PID:** certificación de que los valores de cada actuador PID son los correctos para cada fase del desarrollo. Calibración a base de prueba y error.
- **Pruebas de funcionamiento:** elaboración, ejecución y comprobación de una batería de pruebas que certifiquen la precisión, rendimiento y fiabilidad del sistema.
- **Documentación del proyecto:** elaboración de la documentación tales como la memoria y las diapositivas de presentación del proyecto.

6. Entorno de desarrollo

6.1. Repositorio

Con la finalidad de poder seguir un control del código elaborado y establecer cierta redundancia se ha creado un repositorio en la plataforma GitHub, es posible consultarla en el siguiente enlace: <https://github.com/Ampaex/LibelulaX>

Accediendo a la web podemos ver un desglose de los lenguajes utilizados y el contenido del repositorio.

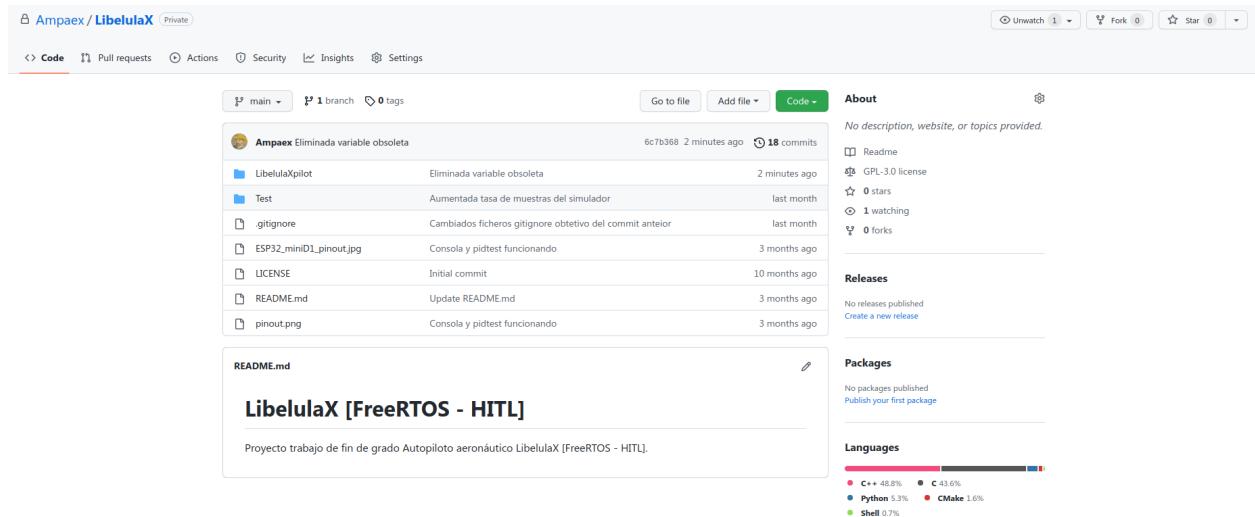


Figura 12: Página principal del proyecto en la plataforma github.

7. Desarrollo

7.1. ESP-IDF - Proyecto básico

ESP-IDF Se trata de un software que proporciona el mismo **fabricante** el cual se encuentra basado en **FreeRTOS**. El proveedor se ha encargado de adaptarlo a las necesidades del microcontrolador para hacer un poco más sencillo su manejo. Una vez creada la estructura del proyecto conforme a sus especificaciones podremos destacar de entre sus funcionalidades un menú de configuración del proyecto o poder compilar o limpiarlo con un solo comando.

Este menú de configuración se llama **menuconfig** y permite una **personalización considerable** del proyecto, pudiendo incluso añadir opciones propias utilizando el sistema de kconfig:

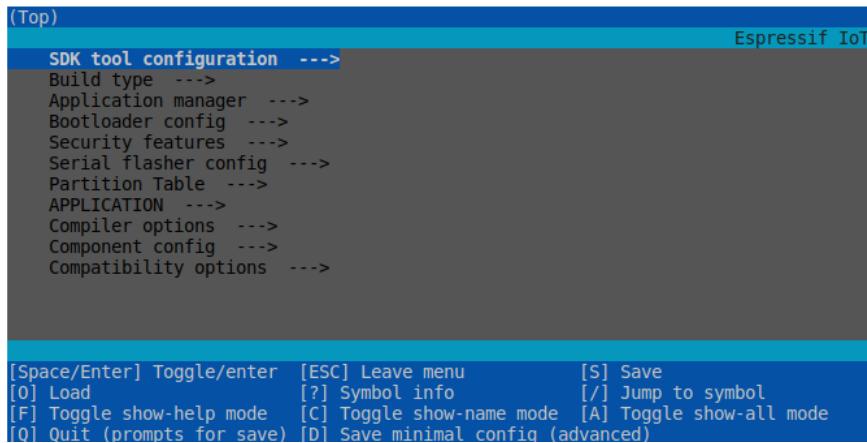


Figura 13: *Interfaz de configuración del proyecto(menuconfig)*

Encontramos también muy útil la **documentación web** de esta herramienta en la que podemos encontrar bastantes ejemplos sobre como utilizar casi todas las funcionalidades del microcontrolador con este sistema operativo FreeRTOS(ESP-IDF).

The screenshot shows the official **ESP-IDF Programming Guide** website. The left sidebar has a dark background with the **ESPRESSIF** logo at the top. It includes dropdown menus for 'ESP32' and 'master (latest)', and a 'Search docs' input field. The main content area has a light background. At the top, it shows the title 'ESP-IDF Programming Guide' and a 'Edit on GitHub' link. Below the title, there is a section for 'API Reference' with a link to '[中文]'. The main text states: 'This is the documentation for Espressif IoT Development Framework (esp-idf). ESP-IDF is the official development framework for the [ESP32](#), [ESP32-S](#) and [ESP32-C Series SoCs](#).'. A note below says: 'This document describes using ESP-IDF with the ESP32 SoC. To switch to a different SoC target, choose target from the dropdown in the upper left.' Below this, there is a grid of icons representing different sections: a timer for 'Get Started', a book for 'API Reference', a chip for 'H/W Reference', a compass for 'API Guides', a puzzle piece for 'Contribute', and a speech bubble for 'Resources'.

Figura 14: *Página de documentación y ayuda de ESP-IDF.*

7.2. Conexión HITL

Este tipo de conexión tiene como objetivo el de **simular** tanto los **sensores** como los **actuadores** para "hacer creer" al microcontrolador que se encuentra realmente reaccionando a estímulos. Esta funcionalidad nos permitirá poder **probar** el dispositivo en todo tipo de situaciones y además con la ventaja de que no se pone en riesgo ningún componente físico para las pruebas.

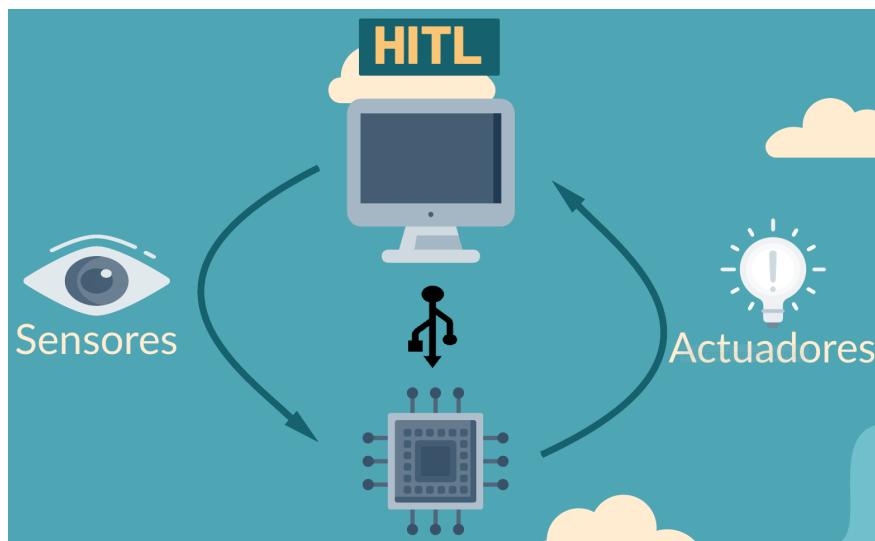


Figura 15: *Flujo de hardware in the loop.*

El PC, en la parte superior de la figura, ejecutará un **simulador aeronáutico** y se encargará de **aportar** los valores que informan sobre la **situación** en la que se encuentra la aeronave al microcontrolador, en la parte inferior, el dispositivo responderá con los valores de los **actuadores** que decidirá establecer para **controlar** el avión.

Haremos uso de un protocolo de interconexión personalizado para recibir los valores de los sensores y comunicar la posición de los actuadores.

7.3. Control de la aeronave

Vamos a explicar a continuación todas las variables de entrada a tener en cuenta y de qué forma vamos a poder modificar los actuadores para así poder lograr el efecto de control deseado.

7.3.1. Superficies de control

Los tres ejes principales en los que se puede mover una aeronave son los siguientes:

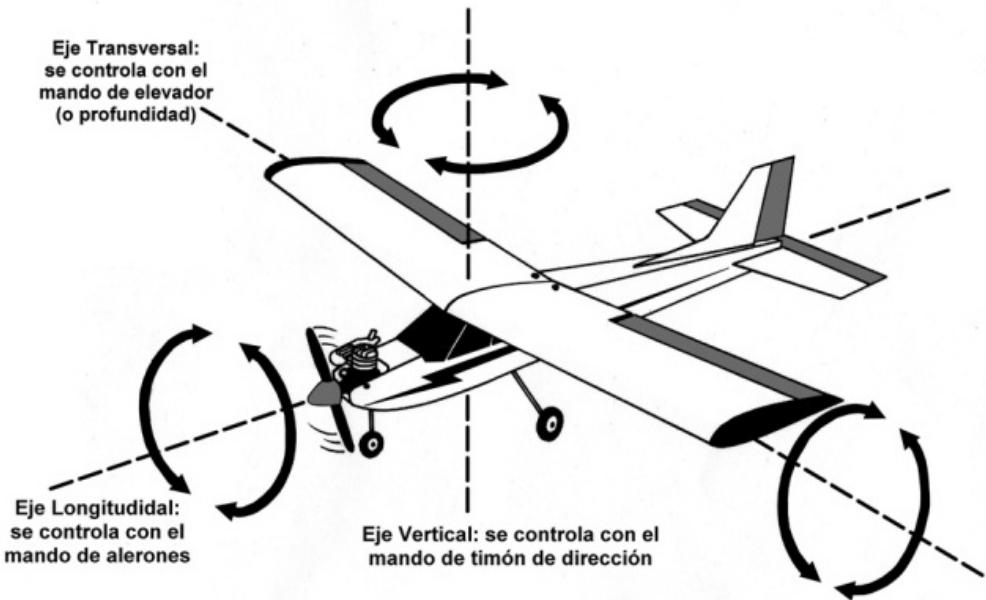


Figura 16: *Ejes de movimiento de una aeronave.* («Blog aerodeporte», 2022)

- En el **eje vertical** tendremos que utilizar el **timón de cola** para así variar el ángulo en esta dirección. También se le conoce como eje de guiñada.
- En el **eje transversal** utilizamos el **elevador**. También se le conoce como eje de cabeceo.
- En el **longitudinal** utilizamos **alerones**. También se le conoce como eje de alabeo.

Haciendo uso de estos actuadores tendremos como objetivo ofrecer al microcontrolador una forma fiable de interactuar con el medio para lograr las inclinaciones necesarias para llevar el avión a su destino.

7.4. Creación del modelo PID

7.4.1. Introducción

El PID es un **mecanismo de control** que, a través de un **lazo de retroalimentación** realiza cálculos de tipo **integral, derivada y proporcional** sobre la salida actual y la entrada para ofrecer la siguiente salida. Este tipo de control es especialmente útil en casos en los que los sistemas a controlar poseen una inercia y se debe de tener cierta anticipación para no comenzar a oscilar con la salida.

En la siguiente figura tenemos un esquema del flujo de datos que sigue este control:

- $r(t)$ es el **valor objetivo**, el que queremos alcanzar con este control.
- $y(t)$ es el **valor actual** en el que se encuentra el sistema.
- $e(t)$ es el error del sistema en base a los dos valores anteriores.

- Los bloques P, I y D son el cálculo proporcional, integral y derivado respectivamente.
- $u(t)$ es la **salida del control PID**, la que se encargará de controlar un actuador para modificar el estado del sistema. Se conforma por la suma de los bloques P, I y D.

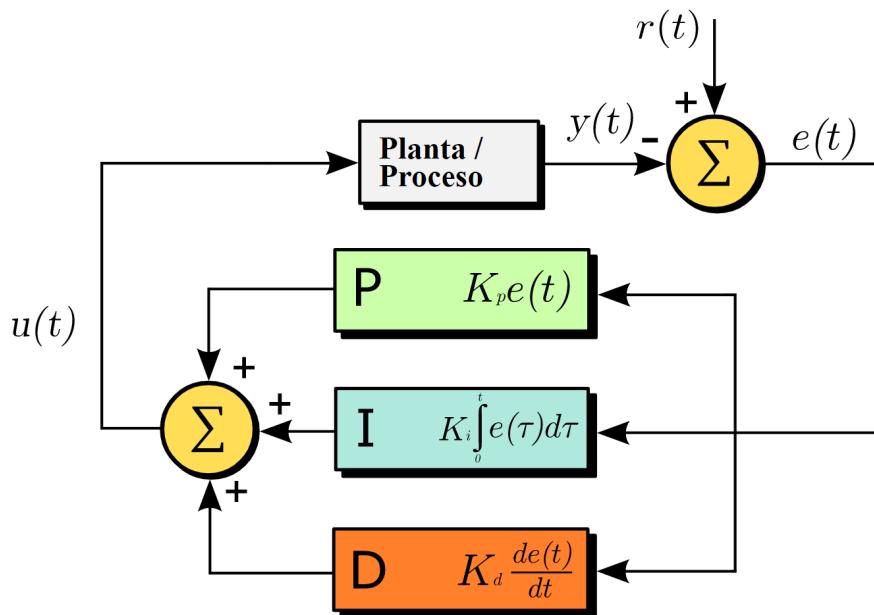


Figura 17: Esquema de flujo de un PID

Cada uno de los bloques P, I y D tendrán un **valor de ponderación**, los cuales se utilizarán para dar mayor o menor importancia a cada componente de tal forma que modelaremos así un tipo de comportamiento, ya sea por ejemplo una respuesta **más rápida o más lenta**.

Una **caso de uso** de este sistema podría ser un **calentador** que tiene que controlar la temperatura del agua y mantenerla lo más estable posible.

En este caso:

- El **valor objetivo** del sistema sería la temperatura que deseemos establecer.
- El **valor actual** sería la temperatura del agua al momento del cálculo.
- La **salida** sería la potencia a la que el sistema decide poner el calentador.

El controlador debería tener en cuenta que, en el caso de calentar, cuando la temperatura se encuentre próxima al valor objetivo debería ir reduciendo con anticipación la potencia del calentador, para así no excederse.

Debido a que este proyecto de automatización se basa fundamentalmente en un control de tipo PID se ha de comenzar por elaborar un **modelo fiable y probado** de control PID.

7.4.2. Implementación

Se partió de la implementación de un objeto C++ básico de un modelo de PID. Dicha base se tuvo que adaptar a las necesidades del proyecto para además añadir funcionalidades adicionales requeridas por el tipo de aplicación.

Finalmente estas son las funcionalidades más destacables:

- Limitación del rango de valores de salida.
- Cálculo del punto medio y de la amplitud de la salida.
- Posibilidad de crear un control circular, es decir, interpretar los límites como si el principio y el final se encontraran unidos. Como en una brújula.
- Posibilidad de deshabilitar y habilitar el PID.
- Reinicio del control.
- Capacidad de establecerlo como un PID secundario, lo que significa que se encuentra en segundo lugar controlando un actuador, es decir que entre el actuador y este se encuentra otro PID.

7.4.3. Pruebas de control

Para poder evaluar el control en este primer momento en el que todavía no se disponía de un simulador acoplado se hizo necesario crear un **banco de pruebas**. El test consistiría en simular el comportamiento del elevador de la aeronave, tomando como objetivo establecer una inclinación deseada haciendo uso únicamente de este actuador.

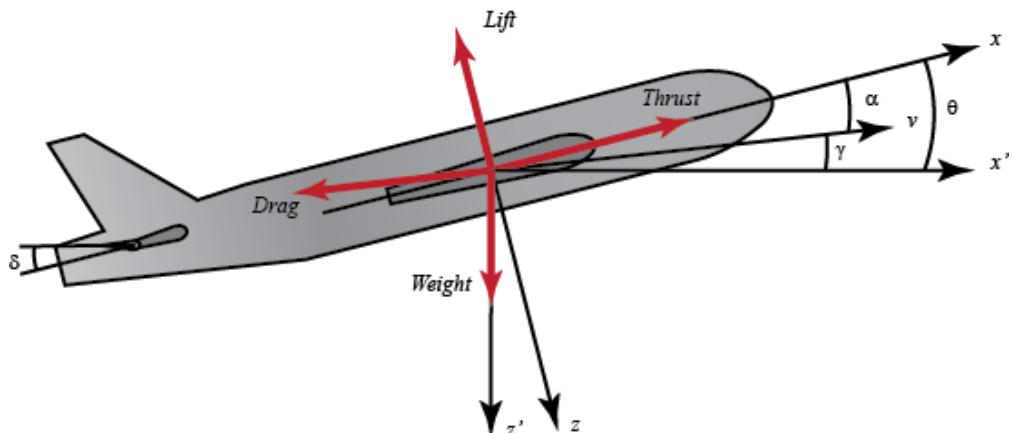


Figura 18: Esquema del cabeceo de la aeronave.

Modelo de PID:

- Entrada: Inclinación de la aeronave y ángulo deseado.

- Salida: Posición del elevador.

Este test se encuentra implementado en **python** y realiza el siguiente proceso:

1. Apertura de la conexión por puerto serie.
2. Bucle
 - Envío de comando de estado del sistema hacia el puerto serie.
 - Recepción y decodificación de la acción del microcontrolador.
 - Simulación de un modelo de cinemática uniformemente acelerado.

Para poder visualizar los resultados y analizar si se encuentra funcionando de forma esperada se utilizó la herramienta llamada **gnuplot** para graficar los datos. En cada iteración del bucle se procedería a actualizar el gráfico. Estos son los gráficos resultantes de los distintos casos de prueba utilizando distintos valores de calibración:

LEYENDA

Eje X: tiempo (s).

Eje Y: valor del parámetro correspondiente.

Elevator: posición del elevador.

Inclination: inclinación de la aeronave.

Target: ángulo objetivo.

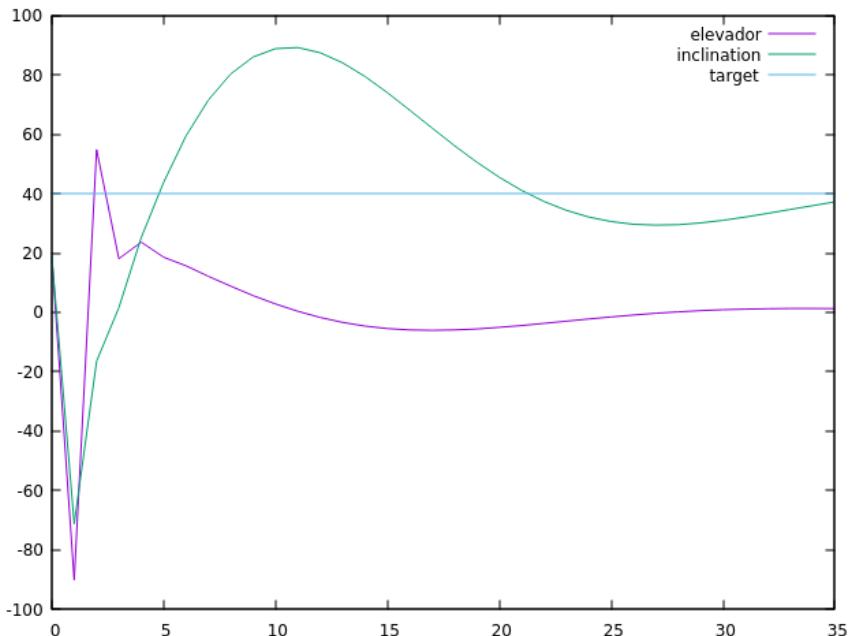


Figura 19: Caso de prueba 0 PID

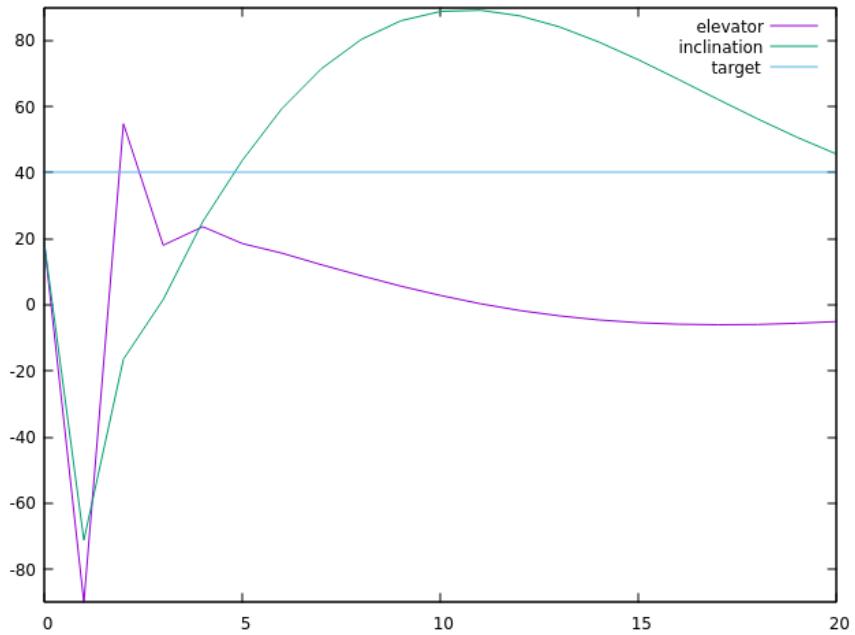


Figura 20: Caso de prueba 1 PID

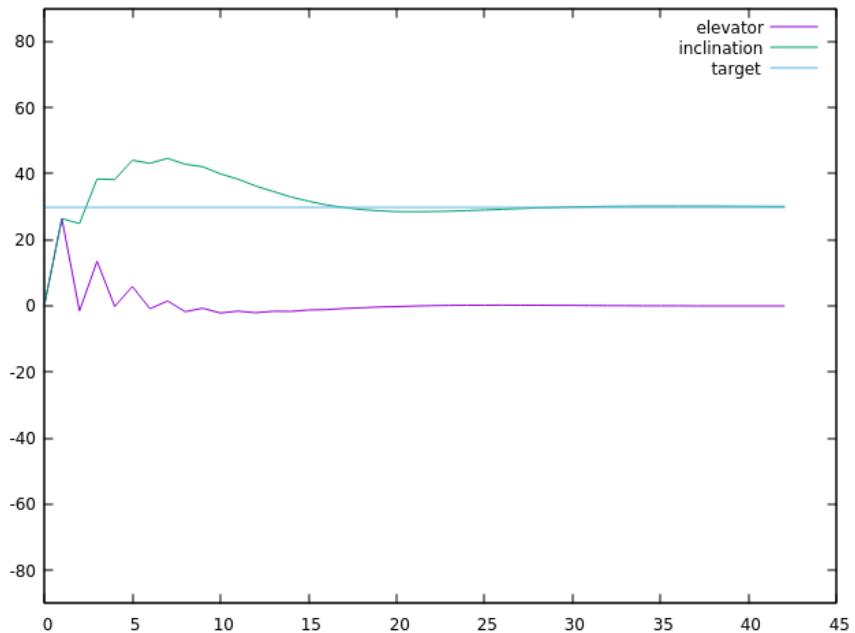


Figura 21: Caso de prueba 2 PID

Las **conclusiones** que podemos obtener de los gráficos es que, a falta de una calibración óptima, podemos observar la convergencia y el correcto funcionamiento del modelo de control PID.

7.5. Creación de una consola por puerto serie

Desde el principio, la **comunicación** comenzó a plantear uno de los grandes retos del proyecto, el primero de ellos fue el de cómo se iba a lograr una **comunicación fiable** entre el PC y el microcontrolador.

Se hizo necesaria una implementación con cierta complejidad para poder albergar múltiples **comandos** y también permitir cierta variabilidad en los **parámetros**.

La solución escogida será la implementación de una consola que se encuentra dentro del entorno de desarrollo ESP-IDF, aunque este cuenta con escasa información o documentación.

Tras probar y examinar su funcionamiento se consiguieron incorporar nuevas funcionalidades y comandos a dicha consola, lo que nos permitirá posteriormente establecer la deseada comunicación por comandos.

La terminal implementada se encuentra realizada tal y como nos la podríamos encontrar en un sistema operativo como linux. Tenemos a continuación una captura del comando *help*, el cual muestra una ayuda de todos los comandos que tenemos disponibles. Estos nos dan una libertad de acción casi total.

```

help
    Print the list of registered commands

free
    Get the current size of free heap memory

heap
    Get minimum size of free heap memory that was available during program execu
    tion

version
    Get version of chip and SDK

restart
    Software reset of the chip

deep_sleep [-t <t>] [--io=<n>] [--io_level=<0|1>]
    Enter deep sleep mode. Two wakeup modes are supported: timer and GPIO. If no
    wakeup option is specified, will sleep indefinitely.
    -t, --time=<t> Wake up time, ms
        --io=<n> If specified, wakeup using GPIO with given number
        --io_level=<0|1> GPIO level to trigger wakeup

light_sleep [-t <t>] [--io=<n>]... [--io_level=<0|1>]...
    Enter light sleep mode. Two wakeup modes are supported: timer and GPIO. Mult
    iple GPIO pins can be specified using pairs of 'io' and 'io_level' arguments
    . Will also wake up on UART input.
    -t, --time=<t> Wake up time, ms
        --io=<n> If specified, wakeup using GPIO with given number
        --io_level=<0|1> GPIO level to trigger wakeup

join [--timeout=<t>] <ssid> [<pass>]
    Join WiFi AP as a station
    --timeout=<t> Connection timeout, ms
        <ssid> SSID of AP
        <pass> PSK of AP

nvs_set <key> <type> -v <value>
    Set key-value pair in selected namespace.
Examples:
    nvs_set VarName i32 -v
        123
    nvs_set VarName str -v YourString
    nvs_set VarName blob -v 0123456789
        abcdef
            ↗ Shows key of the value to be set

```

Figura 22: Salida del comando `help` en el terminal serie. Tenemos comandos de utilidad para poder ver la memoria restante, la versión del sdk o reiniciar el dispositivo.

7.6. Elección y control del simulador

Un punto crucial dentro del desarrollo fue la elección del simulador de vuelo debido a que ha de ser capaz de ofrecernos una **interfaz válida** que seamos capaces de utilizar para poder **comunicarnos por usb** con el fin de posteriormente implementar un hardware in

the loop(HITL).

Además ha de tener disponible algún modelo de avión radiocontrol para que las pruebas sean lo más fiel posible a la realidad.

7.6.1. Valoración de alternativas

Para este proyecto solamente tuvimos en consideración simuladores que permitieran simular aviones radio-control y además fueran gratuitos. Procedamos a ver como fueron evaluados los que más se acercaban a nuestras necesidades.

PICASIM

Este software es frecuentemente utilizado por personas que quieren practicar antes de comenzar a volar con sus modelos reales de aviones radio-control. Aunque es una opción gratuita tuve serios problemas para encontrar documentación al respecto, además de que no dispone de versión para sistema operativo linux.



CRRCsim

Esta alternativa es de código abierto y por lo tanto, gratuito. El proyecto se encuentra en este momento en práctico abandono y no dispone, que se haya encontrado, de ninguna interfaz, protocolo o API que permita tener información acerca del estado de ejecución del programa. La única opción que se pensaba viable al comienzo del proyecto era la de editar y compilar el código fuente para introducir alguna forma de comunicación. Esta opción fue posteriormente descartada al tener en cuenta el siguiente candidato.



FLIGHTGEAR

La opción finalmente escogida. En primera instancia no entraba dentro de las posibles alternativas ya que no se trata de un simulador enfocado a las aeronaves radio-control, sino de un simulador de alta fidelidad de aviones comerciales reales.

En la búsqueda dimos por casualidad con la existencia de una aeronave dentro del simulador de tipo radio-control, el *Rascal110*, que fue creada para un proyecto de la universidad de Minnesota con la finalidad de hacer pruebas de aeronave no tripulada.

El modelo real es un avión fabricado por *"Sig manufacturing."*.

Además, el simulador también posee un método de elaborar un protocolo personalizado de

comunicación con el simulador y una interfaz gráfica para la visualización del vuelo en un mapa.

Resultó ser una opción bastante adecuada.



7.7. Protocolo de interconexión

Para lograr la **intercomunicación entre el microcontrolador y el simulador** necesitamos un protocolo que sea conocido por ambos dispositivos y bidireccional.

Esto lo hemos conseguido gracias a una funcionalidad del simulador flightgear para crear un **protocolo personalizado** gracias a la especificación del mismo en forma de archivo xml. Se ha creado un **método de comunicación propio** tanto de entrada como de salida.

7.7.1. Variables de entrada

Para que el pilotado sea el correcto será necesario conocer los siguientes **datos acerca del entorno**:

- Altitud (alt).
- Ubicación GPS (Latitud y longitud)(lat, lon).
- Ángulo de guiñada(yaw).
- Ángulo de alabeo(rol).
- Ángulo de cabeceo(pitch).

Estas variables se utilizarán para evaluar el estado del avión en cada momento y actuar en consecuencia.

7.7.2. Variables de salida

Una vez se ha realizado el procesado en el microcontrolador se devuelven los siguientes datos al simulador:

- Alerón(aileron).
- Elevador(elevator).
- Timón de cola(rudder).
- Aceleración(throttle).

Estas variables se utilizarán para modificar la posición de los **actuadores** dentro del simulador.

7.7.3. LibelulaXProtocol

La especificación del formato consiste en una serie de campos xml a llenar de forma que obtengamos el patrón de comunicación que deseemos.

- **Salida del simulador:** Debido a que en el terminal del microcontrolador disponemos de un **terminal de comandos** el patrón de **salida** del simulador ha de tener el **formato de un comando** estilo linux. Ejemplo: *sim_flight_data -pit=30 -rol=45*
- **Entrada del simulador:** En el sentido hacia el simulador se enviarán todas las posiciones de los actuadores en formato float y separados por comas. Ejemplo: *0.4,0.2,0.6*

A continuación, una imagen con parte de la especificación del **comando enviado al microcontrolador**.

```

<output>
    <binary_mode>false</binary_mode>
    <line_separator>\n</line_separator>
    <var_separator> </var_separator>
    <preamble></preamble>
    <postamble></postamble>

    <chunk>
        <name>Pitch</name>
        <node>/orientation/pitch-deg</node>
        <type>double</type>
        <format>sim_flight_data --pit=%f</format>
    </chunk>

    <chunk>
        <name>Roll</name>
        <node>/orientation/roll-deg</node>
        <type>double</type>
        <format>--rol=%f</format>
    </chunk>

    <chunk>
        <name>Yaw</name>
        <node>/orientation/yaw-rate-degps</node>
        <type>double</type>
        <format>--yaw=%f</format>
    </chunk>

    <chunk>
        <name>Altitude</name>
        <node>/position/altitude-agl-ft</node>
        <type>double</type>
        <format>--alt=%f</format>
    </chunk>

```

Figura 23: Especificación del protocolo en sentido hacia el microcontrolador.

En este fragmento podemos observar, entre otros, que podemos elegir cómo se van a separar las cadenas(line.separator), separadores entre variables(var.separator) y cada uno de los datos que se van a enviar, encasillados en la etiqueta `<chunk>`.

Podemos fijarnos también en que en la primera casilla de datos, la casilla formato, va incluido el comando tipo consola que recibirá el microcontrolador. Cada uno de los parámetros va en forma de parámetro de comando linux, de esta forma, aunque se alterase el orden o faltara alguno de ellos no causaría ningón fallo.

Este comando cumple la función de aportar información al microcontrolador acerca de las condiciones en las que se encuentra en ese momento la aeronave.

De similar forma, aunque con menos opciones, para la **parte de recepción** de comandos de estado del microcontrolador también tenemos otro apartado.

```

<input>

    <binary_mode>false</binary_mode>
    <line_separator>\n</line_separator>
    <var_separator>,;</var_separator>

    <chunk>
        <name>Elevator</name>
        <node>/controls/flight/elevator</node>
        <type>float</type>
    </chunk>

    <chunk>
        <name>Aileron</name>
        <node>/controls/flight/aileron</node>
        <type>float</type>
    </chunk>

    <chunk>
        <name>Rudder</name>
        <node>/controls/flight/rudder</node>
        <type>float</type>
    </chunk>

    <chunk>
        <name>Throttle</name>
        <node>/controls/engines/engine/throttle</node>
        <type>float</type>
    </chunk>

</input>
```

Figura 24: Especificación del protocolo en sentido desde el microcontrolador.

Este aporta al simulador en qué posición tiene que poner cada uno de los controles o superficies de control para así poder ir manejando la nave.

Utilizaremos un parámetro para poder indicar al simulador que utilice este protocolo y a qué frecuencia ha de escanear se indicará en el comando de arranque del software. Estos son los parámetros a incluir en el parámetro del comando de arranque del simulador:

```
--generic=file,in,3,inFile,LibelulaXProtocol
--generic=file,out,3,outFile,LibelulaXProtocol
```

Esto indica; el tipo de protocolo, el sentido del flujo, la frecuencia de muestreo, el nombre del archivo de donde recoger o ubicar la salida/entrada y el nombre del archivo de protocolo respectivamente.

7.7.4. Inconvenientes del protocolo ⇒ script de comunicación

En el momento de las **pruebas**, el protocolo no funcionaba siguiendo las especificaciones, parece ser que la **funcionalidad bidireccional** no estaba todavía bien implementada. El problema que tenía era que **no funciona** utilizando la misma interfaz (o archivo) tanto para entrada como para salida, por lo que tuvimos que **crear** un **código intermedio** que se encargara de separar las salidas y entradas en dos archivos e ir transfiriendo los mensajes. Es la razón de por qué, en los parámetros acabados de mencionar, existe un "inFile" y un "outFile", y no uno solo. Son los correspondientes a **ambos sentidos** de la comunicación.

Este código intermedio o script lo hemos llamado *serialparser*, e incluye la funcionalidad de: comunicarse por puerto serie con el microcontrolador, transmitir los mensajes y separar en dos archivos las comunicaciones según el sentido.

En el siguiente diagrama podemos ver gráficamente el flujo de datos que engloba al script:

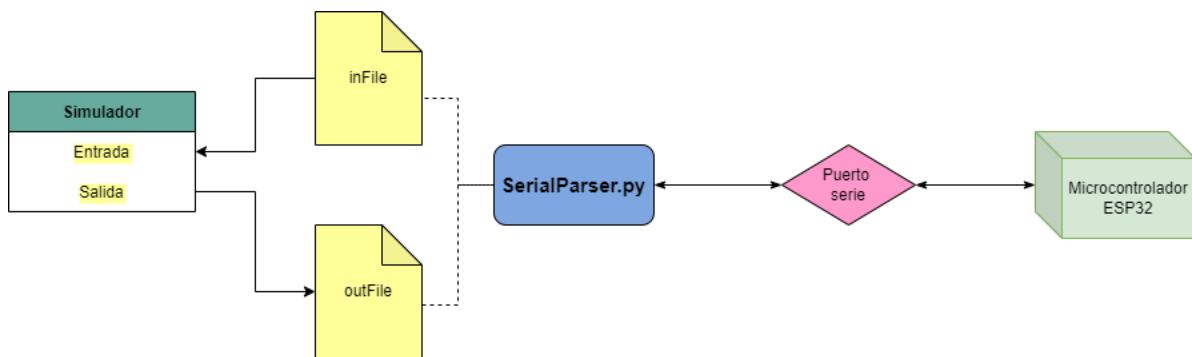


Figura 25: Esquema de funcionamiento de script `serialParser.py`

7.8. Brújula y guiado GPS

Para poder tratar los datos del GPS y encontrar el ángulo de brújula al que deberíamos dirigirnos se ha tenido que investigar acerca del funcionamiento del GPS y como hacer cálculos de posicionamiento global.

Examinaremos a continuación ciertos conceptos necesarios para llevarlo a cabo.

7.8.1. Posicionamiento GPS

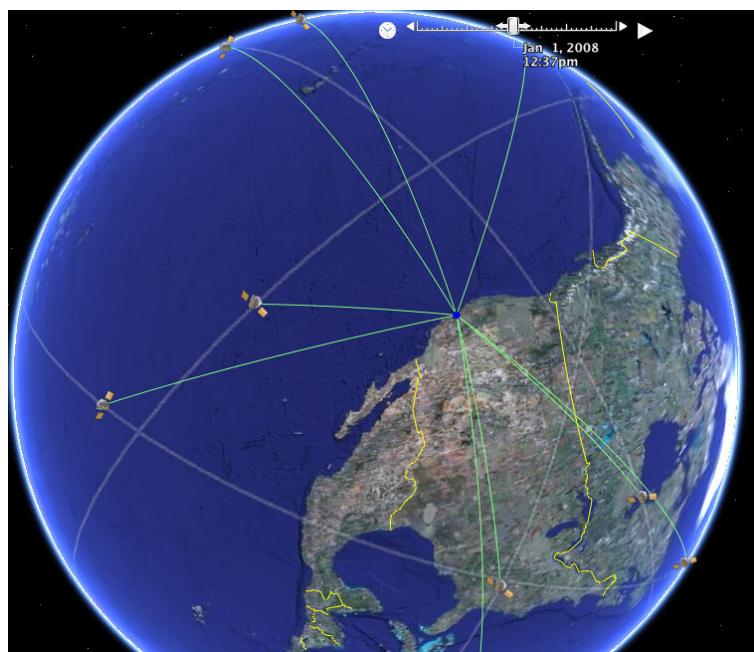


Figura 26: Imagen google earth, satélites GPS.

Citando la página web de Geotab («Geotab ¿Qué significa GPS?», 2022), el GPS funciona de la siguiente forma:

”El GPS funciona a través de una técnica llamada trilateración. Utilizada para calcular la ubicación, la velocidad y la elevación, la trilateración recopila señales de los satélites para enviar información de ubicación.”

”Los satélites que orbitan la Tierra envían señales para que las lea e interprete un dispositivo GPS situado en la superficie de la Tierra o cerca de ella. Para calcular la ubicación, un dispositivo GPS debe poder leer la señal de al menos cuatro satélites.”

Con esta breve pero útil explicación podemos resumir que, el módulo GPS se apoya en una serie de satélites para recabar información acerca de la ubicación, velocidad actual y altitud, parámetros esenciales para el control de la avioneta.

Para la explicación de lo que nos acontece a continuación vamos también a remarcar que el valor de ubicación se compone de dos datos.

- Latitud: Es la distancia medida en grados a partir del ecuador del globo terráqueo.
- Longitud: Es la distancia medida en grados a partir del meridiano de greenwich.

El formato de coordenadas es el par de latitud y longitud, con signo menos (-) para la dirección latitud sur o longitud oeste, aquí un ejemplo:

Formato

- 52.5163 , 13.3779 (N, E)

- 40.7682 , -73.9816 (N, O)
- -22.9708 , -43.1830 (S, O)

7.8.2. Distancia entre dos puntos

Para calcular la distancia más corta entre dos puntos sobre la forma geodésica de la tierra **se aproxima**, como forma de simplificación, a lo que sería **una esfera** de radio

$$R = 6371 \text{ km} \text{ (radio cuadrático medio)}$$

Debido a esta aproximación a una esfera, el cálculo dispone de un **error de distancia** de $\pm 3\%$. Particularmente acentuada en casos de extremos polares o largas distancias a través de varios paralelos.

Vamos a utilizar la fórmula "haversine" para calcular la distancia más corta sobre la superficie de la tierra, dando una distancia "en línea recta" entre los puntos.

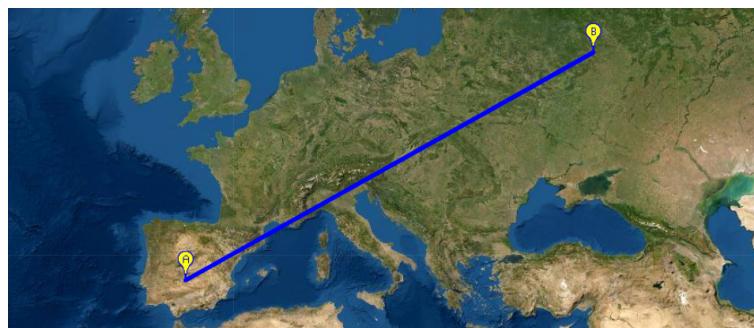


Figura 27: Distancia entre A y B.

Por lo tanto, dados dos puntos A y B de la esfera expresados por latitud (lat) y longitud (lon) tenemos que la función de cálculo de distancia queda de la siguiente manera:

$$\begin{aligned}\mathbf{a} &= \sin\left(\frac{\Delta \text{lat}}{2}\right)^2 + \cos(\text{lat}A) \cdot \cos(\text{lat}B) \cdot \sin\left(\frac{\Delta \text{lon}}{2}\right)^2 \\ \mathbf{b} &= 2 \cdot \text{atan2}(\sqrt{\mathbf{a}}, \sqrt{1 - \mathbf{a}}) \\ \mathbf{d} &= R \cdot \mathbf{b}\end{aligned}$$

Donde "R" el radio de la tierra.

Los ángulos utilizados han de expresarse en radianes, la conversión entre grados y radianes se obtiene multiplicando el ángulo por π y dividiendo por 180.

Esto, programáticamente hablando se ha traducido en la siguiente función en lenguaje C++:

```

double GPS::distanceBetweenCoord(GPS_coordinate coord_A, GPS_coordinate coord_B)
{
    GPS_coordinate _A = coord_A.toRadians();
    GPS_coordinate _B = coord_B.toRadians();

    // Haversine Formula
    double dlong = _B.longitude - _A.longitude;
    double dlat = _B.latitude - _A.latitude;

    double ans = pow(sin(dlat / 2), 2) +
                | cos(_A.latitude) * cos(_B.latitude) *
                | pow(sin(dlong / 2), 2);

    ans = 2 * asin(sqrt(ans));

    // Radius of Earth in
    // Kilometers, R = 6371
    long double R = 6371;

    // Calculate the result
    ans = ans * R;

    return ans;
}

```

Figura 28: *Código de cálculo de distancias*

Fórmulas y conceptos de («Sunearthtools web», 2022) («Movable type scripts web», 2022).

7.8.3. Rumbo - brújula entre dos puntos

De nuevo, para este cálculo, tendremos que tener en cuenta la forma "esférica" de la tierra y no contentarnos con el mismo resultado que si lo hicéramos de forma vectorial. En este punto podremos tener la distancia, pero para llegar al destino necesitaremos también saber hacia a donde ir, esto lo otorga el rumbo(en inglés *bearing*), que viene dado por un ángulo de brújula.



Figura 29: *Por ejemplo, el camino seguido para ir desde Buenos Aires a Pekín por la distancia más corta.*

En general, el rumbo presente variará a medida que siga la ruta de círculo máximo (ortodrómica); el rumbo final diferirá del rumbo inicial en diversos grados según la distancia y la latitud (si se tuviera que pasar de, digamos, 35°N,45°E (Bagdad) a 35°N,135°E (Osaka), comenzaríamos con un rumbo de 60° y la ruta terminaría con un rumbo de 120°).



Figura 30: *Bagdad - Osaka*

Esta fórmula es para el rumbo inicial (a veces denominado azimut directo) que, si se sigue en línea recta a lo largo de un arco de círculo máximo, nos llevará desde el punto inicial hasta el punto final.

Con esto queda claro el problema a resolver, para hacer el cálculo utilizaremos la siguiente fórmula:

$$\begin{aligned} \textbf{Rumbo} = \\ \text{atan2}(\sin(\nabla\text{lon}) \cdot \cos(\text{latB}), \cos(\text{latA}) \cdot \sin(\text{latB}) - \sin(\text{latA}) \cdot \cos(\text{latB}) \cdot \cos(\nabla\text{lon})) \end{aligned}$$

Dado que atan2 devuelve valores en el rango $-\pi \dots +\pi$ (es decir, $-180^\circ \dots +180^\circ$), para normalizar el resultado a un rumbo de la brújula (en el rango $0^\circ \dots 360^\circ$) es necesario convertirlo a grados y luego usar $(\text{angulo}+360) \% 360$, donde $\%$ es módulo.

En nuestro caso, hemos traducido todo esto en una función C++ la cual realizará el cálculo dentro de nuestro microprocesador:

```
float GPS::compassBetweenCoord(GPS_coordinate coord_A, GPS_coordinate coord_B)
{
    GPS_coordinate _A = coord_A.toRadians();
    GPS_coordinate _B = coord_B.toRadians();

    double y = sin(_B.longitude - _A.longitude) * cos(_B.latitude);
    double x = cos(_A.latitude) * sin(_B.latitude) -
    |   |   |   sin(_A.latitude) * cos(_B.latitude) * cos(_B.longitude - _A.longitude);
    float bearing = atan2(y, x);

    // Return the radian angle in form of sexagesimal degree
    return fmod(degrees(bearing) + 360, 360);
}
```

Figura 31: Código de cálculo de rumbo

Para obtener más información acerca de rutas ortodrómicas visitar (también fuente de imágenes): («Web geografía infinita», 2022)

Fórmulas y conceptos de («Sunearthtools web», 2022) y («Movable type scripts web», 2022).

7.8.4. Elaboración y visualización de rutas

Una ruta consiste en una serie de coordenadas por las que el vehículo ha de pasar de forma secuencial a una distancia aceptable para considerar que esta ha sido seguida. En las rutas, en el caso del avión, necesitaríamos incluir además una componente más, esta es la altura, es totalmente necesaria para poder tener el control total y así ser capaces de poder evitar obstáculos.

En este proyecto, dada la naturaleza del objetivo de control de la aeronave, esta no podrá variar su ruta mientras se encuentre ejecutándola, ya que durante la mayor parte del tiempo no tendrá conexión de ningún tipo con tierra.

Una vez finalizada la ruta este volverá al punto de partida y comenzará a orbitar en círculos el mencionado sitio a la espera del control manual por parte de tierra.

PLANIFICACIÓN DE RUTAS

Para planificar una ruta previamente al vuelo servirá cualquier software que sea capaz de generar una lista de coordenadas para luego transferirlas al microcontrolador.

Un software que es muy interesante para esta tarea es el **Mission planner**, de Ardupilot. Este posee múltiples funciones y es muy intuitivo a la hora de manejarlo. Viene pensado para ser utilizado junto al software Ardupilot, pero al dejarnos exportar la ruta, nos sería posible utilizarlo para nuestro fin.

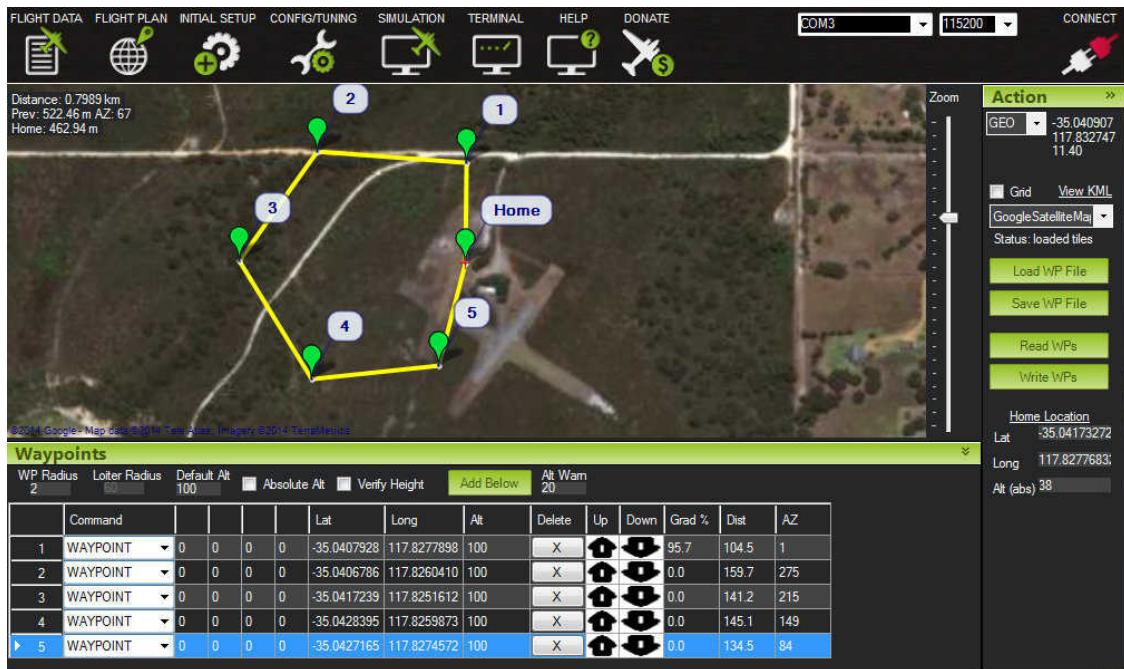


Figura 32: Software mission planner. («Mission Planner software», 2022)

Otro generador de rutas que podemos utilizar sin necesidad de descargar nada sería la página web de GPS Geoplaner («GPS geoplaner web», 2022). En nuestro caso es el que hemos utilizado debido a que también es muy fácil de usar, no contiene elementos extras que no nos sirven de utilidad y además, no es necesario descargarlo. Se puede utilizar desde la misma web.

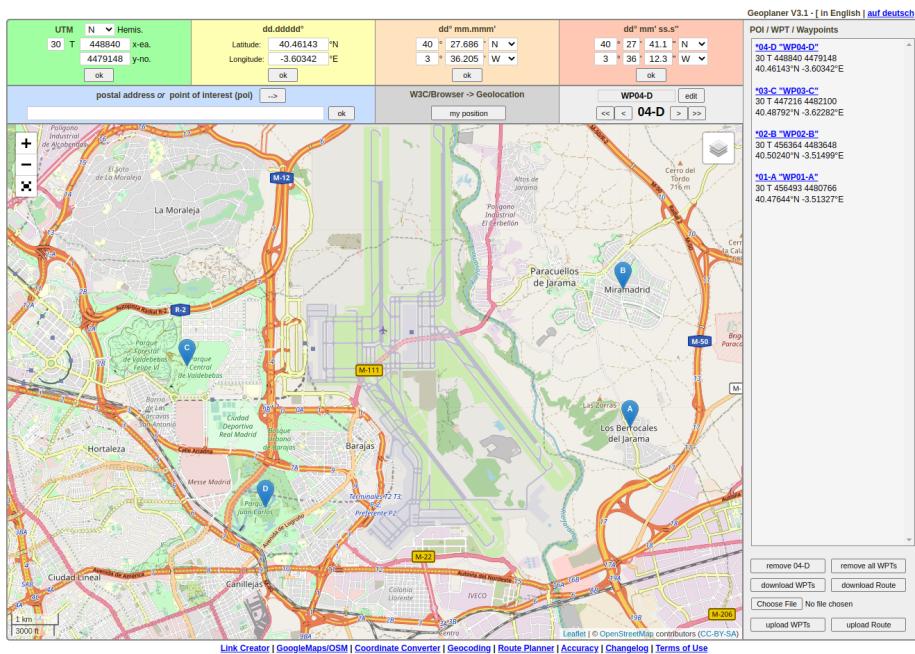


Figura 33: Página de elaboración de rutas Geoplanner.

INTERFAZ FLIGHTGEAR

Muy convenientemente el programa simulador FlightGear nos ofrece una interfaz web a través de la que podemos observar el estado en tiempo real del avión dentro del simulador, así como la trayectoria que ha seguido desde el momento que comenzó el vuelo. Para activarla es necesario introducir un parámetro al momento de arranque del programa para que inicie un servidor HTTP desde el cual, utilizando el navegador, podremos consultar los datos de forma gráfica.

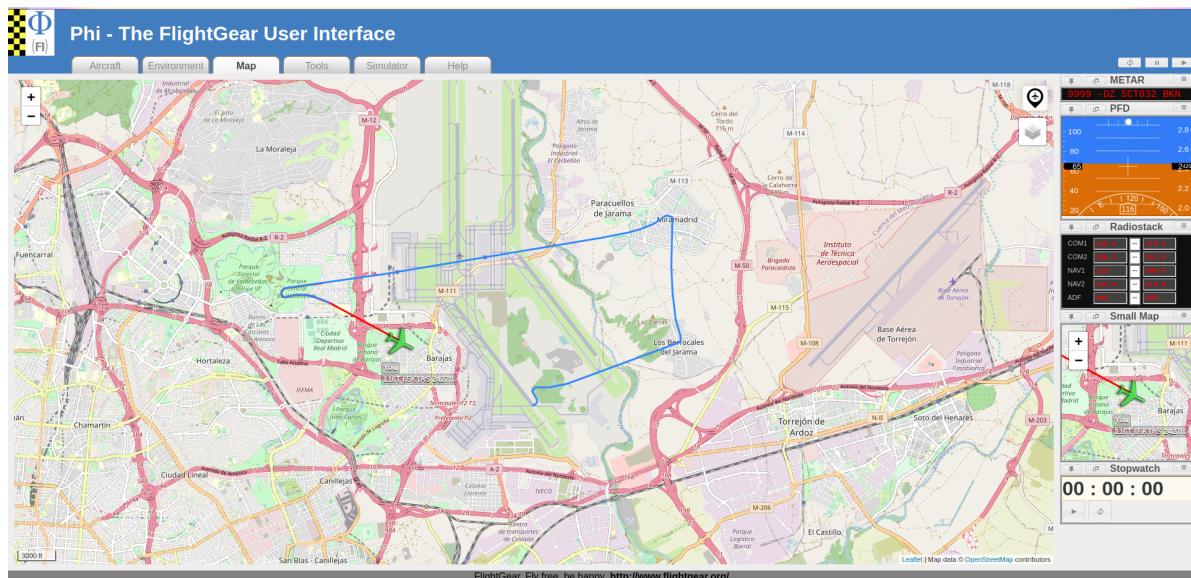


Figura 34: *Interfaz del simulador FlightGear con historial de una ruta realizada por este mismo proyecto.*

Esta interfaz se puede conseguir introduciendo el parámetro: `-httpd=8080`

7.9. Elementos PID y de control

A continuación veremos de qué forma estamos controlando cada uno de los actuadores para tener total control sobre la aeronave y además cumplir con la resiliencia a ráfagas de viento o turbulencias.

7.9.1. Cabeceo - PID



Figura 35: *Figura que detalla el cabeceo del avión («Blog aerodeporte», 2022)*

El cabeceo consiste en el movimiento que se muestra en la figura, y para realizarlo tendremos que emplear el elevador del avión.

El modelo PID de este elemento tendrá las siguientes características:

- **Entrada:** Inclinación del avión en el eje transversal en grados.

- **Objetivo:** Ángulo de inclinación del eje transversal final [90, -90].
- **Salida:** Posición del elevador en un intervalo [1,-1].

7.9.2. Alabeo - PID

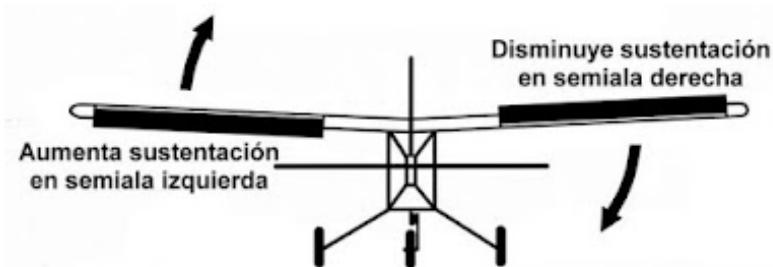


Figura 36: Figura que detalla el alabeo del avión («Blog aerodeporte», 2022)

Cuando realizamos el alabeo del avión tenemos que girar sobre el eje longitudinal, tal y como se muestra en la figura. Para controlar este giro necesitaremos utilizar ambos alerones de forma complementaria, es decir, cuando uno sube el otro baja.

Algunos aviones radiocontrol implementan esta acción por separado, con un servo por alerón, en cambio, en otros, una varilla realiza esta función simplificando a un solo servo. Tendremos que tener en cuenta esto en función del modelo.

En el caso del modelo del simulador se dispone únicamente de un valor para controlar ambos, por lo que sería como tener el segundo caso, el que tiene una varilla.

El modelo PID de este elemento tendrá las siguientes características:

- **Entrada:** Inclinación del avión en el eje longitudinal en grados.
- **Objetivo:** Ángulo de inclinación del eje longitudinal final [90, -90].
- **Salida:** Posición de los alerones en un intervalo [1, -1].

7.9.3. Guiñada

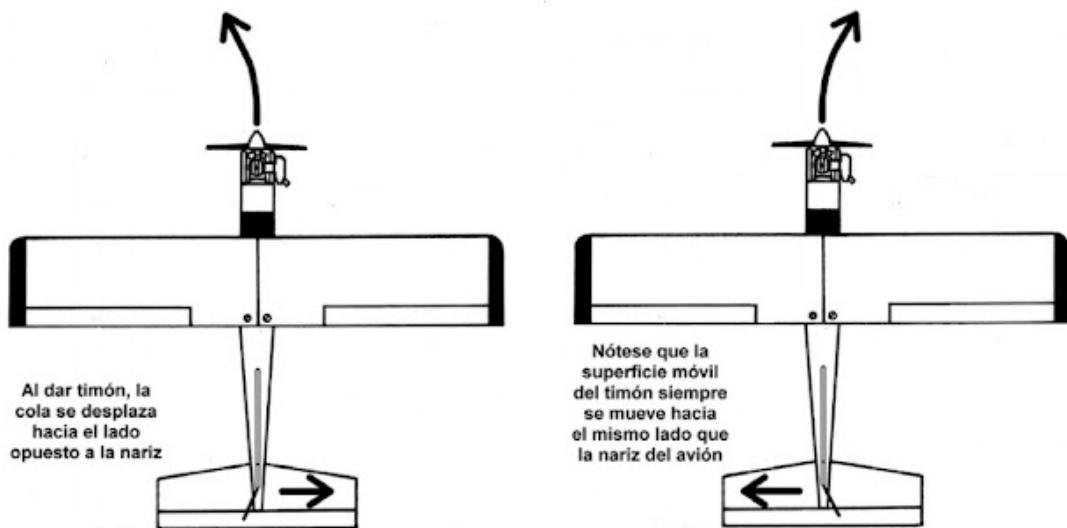


Figura 37: Figura que detalla la guiñada del avión («Blog aerodeporte», 2022)

El eje vertical o de guiñada lo controlaremos modificando el ángulo del timón de cola, esto provocará que el aire que circula por la parte trasera se vea desviado y la cola se moverá hacia el lado opuesto, esto es algo a tener en cuenta a la hora de programar la acción de este control.

Manteniendo constante el resto de ángulos y utilizando este podemos hacer giros muy suaves, bastante convenientes para hacer el seguimiento de rutas.

En este caso no es relevante hacer un PID ya que este eje no es tan fácilmente medible y tampoco influye en la estabilidad de la aeronave. Por eso, su control es proporcional y tomaría un valor de control en el intervalo [1, -1]

7.9.4. Velocidad

Al ser el caso de un modelo a pequeña escala, la velocidad, es algo que no pondremos como objetivo de importancia para dejar cero, aunque podremos hacer su lectura a través del GPS. Para el manejo, a groso modo, de la velocidad bastará con regular las revoluciones por minuto del propulsor.

En el autopiloto se ha permitido variar este elemento en función del estado en el que se encuentre el avión, ya sea por ejemplo despegando o en vuelo de crucero.

7.9.5. Altitud - PID secundario

Este parámetro, brevemente mencionado en la sección 5.2, es medido de forma indirecta por el **sensor de presión**.

Debido a la condición en la que la atmósfera terrestre es menos densa a medida que subimos en altitud podremos aprovecharnos de esto como punto de referencia para **evaluar a qué altitud** nos encontramos.

Hemos de tener en cuenta de que esta medida de altitud puede verse alterada por cambios de clima o de localización geográfica. Por ello, es altamente conveniente cotejar esta información de altitud junto a la proveída por el GPS.

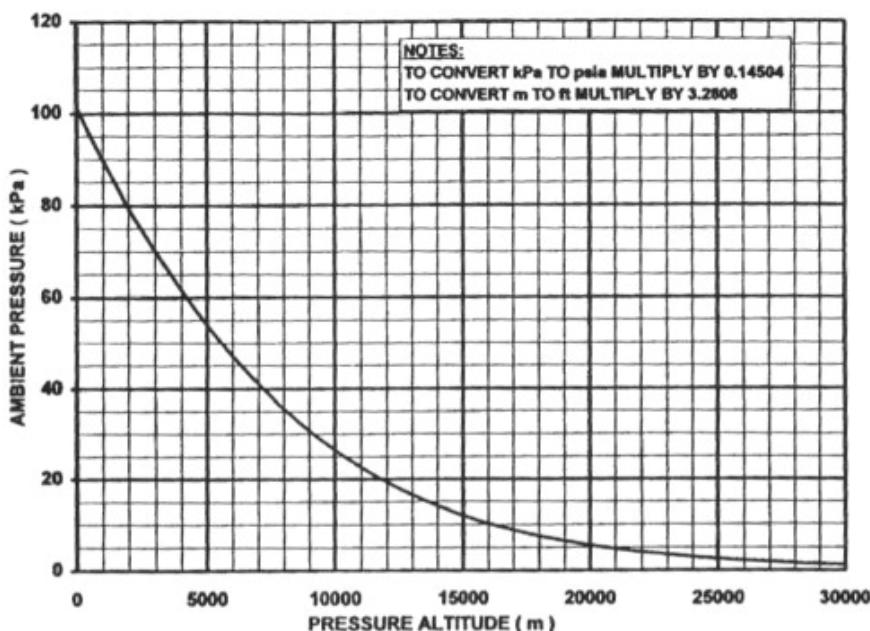


Figura 38: Relación entre presión atmosférica y altitud (Soares, 2015)

El valor de la presión decrece de forma exponencial con respecto a la altitud. Para hacer el cálculo de la altitud tendremos que utilizar la siguiente fórmula:

$$P(\text{mbar}) = 1013,25 \cdot (1 - 0,0000225577 \cdot A)^{5,2559}$$

La presión se expresará en mbar o hPa. Siendo P la presión, y A la altitud.

Esta ecuación nos proporciona la **relación entre la presión atmosférica y la altitud** según la Atmósfera Estándar Internacional o ISA (International Standard Atmosphere). Estos valores también sirven como referencia para otros cálculos. A nivel del mar se considera que existe una presión de 1035,25 mbar, según aumentamos en altitud, disminuirá la presión.

(«Herramientasingenieria web», 2022)

Este control es de tipo **PID secundario** debido a que el funcionamiento de este PID influye sobre otro, es decir, la salida de este será entregada a la entrada del siguiente.

El modelo PID de este elemento tendrá las siguientes características:

- **Entrada:** Altitud actual.
- **Objetivo:** Altitud final a conseguir.
- **Salida:** Ángulo que debe de tomar el avión para ascender o descender.

7.9.6. Rumbo de brújula - PID

Para poder seguir rutas será necesario tener la capacidad de girar utilizando el timón de cola para así poder **encaminarnos** correctamente hacia el **objetivo**. Estos giros no podrán ser demasiado bruscos, por lo que nos servirá de ayuda un control de tipo PID para lograr un giro suave y un afinamiento constante en el rumbo a seguir.

Periódicamente nos encontraremos evaluando el rumbo de brújula hacia el siguiente objetivo y pasando este valor de rumbo a este controlador.

El modelo PID de este elemento tendrá las siguientes características:

- **Entrada:** Rumbo de brújula actual.
- **Objetivo:** Ángulo de rumbo final.
- **Salida:** Posición del timón de cola.

7.10. Modos de vuelo

A continuación, detallaremos cuales serán los modos de vuelo en los que se podrá poner el piloto automático de forma que permita más o menos libertad al control manual.

Estos modos de vuelo forman parte del funcionamiento real de este piloto automático, y se aclaran para entender mejor el objetivo del proyecto, en cambio, **el único modo que se ha implementado es el modo automático**. Para poder evaluar en qué modo quiere el usuario que opere el avión se utilizará uno de los canales de la emisora para establecer cual de los modos es el deseado.

7.10.1. Control manual

Este modo de vuelo es el que tiene como único protagonista el usuario, en este caso, el **microcontrolador** no actuaría ni se encargaría de realizar **ninguna tarea**. Únicamente quedar a la espera de que el mando le de el control y transfiriendo los datos de control a los actuadores.

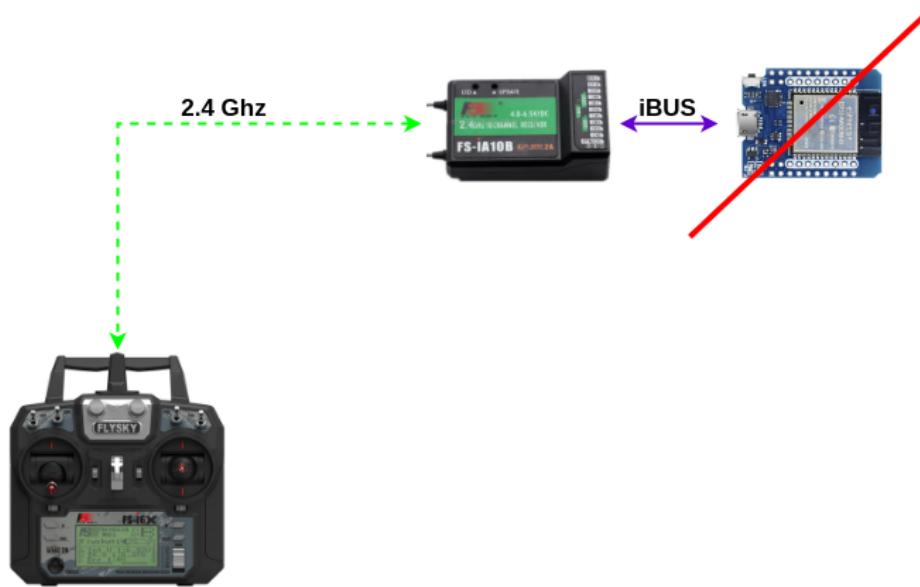


Figura 39: Esquema de conexión del modo de vuelo manual.

En el esquema superior podemos ver, de izquierda a derecha, la emisora, el receptor y el microcontrolador. La simbología que esto tiene es que el transmisor se conecta con el receptor por medio de radiofrecuencia y que el microcontrolador quedaría inactivo.

Esta forma de control equivaldría al tradicional en el cual no se dispone de ningún sistema automático.

7.10.2. Control semi-automático

La función principal de este modo es la de favorecer que aficionados que se encuentran empezando con el aeromodelismo tengan cierta seguridad de que no se les va a descontrolar, acabando con su incursión en este mundo de forma precoz.

En este modo podemos aprovecharnos de las capacidades de estabilización del modo automático para implementar asistencias al vuelo.

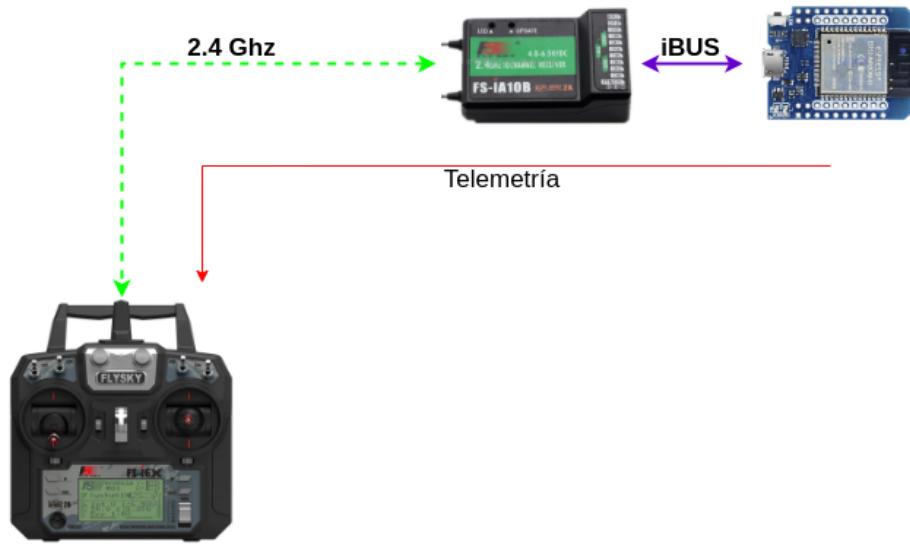


Figura 40: Esquema de conexión del modo de vuelo semi-automático.

En este modo, **todos los componentes del sistema se encuentran habilitados** el procesador se comunica con el receptor por un bus privado de la marca FlySky/Turnigy llamado iBUS, nos permitirá una conexión bidireccional tanto para leer lo que ha recibido desde la emisora como para enviar datos de telemetría de vuelta.

De nuevo, el que tiene el **control** sobre los actuadores es el microcontrolador, pero este funcionará transmitiendo en un principio lo mismo que recibirá por parte de la emisora.

La **asistencia al vuelo** entrará en juego cuando el usuario comience a **sobrepasar los límites** de lo que se considera seguro hacer durante el vuelo.

Para controlar este tipo de eventos se medirán las lecturas de los ángulos en todos los ejes, y, en caso de salirse de cierto intervalo de ángulo seguro, entonces se procederá a **tomar el control hasta hacerlo volver** de nuevo a ese ángulo seguro.(Figura 41)

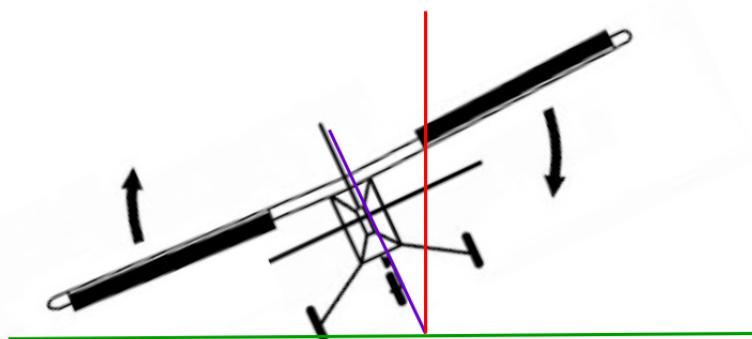


Figura 41: Esquema de ángulo inseguro y recuperación.

7.10.3. Control automático

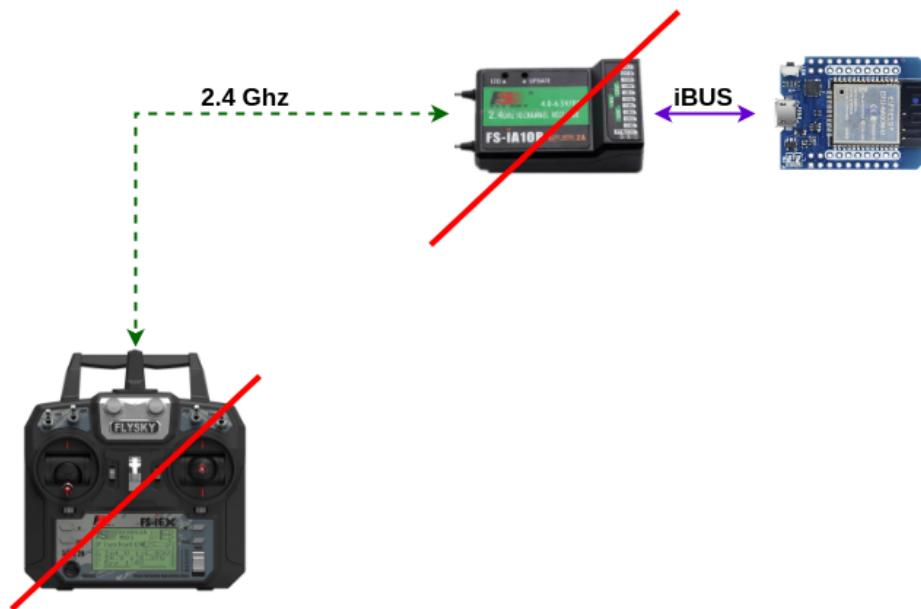


Figura 42: Esquema de conexión del modo de vuelo automático.

Al modo de control automático pasará el microcontrolador a partir del momento que se detecte que **no hay señal** por parte del receptor o si se ha establecido el control automático en el mando radiocontrol.

En este caso de automatización no contaremos con **ningún tipo de asistencia** ni entrada por parte de la emisora, tampoco se enviarán datos de telemetría, bien porque se supone que el avión se encuentra lejos del punto de **transmisión de tierra** o bien, porque así lo ha establecido el usuario.

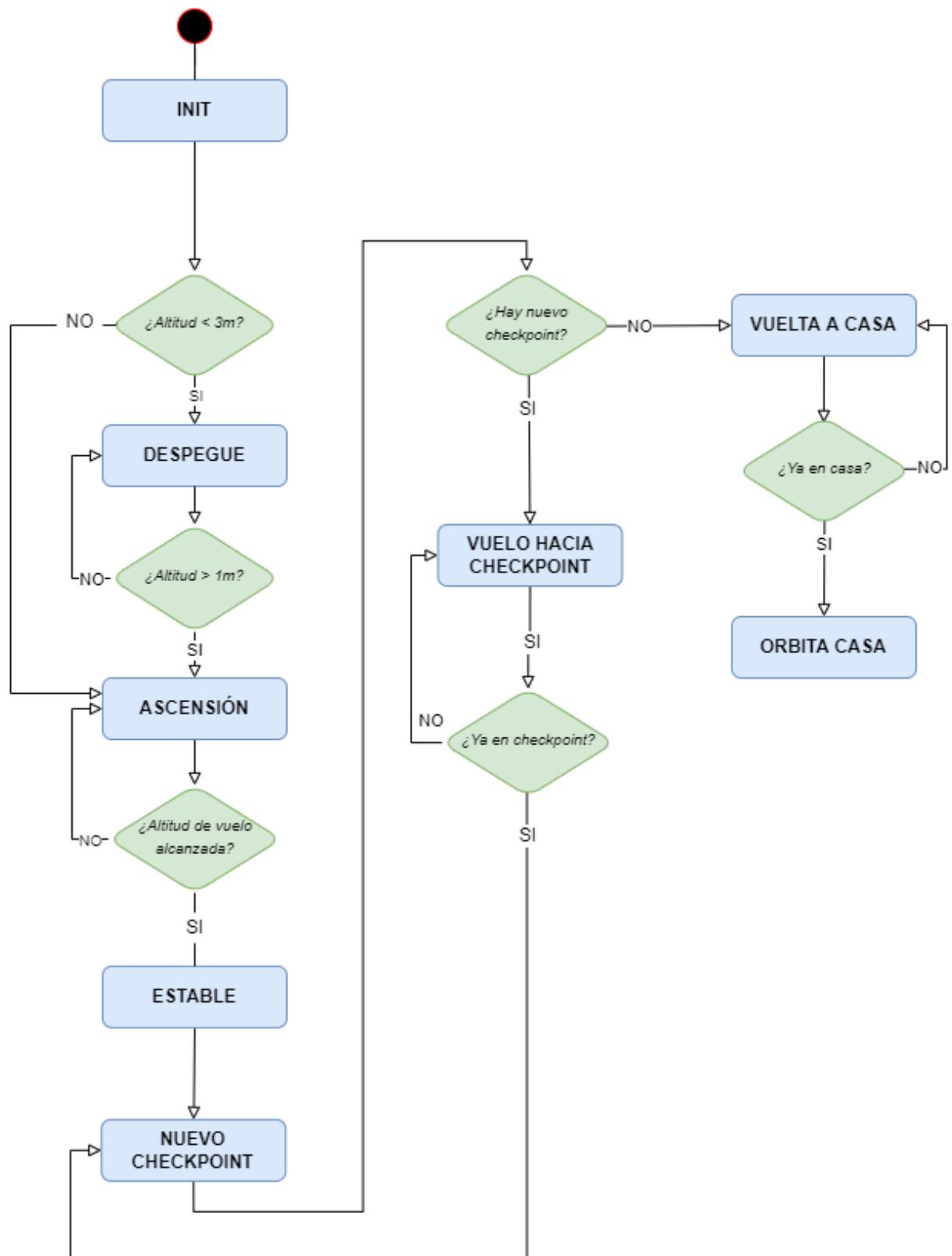
El microcontrolador tendrá que servirse **únicamente de los sensores y del GPS** para seguir la ruta que le ha sido designada, en caso de que haya terminado o no existan más puntos de ruta este volverá al punto de despegue a la **espera de una entrada manual** por parte del usuario para ser aterrizado.

Durante todo el vuelo se tendrá como objetivo tener **lo más estable posible** la aeronave tanto en el eje longitudinal como en el transversal y a un ritmo de velocidad constante mientras el timón de cola se encarga de hacer a la aeronave **seguir la ruta designada**.

7.10.4. Máquina de estados

Para poder apreciar mejor de qué forma va a ser controlada la aeronave dentro del modo automático se ha elaborado un diagrama de flujo:

MODO AUTOMÁTICO



Estados:

- **INIT:** se inicializa o se llaman a todas las funciones de inicialización del microcontrolador.
- **DESPEGUE:** este es el estado que se encarga de despegar a la aeronave del suelo hasta unos 3m de altura.
- **ASCENSIÓN:** una vez el avión ha despegado del suelo este estado realiza la función de elevar el avión hasta una altitud segura.
- **ESTABLE:** se encarga de mantener el avión a cierta altura y completamente estable.
- **NUEVO CHECKPOINT:** evalúa si tiene algún checkpoint todavía en la lista de puntos de ruta.
- **VUELO HACIA CHECKPOINT:** utiliza los actuadores para dirigir a la aeronave hacia el checkpoint indicado.
- **VUELTA A CASA:** realiza un control para llevar el avión hasta el punto de despegue.
- **ORBITA CASA:** dada una coordenada se encarga de sobrevolarla en círculos a cierta altitud.

7.11. Sistema operativo

Una de las partes más importantes de este proyecto y más críticas es la de utilizar un sistema operativo como FreeRTOS dentro del microcontrolador. La incorporación de este software nos proveerá de características muy convenientes sobretodo teniendo en cuenta el tipo de aplicación para el que lo estamos empleando.

7.11.1. Características de FreeRTOS

El sistema operativo a utilizar, como ya hemos mencionado, es FreeRTOS, pero no es el mismo que se ofrece de forma gratuita, sino una versión modificada por el fabricante espressif para poder aprovechar todas las características de su microcontrolador. Para poder comprender el por qué de la elección de esta alternativa de sistema operativo vamos a nombrar las características que son más relevantes para este proyecto:

- Aprovechamiento de los dos núcleos del ESP32 gracias al planificador de espressif.
- Mecanismos de control de bloqueo.
- Creación de tareas y gestión de recursos.
- Control de flujo.
- Abstracción sobre el hardware del ESP32
- Documentación y toolkit.

7.11.2. Programación

El tipo de programación en todo momento ha ido enfocado a objetos, nos basamos en ellos para crear funcionalidades asociadas a cada uno de ellos, e irlos interconectando para lograr la funcionalidad deseada. Muchos de ellos pueden ir creando tareas en función las vayan necesitando para actualizar o revisar funciones a su cargo.

Vamos a, en los siguientes apartados, ir explicando como se han ido implementando cada una de las funcionalidades requeridas.

El idioma por defecto que hemos utilizado para programar es el inglés, debido a sus posibilidades para luego compartir el código.

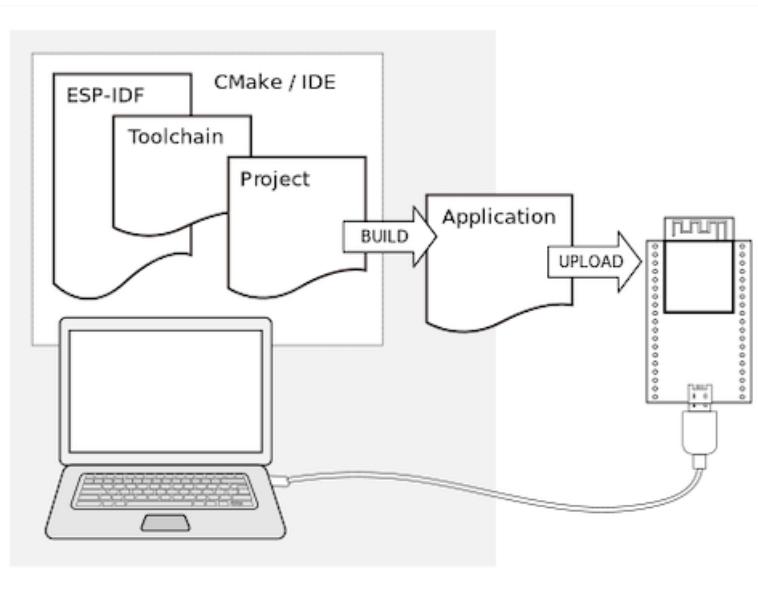


Figura 43: *Entorno de programación ESP-IDF («ESP-IDF documentation», 2022)*

7.11.3. Clase aircraft

Esta clase aircraft primero define los estados posibles de control establecidos en el apartado 7.10.4, que pueden ser MANUAL, SEMI-AUTOMATIC o AUTOMATIC:

```
// Control modes
enum ControlMode_t
{
    MANUAL = 0,
    SEMI_AUTOMATIC = 1,
    AUTOMATIC = 2
};
```

Dentro de la propia clase se encontrará la creación de un objeto para el piloto automático y **contendrá las variables referentes a la aeronave, tales como la posición de los actuadores o datos de sus sensores**. También hará uso del enum anteriormente mencionado para ir almacenando en qué estado de control se encuentra el avión.

```

class Aircraft
{
private:
    ControlMode_t controlMode;

public:

    Autopilot autopilot;

    // Actuators
    float elevator, aileron, rudder, throttle;

    // Altitude in ft
    float altitude;

    // Grouping array
    float *actuators[4] = {&elevator, &aileron, &rudder, &throttle};

    Aircraft();
    ~Aircraft();

    /**
     * @brief Set the control mode
     * @param newControlMode New type of control
     */
    void setControlMode(ControlMode_t newControlMode);

    // Print actuators separated by commas.
    void printActuators();

    /**
     * @brief Set throttle revolutions.
     * Disable speed control from autopilot.
     *
     * @param pThrottle Throttle, from 0 to 1.
     */
    void setThrottle(float pThrottle);

    //Return the altitude
    float getAltitude();
};

```

7.11.4. Clase PID

Esta clase es la que define el modelo PID que vamos a utilizar y todas sus funcionalidades. Para crear esta clase se partió de un modelo básico el cual se fué mejorando según las necesidades que iba requiriendo el control automático. Posee bastantes funcionalidades no tan comunes en un PID tales como la posibilidad de hacerlo secundario o que utilice una

salida de tipo circular.

Aquí tenemos el constructor del objeto, en el que podemos ver una descripción de todos los parámetros:

```
lic:  
/* Constructor for primary PID, which controls an actuator variable.*/ You, hace 3 meses • Simulator working and GPS library  
@param Kp Proportional gain  
@param Ki Integral gain  
@param Kd Derivative gain  
@param max Maximum value of manipulated variable  
@param min Minimum value of manipulated variable  
@param outputVar Variable to store the resulting value.  
@param circularInput If the input variable is circular. (The ends are connected)  
@param maxC Maximum value for input.  
@param minC Minimum value for input.  
*/  
PID(float max, float min, float Kp, float Ki, float Kd, float* outputVar, bool circularInput = false, float maxC=0., float minC=0.);
```

Este objeto tiene también una función llamada calculate() la cual hemos de ir ejecutando de forma periódica cada vez que le vayamos a introducir un nuevo valor para que realice el cálculo con este último.

```
// Receive new data and processses it to a new value trying to achieve the setpoint.  
void calculate(float pv);
```

7.11.5. Clase GPS

El objeto encargado de alojar todo lo referente a coordenadas GPS, maneja desde el tipo de dato para las coordenadas hasta la ruta que ha de seguir el avión.

En el header de esta clase tenemos la definición del objeto que utilizaremos para almacenar las coordenadas de forma estructurada.

```

struct GPS_coordinate
{
    double latitude, longitude, altitude;

    // @brief Construct a new empty gps coordinate.
    GPS_coordinate();

    /**
     * @brief Construct a new gps coordinate.
     *
     * @param latitude Coordinate latitude.
     * @param longitude Coordinate longitude.
     * @param altitude Altitude. If left empty then altitude will not be changed.
     */
    GPS_coordinate(double latitude, double longitude, double altitude = 0.);

    //Convert units into radians
    GPS_coordinate toRadians();

    //Converts units into degrees
    GPS_coordinate toDegrees();

    //Check if this coordinate has not been initialized
    operator bool() const {return latitude != 0. || longitude != 0. || altitude != 0.;}
};

```

Apreciamos como esta clase almacena las coordenadas de latitud, longitud y altitud. También incorpora funciones para transformar los ángulos tanto a grados como a radianes.

De entre las funciones más destacadas de la clase GPS tenemos las correspondientes al control de rutas:

```

//##### PATH CONTROL #####
// Add new coordinate to the end of path
void appendCoordinate(GPS_coordinate coord);

// Append a new path to the old one
void appendPath(vector<GPS_coordinate> coords);

// Delete the old path and start a new one
void newPath(vector<GPS_coordinate> coords);

size_t getPathSize();

//@return Return the next coordinate and erase it
GPS_coordinate popCoord();

GPS_coordinate getCoordAt(int i);

```

Estas se encargan de gestionar un vector de coordenadas, que es el camino a seguir, e incorpora funciones de utilidad.

Como funciones muy importantes tenemos las mencionadas en el apartado 7.8, las cuales realizan los cálculos terrestres para calcular rumbo y distancia al objetivo:

```
//##### DIRECTION CONTROL #####
void setCurrentPosition(GPS_coordinate coord);
GPS_coordinate getCurrentPosition();
void setHome(GPS_coordinate coord){home = coord;};
GPS_coordinate getHome(){return home;};
void setCompass(float compass){currentCompass = compass;};
float getCompass(){return currentCompass;};

/**
 * @brief Compass angle from current position towards desired coordinate.
 *
 * @param coord Destination point.
 * @return Compass angle in clockwise sexagesimal degrees.
 */
float compassToCoord(GPS_coordinate coord);

/**
 * @brief Distance between current position and the desired coordinate.
 * The calculation could have a distance error of 0.3%, especially at the polar extremes and for long distances through various parallels.
 *
 * @param coord Destination point.
 * @return Distance in Km.
 */
double distanceToCoord(GPS_coordinate coord);

/**
 * @brief Compass angle from A towards B coordinates.
 *
 * @param coord_A Point A.
 * @param coord_B Point B.
 * @return Compass angle in clockwise sexagesimal degrees.
 */
float compassBetweenCoord(GPS_coordinate coord_A, GPS_coordinate coord_B);
```

7.11.6. Clase autopilot

Este es el corazón del piloto automático, donde realmente se sigue el diagrama de estados del apartado 7.10.4, se evalúan y se ejecutan las órdenes de control.

Como el resto, este también se encuentra definido dentro de su propia clase y contiene primeramente una definición de todos los estados posibles de la máquina de estados:

```
// Autopilot state machine
enum ApStateMachine_t
{
    INIT = -1,
    PAUSE = 0,
    TAKEOFF = 1,
    ASCENSION = 2,
    STABLE = 3,
    NEW_CHECKPOINT = 4,
    TO_CHECKPOINT = 5,
    TO_HOME = 6,
    ORBIT_HOME = 7
};
```

Dentro de la propia clase de autopilot nos encontramos primero con la llamada a distintos elementos que serán necesarios para la ejecución:

```

-----  

TaskHandle_t updateHandler;  

Aircraft *aircraft;  

bool enabled;  

ApStateMachine_t currentState;  

// Orientation  

PID pitch, yaw, roll;  

PID speed;  

// Grouping array  

PID *pidControllers[4] = {&pitch, &yaw, &roll, &speed};  

//Secondary PID for pitch  

PID altitudeControl;

```

Observamos que necesita una referencia a la aeronave a controlar, crea una variable para almacenar el estado actual y por último todos los PID mencionados en el apartado 7.9, para controlar el avión.

El resto de funciones que este incorpora son de utilidad y para habilitar y deshabilitar ciertos elementos del piloto.

De aquí, la parte más importante a destacar es la tarea de actualización de estado, esta debe, a una cierta frecuencia, ir comprobando los valores de los sensores almacenados en "aircraft" e ir evaluando el nuevo estado.

Esta tarea es la llamada taskUpdateAutopilot(), y es creada dentro del constructor de la clase autopilot.

Aquí tenemos una captura de ejemplo de esta función:

```

void taskUpdateAutopilot(update_args *pArgs)
{
    Autopilot *_autopilot = pArgs->autopilot_arg;
    Aircraft *_aircraft = pArgs->aircraft_arg;
    float distance;
    int orbitCounter = 0;
    while (true)
    {
        switch (_autopilot->getCurrentState())
        {
        case INIT:
            _aircraft->elevator = 0;
            _autopilot->disableAltitudeControl();
            _autopilot->disablePitch();
            vTaskDelay(pdMS_TO_TICKS(23000));

            if (_aircraft->getAltitude() < 3.)
                _autopilot->setCurrentState(TAKEOFF);
            else
                _autopilot->setCurrentState(ASCENSION);
            break;

        case TAKEOFF:
            if (_aircraft->getAltitude() > 1.)
            {
                _autopilot->setCurrentState(ASCENSION);
            }

            if (!_autopilot->gps.getHome())
            {
                _autopilot->gps.setHome(_autopilot->gps.getCurrentPos:
ESP_LOGI(__func__, "HOME: %f, %f\n", _autopilot->gps.(
}
        }
    }
    _autopilot->clearPids();
}

```

7.11.7. Componente cmdConsole

Este es el componente que genera la consola de comandos en el puerto serie para poder comunicarnos con el pc, simplemente es una tarea que se ejecuta constantemente y trata de identificar que comando es el introducido.

En la carpeta llamada cmd tenemos la implementación de todos los comandos de la consola. Aquí un ejemplo del reconocimiento de cadenas:

```

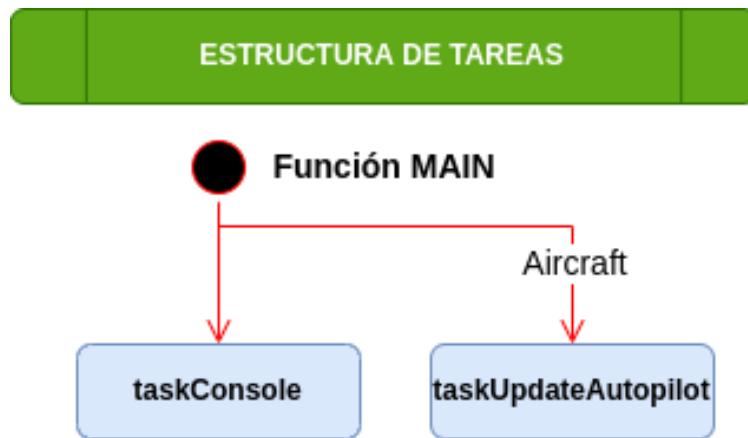
/* Add the command to the history if not empty*/
if (strlen(line) > 0)
{
    linenoiseHistoryAdd(line);

    /* Try to run the command */
    int ret;
    esp_err_t err = esp_console_run(line, &ret);
    if (err == ESP_ERR_NOT_FOUND)
    {
        printf("Unrecognized command\n");
    }
    else if (err == ESP_ERR_INVALID_ARG)
    {
        // command was empty
    }
    else if (err == ESP_OK && ret != ESP_OK)
    {
        printf("Command returned non-zero error code: 0x%x (%s)\n", ret, esp_err_to_name(ret));
    }
    else if (err != ESP_OK)
    {
        printf("Internal error: %s\n", esp_err_to_name(err));
    }
    /* linenoise allocates line buffer on the heap, so need to free it */
    linenoiseFree(line);
}

```

7.11.8. Diagrama de tareas

En el estado actual del proyecto el diagrama de tareas es el siguiente:



El SO crea una tarea para la consola y otra para el piloto automático, cada vez que se recibe un comando con nueva información, la consola actualiza los datos de sensores del piloto automático, este procesa y devuelve la información a la consola para que sea enviada.

8. Banco de pruebas

8.1. Objetivo

La finalidad de realizar un banco de pruebas es la de poder mostrar de forma **objetiva** el rango de **precisión** y **eficiencia** que puede alcanzar este piloto automático mediante la **medición de una serie de parámetros**.

Para ello se ha evaluado un parámetro a optimizar y además, como **prueba de resiliencia** se le ha aplicado una función de **ruido** a uno o varios **sensores** que pudieran estar involucrados en el resultado para así poder observar el impacto que esto tiene sobre el resultado ideal.

El indicador principal que hemos utilizado para medir la eficiencia de la solución es la **precisión al pasar un punto de ruta**, esto nos da indicación de que la aeronave no ha llegado hasta al punto de **estrellarse** y que además es capaz de mantener cierta **estabilidad al desvío**.

8.2. Prueba 1: Precisión de ruta

En este caso de prueba vamos a examinar, en condiciones **con** y **sin interferencias**, la **precisión** que obtiene la aeronave pasando sobre un punto determinado de una **ruta** previamente establecida.

8.2.1. Escenario

Se establecerá una **ruta de 4 puntos**, examinaremos la precisión con la que es capaz de atravesar el segundo punto.

- **Objetivo:** Segundo punto de ruta.
- **Objeto de medición:** Distancia mínima entre el avión y el segundo punto indicado dentro de la ruta.
- **A optimizar:** Mejor cuanta menor distancia obtenida.

8.2.2. Ejecución y resultados

Se aplicará la interferencia o **ruido sobre la señal de la brújula**, simulando un fallo o turbulencias en el sensor magnético.

- El primer caso, con $0,1^\circ$ de desviación, podría corresponderse a ruido introducido por las vibraciones de la aeronave.
- El segundo caso, con 2° de desviación podría achacarse a un malfuncionamiento leve de la brújula.

Tipo de ruido	Distancia mínima
Sin ruido	10'16m
Ruido blanco ($\pm 0,1^\circ$)	10'23m
Ruido blanco ($\pm 2^\circ$)	12'47m

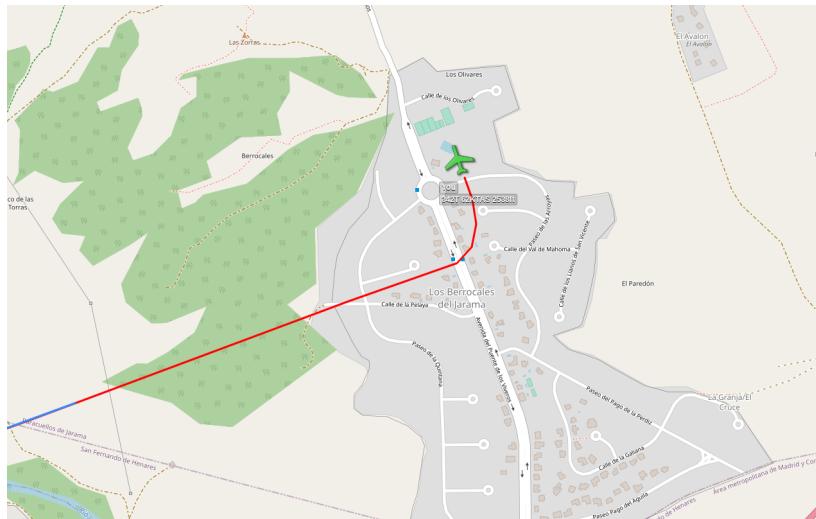


Figura 44: *Trazada real de la aeronave sobre un punto de ruta.*

8.2.3. Análisis de resultados

De los resultados de esta prueba, en la que simulamos turbulencias en los valores proporcionados por la brújula, podemos extraer que el sistema tiene una **precisión aproximada de 10m** en su cercanía al punto indicado.

Cuando se presentan **turbulencias leves** el resultado queda prácticamente **inalterado**. Ante la presencia de un ruido de **mayor magnitud**, el valor ha sido desviado en muy **poca medida**.

Se considera por tanto que la aeronave es capaz de atravesar el punto con una precisión notable incluso con ruido moderado.

8.3. Prueba 2: Ruido giroscópico

En el terreno real existen ciertas interferencias que no son tenidas en cuenta dentro del simulador, como por ejemplo las **vibraciones provocadas por el motor** que pueden alterar las mediciones del giroscopio, **perdiendo así precisión** en la medida.

En esta prueba vamos a intentar simular turbulencias de distintas magnitudes para comprobar si la aeronave puede permanecer en vuelo de forma estable mientras se alteran los valores proveídos por el giroscopio.

8.3.1. Escenario

De forma similar a la prueba anterior se establecerá una **ruta de 4 puntos** y examinaremos la precisión con la que es capaz de atravesar el segundo punto siendo sometido a turbulencias en el giroscopio.

- **Objetivo:** Segundo punto de ruta.
- **Objeto de medición:** Distancia mínima entre el avión y el segundo punto indicado dentro de la ruta.
- **A optimizar:** Mejor cuanta menor distancia obtenida.

8.3.2. Ejecución y resultados

Se aplicará la interferencia o **ruido sobre la señal del giroscopio en varios ejes**.

- El primer caso, con $0,1^\circ$ de desviación, podría corresponderse a ruido introducido por las vibraciones de la aeronave.
- El segundo caso, con 2° de desviación podría achacarse a un malfuncionamiento leve del giroscopio.
- El tercer caso, con 10° de desviación podría deberse a un malfuncionamiento grave del giroscopio.

Tipo de ruido	Distancia mínima
Sin ruido	10'16m
Ruido blanco ($\pm 0.1^\circ$)	9'86m
Ruido blanco ($\pm 2^\circ$)	10'96m
Ruido blanco ($\pm 10^\circ$)	11'60m

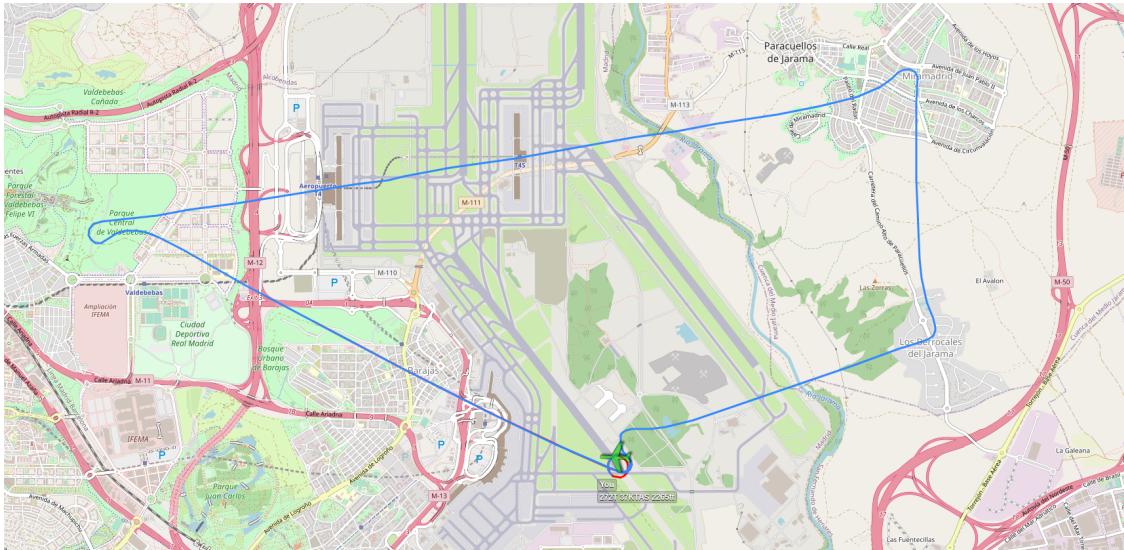


Figura 45: Trazada real de la aeronave sobre la ruta completa.

8.3.3. Análisis de resultados

El control de la aeronave se ve afectado en poca cantidad por el ruido en el sensor giroscópico. Este poco desvío se debe a que el controlador PID una vez ha obtenido suficientes muestras es capaz de obtener el "valor medio" y consigue minimizar el desvío de la ruta.

9. Compilación y ejecución de la demostración

9.0.1. Compilación

Para compilar este proyecto serán necesarios los siguientes pasos:

- Descargar e instalar el SDK llamado ESP-IDF.
- Ejecutar el script para definición de variables de entorno necesarias para la compilación dentro de la carpeta de instalación: /esp-idf/export.sh
- Dirigirse al directorio del proyecto y lanzar el comando de compilación "idf.py build".

* En caso de querer configurar el proyecto lanzar "idf.py menuconfig".

9.0.2. Ejecución

Los pasos a seguir para poder ejecutar el simulador tal y como se ha hecho en la demo:

- Dirigirse a la carpeta Test/flightgear
- Conectar la placa
- Ejecutar el script que hemos creado llamado runSimulation.sh. (Este script solamente funcionará en linux con flightgear y la app Konsole instalada)

10. Conclusiones

Finalmente, se procederá a hacer una **recapitulación** de las partes o conceptos más relevantes en esta sección de conclusiones, de manera que se recoja de forma clara y concisa cuales han sido los resultados que hemos obtenido después del tiempo invertido, además de las sensaciones finales con respecto a como se comenzó.

Una de las primeras inquietudes que surgió fue la **percepción de escala** de este proyecto. Al ser un proyecto que, normalmente, precisaría de un equipo desarrollador completo, resultó de gran importancia definir un alcance preciso si no se deseaba un resultado incompleto. Finalmente, resultó en un correcto dimensionado con un control totalmente automático de la aeronave en simulador.

Otro de los desafíos que nos pudimos encontrar fue el de decidir si realizarlo de **forma física** con un avión radiocontrol o por el contrario proceder con un **proyecto en simulador**, en este caso, y en relación al punto anterior, por controlar la escala del proyecto y no incurrir en gastos de constantes reparaciones nos decantamos por la solución simulada. Consideramos que esto fue acertado debido a que, teniendo en cuenta la cantidad de veces que se ha tenido que probar y simular el modelo, se habría hecho **imposible** llegar a término en el tiempo estimado.

En relación a la **calidad** que se esperaba de este proyecto, hemos de confesar que no se estimaba inicialmente un nivel de **desempeño** tan competente como finalmente hemos obtenido a la hora de superar las pruebas, tanto de **seguimiento de rutas** como **resiliencia al ruido**. La estabilidad obtenida y la posibilidad de hacer esta **solución real** ha sido el resultado del trabajo y la ejecución de **pruebas** tanto manuales como de tests sintéticos para asegurar su comportamiento en múltiples condiciones. El **sistema desarrollado** nos deja un producto que encajaría muy bien en relación prestaciones/precio dentro del mercado actual.

Al comienzo, el **cambio** de rumbo hacia un **paradigma virtual** nos supuso incertidumbre, sobretodo debido al desconocimiento acerca de **interfaces y simuladores compatibles**. Esto no nos hizo cesar en nuestro empeño, y, después de mucho tiempo invertido navegando entre todas las posibilidades se consiguió sacar adelante dentro de uno de los

simuladores más precisos del mercado, como es **FlightGear**.

Con todo esto, los **resultados** que se han obtenido nos han parecido más que **satisfactorios** dadas las dimensiones, complejidad y el reto que planteaba inicialmente este trabajo de fin de máster.

Procederemos a continuación con una serie de **preguntas** que **hipotéticamente** pudieran haber surgido y les daremos respuesta, esto con la finalidad de **mejorar el grado de comprensión** del proyecto.

10.1. Cuestiones

10.1.1. ¿Se ha cumplido el objetivo de este proyecto?

Principalmente, se ha cubierto el objetivo principal con poder conseguir un **autopiloto contundente y resistente** a los vientos en un simulador preciso sin problemas. Debido al **presupuesto de tiempo** y a la extensión del proyecto, se ha hecho necesario concentrar el esfuerzo de desarrollo en conseguir que el sistema de autopilotaje funcione de forma autónoma en un sistema operativo como FreeRTOS.} Por ello, consideramos que este código, con la adecuada implementación e integración de drivers para el hardware, debido a las pruebas de software, podría estar **funcionando con bastante seguridad en un sistema real**.

10.1.2. ¿Cómo de resistente y fiable es el modo automático?

Durante el desarrollo del modelo de piloto automático hemos estado probándolo continuamente tratando de **empujarlo a situaciones inciertas** como por ejemplo en condiciones de viento relativamente altas o intentando, pero sin conseguirlo, que se estrelle jugando con los controladores de la simulación. Incluso volando hacia abajo llega a una posición estable en poco tiempo.

Respecto a la **resistencia al viento**, el control PID hace un trabajo bastante bueno contrarrestando las fuerzas aplicadas al fuselaje y manteniendo el avión estable.} El **sistema operativo** actualmente no tiene problemas con las **restricciones de tiempo real** y conseguir una respuesta de baja latencia, el sistema en este momento tiene dos tareas de computación para dos núcleos y sólo se están evaluando tres veces por segundo. Estimamos que los procesos relativos a las lecturas de los sensores y la transferencia de las señales podrán caber en este sistema en un futuro.

10.1.3. ¿Qué planes futuros esperamos de este proyecto?

Al principio, nuestra intención era hacer este proyecto como un hardware completo para incrustarlo en un avión RC. Algunas limitaciones como el **presupuesto de tiempo** y la gran posibilidad de **accidentar el avión** y no poder hacer más pruebas fueron decisivas para cambiar a un enfoque HITL, todavía usando el hardware pero usando un entorno simulado.

El siguiente paso sería utilizar este modelo desarrollado y crear algunos controladores para los sensores y el hardware para controlar el avión, y finalmente implementarlo en un avión. Muy probablemente, a medida que esto sea probado que funcione en la práctica y dé los resultados esperados procederemos a publicar el proyecto en la plataforma GitHub, permitiendo que más personas en este hobby puedan tener su propio piloto automático.

10.2. Questions

10.2.1. Have the objective of this project been achieved?

Mostly, the main objective has been covered with being able to get a **fully working and resilient** to winds **autopilot** in an accurate simulator without problems. Due to the **time budget** and the extension of the project, it made necessary to concentrate the development effort into getting the autopilot system running autonomously on an operating system like FreeRTOS.

For that reason, we consider that this code, with the appropriate implementation and integration of drivers for the hardware, due to the software testing, could be **running pretty confidently on a real system**.

10.2.2. How resilient and reliable is the automatic mode?

During the development of the autopilot model, we have been testing it continuously trying to **push it into uncertain situations** like in a relatively high wind speeds conditions or trying but not succeeding to make it crash messing with the simulation controllers. Even flying downwards it gets to a stable position in a short time.

Regarding the wind resilience, the PID control does a pretty good job counteracting the applied forces to the fuselage and keeping the airplane stable.

The **operating system** currently does not have problems with **real time restrictions** and getting a low latency response, the system at this moment have two computing tasks for two cores and only are being evaluated three times a second. We estimate that the processes regarding the sensor readings and transferring the signals will be able to fit in this system in a future.

10.2.3. Which future plans we expect from this project?

At first, our intention was to make this project as a complete hardware for embedding it into an RC plane. Some limitations as the **time budget** and the great possibility to **crash the plane** and not being able to test more were decisive to change to a HITL approach, still using hardware but using a simulated environment.

The next step would be to use this developed model and create some drivers for the sensors and hardware for controlling the airplane, then finally implement it into an aircraft. Very likely, as this is tested by us that works in practice and give the expected results we will

proceed to publish the project in the GitHub platform, allowing more people in this hobby to have their own autopilot.

10.3. Ampliaciones y consideraciones

En cuanto a las posibilidades de este proyecto podríamos destacar las siguientes cuestiones:

- Este proyecto se hizo teniendo en mente **abrir el código** de forma que toda la comunidad de aeromodelismo se pudiera beneficiar de este trabajo que es tan necesario para lograr abaratar los **altos costes que imponen** los fabricantes, logrando una opción totalmente competente y de similares características.
- Aunque era la idea inicial, en nuestro caso no se ha realizado la programación de **drivers** ya que se requería todo el tiempo del que se dispusiera para lograr hacer esta implementación del piloto automático **dentro del simulador**. De esta forma **evitamos tener que reparar** y gastar dinero en modelos reales en lo que se realizan las pruebas.
- Es muy posible abrir la veda a posibles **variantes de aviones** u otro tipo de vehículos aéreos contemplando sus particularidades y haciendo la programación lo mas modular posible.
- Se han tenido en cuenta para este proyecto todo el hardware de **forma realista** para poder implementarlo en la realidad sin inconvenientes y en base a algo realizable.

Referencias

- Blog aerodeporte.* (2022). Consultado el 11 de junio de 2022, desde <https://aerodeporte.blogspot.com/2016/10/ejes-del-avion-y-superficies-de-control.html> (accessed: 11/06/2022)
- ESP-IDF documentation.* (2022). Consultado el 16 de junio de 2022, desde <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html> (accessed: 16/06/2022)
- Geotab ¿Qué significa GPS?* (2022). Consultado el 12 de junio de 2022, desde <https://www.geotab.com/es/blog/que-es-gps/#:~:text=El%5C%20GPS%5C%20funciona%5C%20a%5C%20trav%C3%A9s,para%5C%20medir%5C%20%C3%A1ngulos%5C%2C%5C%20no%5C%20distancias>. (accessed: 12/06/2022)
- GPS geoplaner web.* (2022). Consultado el 12 de junio de 2022, desde <https://www.geoplaner.com/> (accessed: 12/06/2022)
- Herramientasingenieria web.* (2022). Consultado el 14 de junio de 2022, desde <https://www.herramientasingenieria.com/onlinecalc/spa/altitud/altitud.html> (accessed: 14/06/2022)
- Hobbyking Akbird2.0 distributor website.* (2022). Consultado el 28 de marzo de 2022, desde https://hobbyking.com/es_es/arkbird-2-0-autopilot-system.html?__store=es_es (accessed: 28/03/2022)
- Mission Planner software.* (2022). Consultado el 12 de junio de 2022, desde <https://ardupilot.org/planner/> (accessed: 12/06/2022)
- Mouser Pixhawk4 distributor website.* (2022). Consultado el 28 de marzo de 2022, desde <https://www.mouser.es/ProductDetail/Seeed-Studio/102090024?qs=w%5C%2Fv1CP2dgqpXc3aavGDx9A%5C%3D%5C%3D> (accessed: 28/03/2022)
- Movable type scripts web.* (2022). Consultado el 12 de junio de 2022, desde <https://www.movable-type.co.uk/scripts/latlong.html> (accessed: 12/06/2022)
- Prisacariu, V. y col. (2017). The history and the evolution of UAVs from the beginning till the 70s. *Journal of Defense Resources Management (JoDRM)*, 8(1), 181-189. <https://doi.org/10.32560/rk.2019.1.13>
- Robotshop Pixhawk distributor website.* (2022). Consultado el 23 de marzo de 2022, desde <https://www.robotshop.com/es/es/piloto-automatico-avanzado-radiolink-pixhawk.html> (accessed: 23/03/2022)
- Soares, C. (2015). Chapter 19 - Basic Design Theory. En C. Soares (Ed.), *Gas Turbines (Second Edition)* (Second Edition, pp. 913-958). Butterworth-Heinemann. <https://doi.org/10.1016/B978-0-12-410461-7.00019-5>
- Sunearthtools web.* (2022). Consultado el 12 de junio de 2022, desde <https://www.sunearthtools.com/en/tools/distance.php#top> (accessed: 12/06/2022)
- Web geografía infinita.* (2022). Consultado el 12 de junio de 2022, desde <https://www.geografiainfinity.com/2018/04/loxodromia-y-ortodromia/> (accessed: 12/06/2022)