Trabajo fin de grado

Grado en ingeniería de tecnologías de telecomunicación

Escuela Técnica Superior de Ingenieros de Telecomunicación

Universidad de Valladolid

Sistema de guiado GNSS para aeronaves en la maniobra de rodaje

Autor: Alejandro Carrasco Rojo

Tutor: Ramón de la Rosa Steinz

Valladolid 9 de julio 2020

Miembros de la comisión evaluadora

Presidente: Javier M. Aguiar Pérez;

Secretario: Ramón de la Rosa Steinz;

Vocal: Juan Carlos Aguado Manzano.

Calificación

10,0 con consideración para matrícula de honor

Dedicado a todos los que en algún momento han aguantado una charla sobre este trabajo, gracias por vuestra paciencia.

También, a Pedro Ruíz de la Loma por cuidar de este proyecto desde que era simplemente una idea y a Ramón de la Rosa Steinz por ayudarme a hacerlo crecer.

Por último, a mi familia, y en especial, a mi padre

Resumen

En este trabajo se desarrolla y pone a prueba un sistema de guiado destinado a aeropuertos. El sistema se apoyará en una red inalámbrica específica para establecer la comunicación entre las aeronaves y la torre de control. El objetivo es analizar la viabilidad de realizar de forma autónoma la maniobra de rodaje en pista. Para ello se construirá un prototipo a escala del sistema de guiado. La red de comunicaciones apoyará el guiado de las aeronaves transmitiendo posiciones de referencia en las pistas de rodadura y servirá para poder enviar y transmitir la posición GNSS del avión. En el prototipo, esta posición será contrastada con una ubicación de referencia definida por la torre de control y mediante algoritmos de posicionamiento GNSS diferencial, la corrección será enviada de vuelta a la aeronave. Además, las aeronaves estarán en contacto pudiendo establecer prioridades entre sí en los puntos de intersección. Este sistema ayudaría a reducir en gran medida el volumen de trabajo y estrés para los pilotos en las fases de rodaje en aeropuertos, ya que actualmente es un proceso manual en el que el único apoyo con el que cuentan son las comunicaciones por radio con la torre de control y un mapa del aeropuerto.

Palabras clave: GNSS, aeronave, torre de control, rodaje, algoritmo, GBAS, movilidad autónoma, prototipo, Python

Abstract

In this work a guidance system for airports is developed and tested. The system will use a specific wireless network in order to establish communication between aircrafts and the control tower. The objective is to determine the viability of performing autonomously the taxiing manoeuvre at the airport. For that reason, a scaled model prototype will be built and tested. The communication network will support the aircraft guidance by transmitting reference positions at the taxiways and by sending and receiving the aircraft GNSS position. In the prototype, this position will be compared with a reference position defined by the control tower and through differential GNSS positioning algorithms, a correction in the aircraft's position will be send back. Moreover, aircrafts will be in touch amount them being able to establish priorities in taxiways junctions. This system will help to reduce the amount of work and stress which pilots have during the taxiing manoeuvre, since nowadays it is a manual process in which the only support they have are the constant communications with the control tower and a map of the airport.

Key words: GNSS, aircraft, control tower, taxiing, algorithm, GBAS, autonomous mobility, prototype, Python

Listado de figuras

Listado de tablas

Índice de contenidos

1. Introducción

1.1. Motivación

El objeto de estudio de este Trabajo Fin de Grado es la automatización de operaciones en el ámbito aeronáutico. Más concretamente, el proceso de rodaje de aeronaves en los aeropuertos. El proceso de rodaje en un aeropuerto, es el trayecto que realiza un avión desde que sale de una de las puertas de embarque hasta que enfila la pista de despegue, y al revés, desde que aterriza hasta que estaciona en su puerta correspondiente.

La aeronáutica fue uno de los primeros lugares donde se empezaron a aplicar técnicas de control autónomo de los vehículos (en este caso, los aviones). Dándole al ordenador de abordo la capacidad de gobernar la aeronave en base a una ruta o una altitud predefinida por la tripulación, permitiéndoles así, tener un vuelo más tranquilo y relajado, para que cuando llegase el momento de aterrizar, uno de los más críticos de todo el vuelo, estuviesen lo suficientemente descansados como para poder reaccionar con rapidez y seguridad si algo imprevisto sucedía.

Sin embargo, aún hoy en día, el proceso de rodaje en los aeropuertos, por ser un entorno muy complicado, con un gran número de obstáculos, la cercanía a otros aviones, operadores, maquinaria, etc... Sigue siendo una operación que los pilotos realizan de forma completamente manual.

Para un piloto, los 15 minutos previos al despegue, en los que el pasajero solo está deseando encontrarse, por fin, en el aire después de un sinfín de colas, controles y esperas, son frenéticos. Debe de estar atento a las indicaciones por radio desde torre de control sobre la ruta que debe seguir por las pistas del aeropuerto, hacer las comprobaciones pre-vuelo, estar atento a las indicaciones finales del personal de tierra, orientarse por un aeropuerto que normalmente no conoce, y por supuesto, llevar el avión hasta la cabecera de la pista desde la que vaya a despegar.

Esto supone un estrés y una prueba para sus capacidades multitarea, según se pudo comprobar a través de una entrevista con un Primer Oficial de A330 de la compañía Iberia. Citando sus palabras "Si existiese algún tipo de ayuda para esta maniobra, estaríamos mucho más tranquilos y habría menos sustos. Es como

cuando se conduce, no siempre todo el mundo respeta las normas de circulación. Hay veces que sin querer la gente se salta un stop o un ceda el paso. En este caso, pasa igual".

La propuesta del proyecto es por tanto, incorporar una ayuda para los pilotos en esta primera y última fase del vuelo, con la que puedan, si no delegar todas las tareas en el ordenador de abordo, por lo menos, sí muchas de ellas y puedan estar más tranquilos durante toda la maniobra.

Como un primer esbozo de la solución propuesta y para que el lector pueda ir componiendo una idea en su cabeza, paso a describir brevemente el sistema. Se creará una red inalámbrica en la que existirá un maestro (la torre de control) y varios esclavos (los aviones). Entre sí pueden enviarse información como el identificador de vuelo, destino, nivel de prioridad del avión, posición, etc... Además, cada uno de los elementos del sistema, contará con un módulo GNSS con el que podrán obtener su posición. Dicha posición se verá afectada por una serie de errores, que se calcularán en la torre de control mediante técnicas de posicionamiento GNSS diferencial. Las correcciones serán transmitidas junto con la ruta a seguir por el aeropuerto a cada una de las aeronaves. Cada una de las aeronaves recibirá esta información y deberá de ser capaz de interpretarla, y en base a esa interpretación, tomar una serie de decisiones sobre sus próximos movimientos por las pistas.

Habiendo presentado la idea, se procede a describir los objetivos y los retos a superar en este proyecto, para posteriormente dar una explicación en mayor profundidad y detalle pasando a describir el estado de la técnica en las tecnologías usadas y seguir con el desarrollo técnico de la solución. Se terminará con una propuesta de línea de trabajo para continuar con el proyecto en el futuro y su posible escalabilidad a un entorno real.

1.2. Objetivos

El objetivo de este trabajo es el de desarrollar un prototipo que implemente una primera versión de un sistema de guiado autónomo basado en tecnología GNSS y redes inalámbricas. Este sistema deberá de ser capaz de generar una ruta y guiar una aeronave a través de las pistas de rodadura de un aeropuerto teniendo en

cuenta las diferentes situaciones que puedan darse a lo largo de toda la maniobra, como por ejemplo, prioridad del tráfico o posibles emergencias en el camino.

Habiendo desarrollado el prototipo, se procederá a la realización de una serie de pruebas en exteriores con una maqueta a escala del Aeropuerto Adolfo Suárez Madrid-Barajas de 34 por 24 metros con las que se comprobará la funcionalidad del diseño elegido, la efectividad de los algoritmos de corrección de posición utilizados y por último, la viabilidad de implementación del sistema en un entorno real.

Por último, y teniendo en cuenta todo lo anterior, se indicarán unas líneas de trabajo futuras mediante las que continuar con el proyecto llevando las pruebas y el desarrollo del sistema a un entorno más realista.

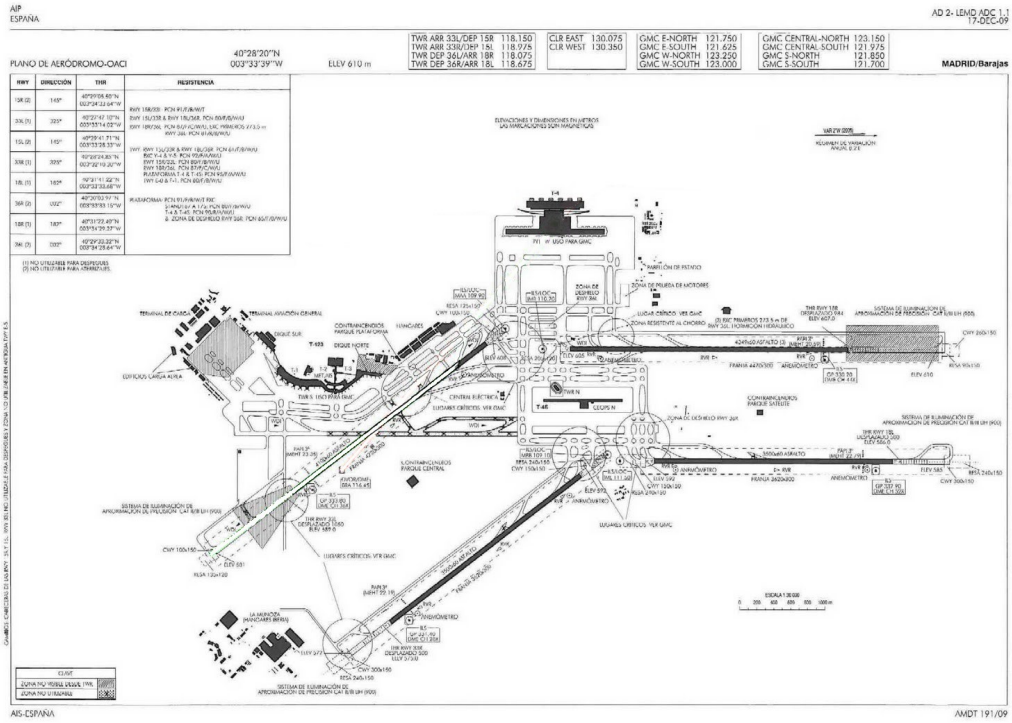

Figura 1: Plano a escala del Aeropuerto Adolfo Suárez Madrid Barajas a partir del cual se ha realizado una simplificación para la creación de la interfaz gráfica del prototipo

1.3. Estructura del documento

A lo largo del documento se trata de guiar al lector a través del problema planteado hasta que se llega a una solución factible. Se comienza dando una visión global del problema a resolver. A continuación se hace un planteamiento teórico sobre el estado actual de las tecnologías que se van a utilizar. Después de adquirir una perspectiva completamente teórica del proyecto y su posible solución, se pasa a la fase de diseño de las pruebas teóricas, donde se explicará el software ideado y su flujo de trabajo, así como el robot diseñado para el prototipo. Una vez se conoce el diseño del prototipo, se pasan a explicar las pruebas realizadas y a exponer los resultados obtenidos. En base a los datos recabados, se exponen las conclusiones obtenidas, se realiza una discusión sobre la viabilidad del diseño en un entorno real y se establecen las líneas futuras de trabajo a partir de las cuales se debería de continuar el proyecto en el futuro. Para una mejor comprensión de todo lo explicado, se adjuntan unos apéndices donde consultar el código realizado, los cálculos matemáticos aplicados, las características del hardware utilizado y, a modo de curiosidad, un breve presupuesto del desarrollo del prototipo, para ayudar a la comprensión y valoración del trabajo llevado a cabo.

2. Estado de la técnica

En esta segunda sección de la memoria del Trabajo de Fin de Grado se va a poner en contexto el estado en el que se encuentran actualmente las tecnologías utilizadas, se hará un breve resumen de su historia hasta llegar a su estado de desarrollo actual y hará una pequeña reflexión sobre cómo su uso y su implementación en el proyecto pueden influir positivamente en el rendimiento del mismo.

Se expondrá principalmente la tecnología de posicionamiento GNSS y más en concreto, de las fuentes de error presentes en el cálculo de posiciones en sistemas GNSS y las técnicas de posicionamiento diferencial. También se tratarán las aplicaciones de dicha tecnología en el ámbito de la aeronáutica, así como de su estandarización. Se hará un breve inciso en el sistema de aumento GNSS GBAS y por último se hablará sobre la conducción autónoma, sus niveles de automatización y las diferentes tecnologías implicadas en la misma.

2.1. La tecnología de posicionamiento GNSS

Ser capaces de determinar la posición en la que nos encontramos, ha sido siempre objeto de inquietud e interés para la humanidad. Desde los primeros marinos que utilizaban las estrellas como guías nocturnos, hasta los más modernos sistemas de posicionamiento global con un margen de error de centímetros hay una gran evolución tecnológica, pero el objetivo siempre ha sido el mismo: ser capaces de identificar nuestra posición en un mapa.

La exactitud con la que se conseguía este objetivo ha ido mejorando conforme la tecnología mejoraba, pero un punto de inflexión clave fue la puesta en marcha del primer sistema de posicionamiento global en 1959 (con una posterior apertura al público en 1967) fue Transit, el sistema de posicionamiento global perteneciente a la US Navy. Funcionó hasta 1996 cuando fue desmantelado y sustituido definitivamente por el sistema americano GPS (que ya había sido puesto en marcha en el mes de abril de 1995) [3].

El sistema de posicionamiento GPS es el sistema de posicionamiento más utilizado actualmente en el mundo. Consta de un total de 24 satélites distribuidos en 6

planos orbitales diferentes en los que cada uno de los satélites tiene una órbita casi circular con un semieje superior de 26.578 km y un periodo de unas 12 horas. [2]. Multiplicando mediante un sintetizador de frecuencias la frecuencia fundamental de 10,23 MHz obtenida a partir de unos relojes atómicos a bordo de los satélites, se consigue transmitir en 3 bandas de frecuencia distintas, la banda L1 (1575,42 MHz), la banda L2 (1227,60 MHz) y la banda L5 (1176,45 MHz). A través de estas tres bandas, los datos de efemérides, modelos ionosféricos y correcciones de reloj son transmitidos en espectro ensanchado CDMA, utilizándose una secuencia pseudoaleatoria de ensanchado diferente. El sistema de coordenadas que utiliza es el datum WGS84, un sistema utilizado en multitud de aplicaciones tanto militares como comerciales a nivel mundial.

La URSS no tardó en desplegar su propio sistema de posicionamiento con el que rivalizar a Estados Unidos. GLONASS comenzó a ser desplegado en 1982 y fue declarado como operativo en enero de 1996 ya bajo la cobertura de Rusia, al desintegrarse la URSS. Utiliza un total de 21 satélites distribuidos en tres planos orbitales en los que cada uno de los satélites opera en una órbita prácticamente circular con un semieje mayor de 25.510 km y un periodo de 11h y 16 min [2]. En cuanto a las bandas de transmisión, utiliza unas bandas propias entre 1602 y 1615,5 MHz y entre 1246 y 1256,5 MHz con canales de 0,5625 y 0,4375 MHz respectivamente. El sistema de coordenadas que utiliza también es propio de este sistema y es conocido como el datum PZ 90. Este sistema de coordenadas ha ido evolucionado hasta llegar al PZ 90.02 y PZ 90.11 en 2005 y 2014 respectivamente [3].

Por último, y a modo de comentario, la Unión Europea y China han desplegado sus propios sistemas de posicionamiento (Galileo y BeiDou respectivamente) con los que están alcanzado a los que han sido siempre los dos sistemas de posicionamiento global por excelencia. Ambos utilizan las mismas bandas que GPS a mayores de algunas bandas propias de sus sistemas. La intención que esconde el utilizar las mismas bandas entre varios sistemas es la posibilidad de tener interoperabilidad entre sistemas [14].

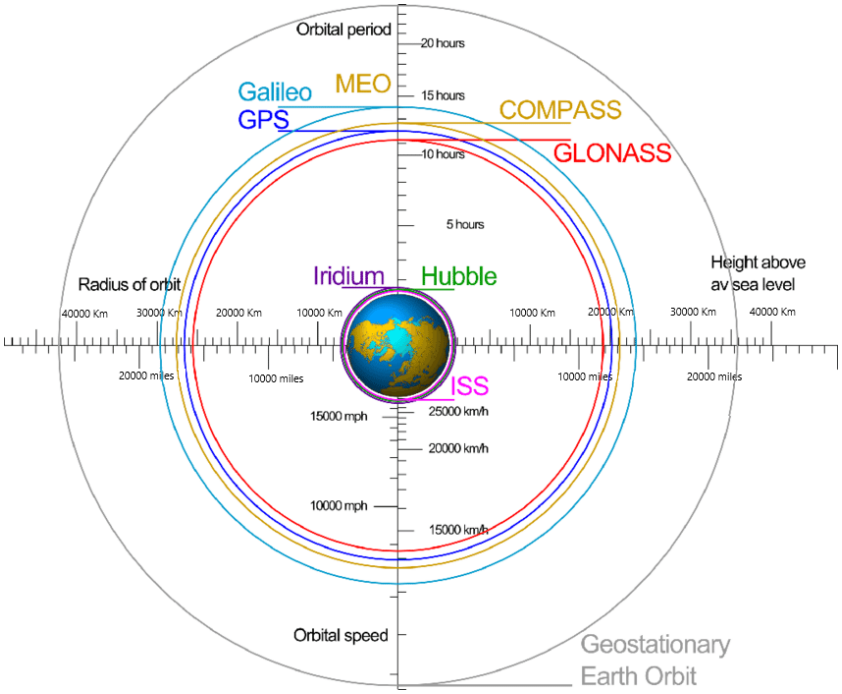

Figura 2: Representación de las constelaciones de todos los sistemas GNSS activos en comparación a otras órbitas de referencia [1]

Como último detalle en este punto introductorio a la tecnología GNSS, es necesario explicar el método básico para la obtención de una posición. La idea principal detrás de los sistemas GNSS es medir la distancia entre los satélites y los usuarios en la superficie de la Tierra. Debido a que los satélites orbitan el planeta en orbitas conocidas y muy precisas, se puede saber en qué posición se encuentra un satélite en cualquier momento. Apoyándose en esto, cuando un receptor recibe una señal procedente de un satélite, puede calcular el tiempo que le ha llevado a la señal recorrer el espacio entre el satélite y él mismo y en consecuencia, calcular una circunferencia sobra la que deberá estar. Haciendo esto con un mínimo de 3 satélites, el receptor puede estimar su posición en el espacio como el punto resultante de la intersección de las 3 esferas generadas a partir de las señales recibidas [2].

Esto, como casi todo en ingeniería, es un caso ideal. A lo largo del viaje entre el satélite y el receptor, la señal se va a ver afectada por multitud de efectos físicos que retardarán la señal, la dispersarán y en definitiva, cambiarán sus características. A continuación se mencionan los principales fenómenos a los que se ve sometida

una señal y se proponen distintas soluciones para minimizarlos y poder obtener una posición mucho más exacta del receptor.

2.1.1. Influencias físicas en la recepción de señales GNSS

Como se ha dicho en la sección anterior, independientemente del sistema de posicionamiento utilizado, las señales emitidas por los satélites se verán afectadas por una serie de efectos físicos que afectarán la señal y por tanto a la exactitud de posición determinada por el receptor. A continuación se pasan a explicar los principales fenómenos causantes de estas alteraciones y se propondrán una serie de soluciones para los mismos.

Podemos clasificar estos efectos como efectos ionosféricos, troposféricos, relativistas, multicamino, las interferencias y los errores de reloj. En concreto vamos a describir los más comunes y los que más afectan a la recepción, los ionosféricos, los troposféricos, los multicamino y los errores de reloj.

Efectos ionosféricos

Los efectos ionosféricos son una de las mayores fuentes de error en los sistemas de posicionamiento GNSS y una de las más complejas de caracterizar debido a la complejidad de las interacciones entre el campo magnético terrestre y la actividad solar. El error que generan puede variar entre 1 y 20 metros en un solo día. Además, hay que tener en cuenta que la ionosfera es un medio dispersivo, por lo que las alteraciones producidas dependen de la frecuencia.

Una señal electromagnética modulada, viajará a una cierta velocidad llamada velocidad de grupo, dependiente de la velocidad de fase y de su longitud de onda. Viene deteterminada por la ecuación:

$$v_g = v_p - \lambda \frac{dv_p}{d\lambda} \tag{1}$$

Si la señal se propagase por el vacío, la velocidad de grupo y la de fase serían ambas la velocidad de la luz en el vacío, pero en el caso de un medio dispersivo esto no es así. Las señales, por tanto, se verán retrasadas por el medio en el que

se encuentren. Esto se caracteriza mediante el índice de refracción

$$v_g n_g = c \tag{2}$$

$$v_p n_p = c \tag{3}$$

Según lo explicado en el capítulo 5.1 de [2], combinando estos factores mediante una integral a lo largo de todo el camino de la señal, se obtiene una expresión que permite caracterizar el error introducido por los efectos ionoféricos tanto para la velocidad de fase (4), como para la velocidad de grupo (5).

$$\delta_p = \int (n_p - 1) ds = \int \left(\frac{a_1}{f^2} + \frac{a_2}{f^3} \right) ds \tag{4}$$

$$\delta_g = \int (n_g - 1) ds = \int \left(-\frac{a_1}{f^2} - \frac{2a_2}{f^3} \right) ds \tag{5}$$

Omitiendo el segundo término del lado derecho de las expresiones, se puede determinar su valor como:

$$\delta_p \approx -\delta_g \approx \int \frac{a_1}{f^2} ds \tag{6}$$

Siendo a_1 y a_2 coeficientes que dependen de la densidad de electrones en la ionosfera (N_e) y pudiendo calcularse a_1 con la expresión:

$$a_1 = -40{,}3 N_e \tag{7}$$

Para eliminar este tipo de efectos, existen varias técnicas; la más simple consiste en la combinación de dos frecuencias en la señal transmitida, de tal forma que mediante una combinación lineal de las observaciones de la fase, los efectos ionosféricos son eliminados. Si reescribimos la expresión de los efectos ionoféricos en la fase como:

$$\delta_p = \frac{\int a_1 ds}{f^2} \tag{8}$$

Para una observación en dos frecuencias distintas obtendremos:

$$\delta_p(f_1) = \frac{\int a_1 ds}{f_1^2} \tag{9}$$

$$\delta_p(f_2) = \frac{\int a_1 ds}{f_2^2} \tag{10}$$

Combinando ambas expresiones de la siguiente forma, eliminaremos el efecto ionosférico producido en nuestra señal.

$$f_1^2 \delta_p(f_1) - f_2^2 \delta_p(f_2) = 0 \tag{11}$$

Este procedimiento consigue eliminar los efectos ionosféricos de primer orden, pero combinando tres, cuatro frecuencias, etc, conseguiríamos eliminar los efectos de segundo orden, tercer orden, etc...

Efectos troposféricos

La troposfera es la primera de las capas de la atmósfera, aquella que se encuentra en contacto con la superficie terrestre. Al contrario que la ionosfera, la troposfera es un medio no dispersivo a las frecuencias utilizadas por los sistemas GNSS, pero aun así, se produce un retardo debido a los efectos meteorológicos del lugar donde se esté recibiendo la señal. Este retardo en la señal se traduce en un error en la posición de unos 2 metros aproximadamente si nos encontramos justo debajo del satélite, y va aumentando a medida que nos alejamos y en consecuencia aumenta el ángulo de cénit respecto del satélite.

Este no es un error pequeño, y por tanto, es necesario tenerlo en cuenta a la hora de realizar los cálculos para obtener la posición. Depende de factores meteorológicos como la temperatura, la presión o la humedad de la troposfera. Además, hay que tener en cuenta que la troposfera no es homogénea a diferentes altitudes. Todo esto afecta de nuevo al índice de refracción del medio por el que se propaga la señal e introduce como ya se ha indicado, un retraso en la señal que se traduce en un error de posición.

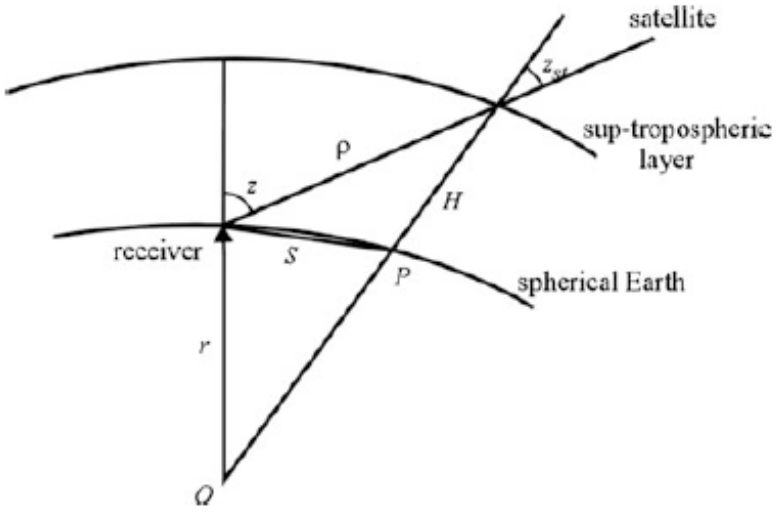

Figura 3: Representación del efecto de la troposfera en las señales GNSS [2]

Para corregir este error, basta con conocer la posición aproximada de recepción y el parte meteorológico de la zona. Con esta información se puede realizar un cálculo por ordenador aplicando el modelo de Saastamoinen (capítulo 5.2 de [2]) con el que obtendremos el valor del retraso producido por la troposfera.

Este modelo se puede enunciar mediante la siguiente ecuación:

$$\delta = \frac{0{,}002277}{\cos(z)} \left[P + \left(\frac{1255}{T} + 0{,}05 \right) e - B \tan z^2 \right] + \delta R \tag{12}$$

siendo δ el error generado en la señal (en metros), T la temperatura ambiente a la que se encuentra el receptor (en grados Kelvin), P la presión atmosférica (en milibares, mb), e la presión parcial del vapor de agua (en milibares también), y por último B y δR son correcciones en términos dependientes de H y z respectivamente siendo H la altitud de la estación del receptor y z la desviación en grados sexagesimales del cénit del receptor. Estos últimos valores de B y δR están tabulados en unas tablas propias del modelo de Saastamoinen.

Efectos multicamino

Este es un tipo de efecto importante en la recepción de una señal GNSS y que por tanto debemos tener en cuenta, especialmente en entornos urbanos o terreno montañoso. En este tipo de fenómenos, el receptor GNSS recibe la misma señal desde más de una dirección. Esto es debido a que la señal enviada desde el satélite puede rebotar con objetos cercanos a la antena y reflejarse dando lugar a dos lecturas de la misma señal en el receptor, pero en dos instantes de tiempo diferentes. Si el posicionamiento GNSS se basa en el tiempo que tarda en llegar una señal desde un satélite hasta el receptor, ¿qué señal debe de tomar como la correcta para realizar el cálculo?

Normalmente, la señales que son reflejadas, tienen una menor potencia que la señal original y por tanto, una menor relación señal a ruido (SNR), lo que hace que sean relativamente fáciles de distinguir. En cualquier caso, la mejor solución al efecto multicamino, ya que depende del entorno circundante a la antena, es situar nuestra antena en el lugar más despejado posible. En este proyecto, tratándose de una aplicación aeroportuaria, no debería de suponer un gran problema ya que a priori, los receptores deberían de estar en entornos bastante despejados, por lo que no se va a entrar en mayor profundidad en este tipo de efectos.

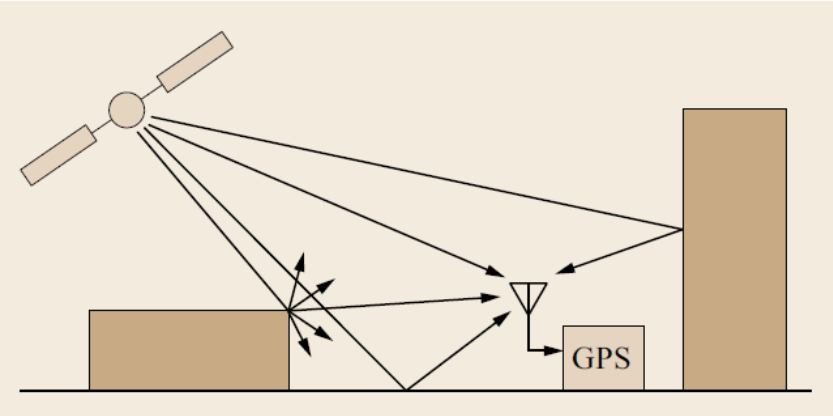

Figura 4: Representación del efecto multicamino en la recepción de señales GNSS [3]

Basado en el capítulo 5.6 de [2] y el capítulo 15 de [3].

Errores de reloj

Debido a que el método utilizado por los sistemas GNSS en la obtención de la posición se basa en la medida de tiempos de propagación entre emisor y receptor, es muy importante que los relojes de ambos elementos estén perfectamente sincronizados para evitar errores en el cálculo de tiempos que lleven a un error en el posicionamiento del receptor. Aun así, existen errores entre los relojes de los sistemas GNSS debido a varios factores. El primero depende de la exactitud del reloj, el segundo con la propia velocidad de los satélites y el tercero y último, con la frecuencia de trabajo del reloj.

El primer tipo de error influye en el cálculo de rutas. Si un reloj tiene un error δt, causará un error en el calculo de la fase de $c\delta t/\lambda$. Dado que c es un número muy grande, el error que esto puede suponer, es muy grande también. Por eso, se deben de utilizar relojes de muy alta calidad y exactitud en los dispositivos GNSS.

Para comprender el segundo tipo de error de reloj, hay que tener en cuenta la dependencia del tiempo tanto de la velocidad como de la posición del satélite. Esto causa un error en el cálculo de la posición del satélite que viene determinado por $v_s\delta t$ donde v_s es el vector velocidad del satélite. Este error puede ser compensado realizando una estimación del δt con una precisión de en torno a 10^{-6} segundos.

Por último, y debido a efectos relativistas debido a la alta velocidad a la que viaja el satélite para mantenerse en órbita, se produce un desajuste en la fase del reloj con respecto a los relojes del receptor en la Tierra de valor $c\delta t/\lambda$, que es equivalente a un error en la frecuencia de trabajo del mismo con un valor $f\delta t$. Igual que en el caso anterior, realizando una correcta aproximación de δt este error puede ser corregido, aunque es un error que se tiene que tener en cuenta al realizar un procesado de datos en los que hay que corregir el efecto Doppler.

Por todas estas situaciones, la sincronización y exactitud de los relojes tanto a bordo de los satélites como de los incorporados en el receptor debe de ser algo a tener muy en cuenta en el diseño de los mismos.

Basado en el capítulo 5.5 de [2].

2.1.2. Posicionamiento GNSS diferencial

Como ya hemos visto, los sistemas de posicionamiento siempre han estado afectados por una serie de efectos que derivaban en errores en la posición determinada por el receptor. Esto supone un problema, ya que muchas de las tecnologías que los implementan, sobretodo las tecnologías de posicionamiento en tiempo real aplicadas a objetos en movimiento, si bien pueden asumir un cierto nivel de error, necesitan un mínimo de exactitud en la posición que obtienen a partir de sus cálculos.

Ya que son errores provenientes de efectos que en la mayoría de los casos no podemos controlar o son inherentes al entorno en el que se encuentra nuestro receptor, una de las soluciones que se ha encontrado para minimizar estos errores en la posición ha sido el posicionamiento GNSS diferencial.

Este tipo de posicionamiento se basa en el cálculo del error producido por los efectos antes mencionados y la sustracción del mismo de la posición que se está obteniendo. Existen dos técnicas, PPP (Precise Point Positioning) y RTK (Real Time Kinematic).

En la primera es el propio receptor quien calcula el error que está percibiendo. Para ello, debe de conocer las órbitas y los relojes de los satélites Esto solo será posible si se siguen los estándares internacionales. En la segunda técnica, se precisa el uso de una estación base, de la cual se debe de conocer su posición exacta. En esa estación base se realizarán una serie de cálculos para hallar el error introducido por los efectos antes mencionados, y retransmitirlos al receptor mediante un enlace de datos para que calcule su posición corregida. Esta última aproximación es la que se seguirá en este proyecto, ya que es la más utilizada en sistemas en los que el objeto se halla en movimiento.

En este último caso, las correcciones calculadas serán válidas siempre que el receptor final se halle en un radio de entre 10 y 15 kilómetros en torno a la estación base como mucho. Más allá de esa distancia, no se puede seguir considerando la atmósfera como uniforme y se deberían de hacer unos nuevos cálculos. Si se necesitase extender esta distancia, lo que se debería de hacer es crear una red de estaciones base, y que el objeto en movimiento fuese obteniendo el error de un

centro de procesado donde se calculase un error válido para todo el área cubierta.

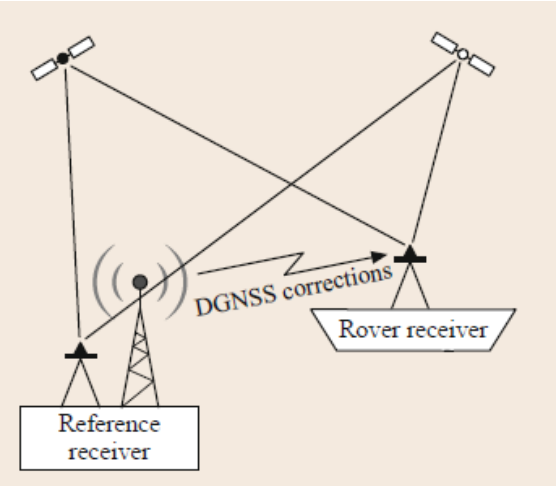

Figura 5: Ejemplo de sistema de posicionamiento RTK [3]

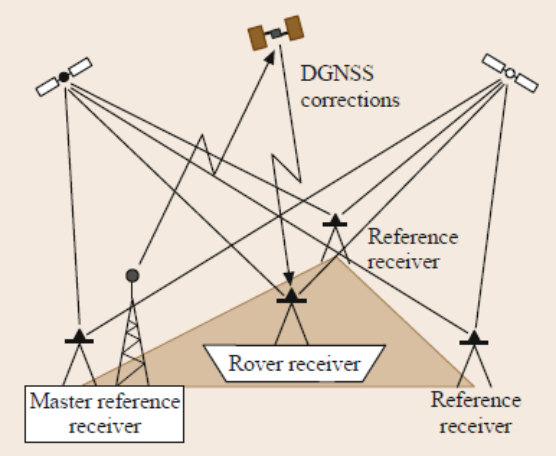

Figura 6: Ejemplo de red de posicionamiento RTK [3]

Basado en el capítulo 26 de [3], los capítulos 10.1 y 10.2 de [14] y el contenido de [15].

2.2. Aplicaciones GNSS en la aeronáutica

La tecnología de posicionamiento GNSS ha tenido una gran acogida en el sector aeronáutico desde sus inicios debido a la capacidad de navegación global de alto nivel que proporciona junto con un bajo coste y un pequeño tamaño, algo que siempre es de agradecer en el mundo aeronáutico.

Por eso, hoy en día se utiliza en todas las aeronaves comerciales del mundo, donde es requerido como un estándar básico del avión y es altamente utilizado en la aviación civil de recreo, donde pequeños dispositivos de posicionamiento GNSS, en combinación con varios sensores más, dotan a una pequeña avioneta de navegación instrumental, permitiéndola realizar vuelos de mayor distancia y en multitud de condiciones meteorológicas que mediante VFR (Visual Fligth Rules) sería imposible de llevar a cabo.

2.2.1. Regularización de la tecnología GNSS en aeronáutica

Durante los años 80, cuando el Presidente de los Estados Unidos Ronald Reagan decidió autorizar el uso del sistema GPS con intenciones civiles, se permitió incorporar el uso de esta tecnología a la aviación. Sin embargo, dado el carácter internacional, y la alta regulación del sector, existen varios acuerdos internacionales mediante los que se regula y estandariza el uso y las características comunes que deben de tener los receptores GPS utilizados en la aviación.

Estos estándares son dictados por la ICAO, la Organización Internacional de Aviación Civil, y al ser una agencia perteneciente a las Naciones Unidas, es reconocida por 191 países en el mundo. En 1989 estableció que el sistema de coordenadas utilizado para la aviación sería el datum WGS84. Esto permitió asegurar la interoperabilidad entre fabricantes y las diferentes infraestructuras que se construirían en cada país.

En un principio, debido a la complejidad de estos sistemas y al coste que tenían, los sistemas GNSS, solo se utilizaban en los grandes vuelos de aviación comercial, pero hoy en día, casi cualquier avioneta que se pueda pilotar con la licencia de piloto privado (PPL) posee este tipo de tecnología.

Figura 7: Receptor GNSS de una aeronave comercial [3]

Figura 8: Display de un módulo GNSS de la marca Garmin para aviación privada [3]

Además de fijar el datum a utilizar, ICAO también fijó unos requisitos mínimos de rendimiento para los sistemas de posicionamiento GNSS que permitiesen asegurar, en función de la fase del vuelo en la que se encontrase, su posición con un error máximo tanto horizontal como vertical, unas probabilidades mínimas de disponibilidad, continuidad e integridad del sistema, así como un tiempo máximo para alertar de cualquier posible fallo que se produjese en el sistema de posicionamiento. Todos estos parámetros vienen resumidos en la tabla/imagen a continuación.

Para poder comprenderla mejor, se van a comentar los cuatro parámetros utilizados para describir el rendimiento de un sistema GNSS embarcado en una aeronave, la exactitud (accuracy), la integridad (integrity), la continuidad (continuity) y la disponibilidad (availability).

Typical operation	Accuracy horizontal[a,c]	Accuracy vertical[a,c]	Integrity[b]	Time-to-alert[c]	Continuity[d]	Availability[e]
En-route	3.7 km	N/A	$1-1\cdot10^{-7}$/h	5 min	$1-1\cdot10^{-4}$ to $1-1\cdot10^{-8}$/h	0.99–0.99999
Terminal	0.74 km	N/A	$1-1\cdot10^{-7}$/h	15 s	$1-1\cdot10^{-4}$ to $1-1\cdot10^{-8}$/h	0.99–0.99999
Non-precision approach (NPA)	220 m	N/A	$1-1\cdot10^{-7}$/h	10 s	$1-1\cdot10^{-4}$ to $1-1\cdot10^{-8}$/h	0.99–0.99999
APV-I[h]	16 m	20 m	$1-2\cdot10^{-7}$ in any approach	10 s	$1-8\cdot10^{-6}$ per 15 s	0.99–0.99999
APV-II[h]	16 m	8 m	$1-2\cdot10^{-7}$ in any approach	6 s	$1-8\cdot10^{-6}$ per 15 s	0.99–0.99999
Category 1 precision approach[g]	16 m	6–4 m[f]	$1-2\cdot10^{-7}$ in any approach	6 s	$1-8\cdot10^{-6}$ per 15 s	0.99–0.99999

Figura 9: Tabla resumen con los estándares de rendimiento exigidos por la ICAO en los sistemas GNSS embarcados en una aeronave [3]

Exactitud La exactitud en la medida del error en la posición, que es la diferencia entre la posición estimada y la posición verdadera. Por ejemplo, siguiendo los requisitos en exactitud de la tabla, una exactitud del 95 % en la exactitud horizontal en la fase de vuelo "en ruta", supone que la probabilidad de que el error en la posición horizontal sea menor a 3,7 km en el 95 % de las ocasiones.

Integridad El riesgo de integridad es la probabilidad de que un usuario experimente un error de posición horizontal mayor que el Límite de Alerta Horizontal (por sus siglas en inglés, HAL) o en el caso de la posición vertical, el VAL sin que sale una alerta dentro del tiempo máximo de alerta (TTA, Time-to-alert) en cualquier instante de tiempo.

Continuidad La continuidad es definida como la probabilidad con la que un usuario es capaz de determinar su posición con la exactitud establecida y que es capaz de monitorizar la integridad de su posición a lo largo de todo el vuelo en cada fase correspondiente del vuelo.

Disponibilidad La disponibilidad es lo mismo que la continuidad, pero referido solamente al inicio de la operación.

La información en la que se basa esta subsección ha sido obtenida del capítulo 30 de [3], los puntos 4.4, 4.5 y 4.6 de [16] y del punto 2.19 de [17].

2.2.2. GBAS (Ground Based Augmentation System)

Un sistema de aumento de un sistema de navegación global es un método de mejora de los parámetros de rendimiento de este sistema (integridad, continuidad, exactitud y disponibilidad) mediante el uso de información externa al sistema GNSS en el proceso de resolución de la posición del receptor.

Existen varios sistemas de aumento como por ejemplo DGNSS, SBAS, PPP o RTK, pero en nuestro caso nos vamos a centrar en el sistema GBAS (Ground Based Augmentation System) ya que es el sistema de aumento utilizado en los aeropuertos para el guiado de los aviones durante las maniobras de aproximación y despegue.

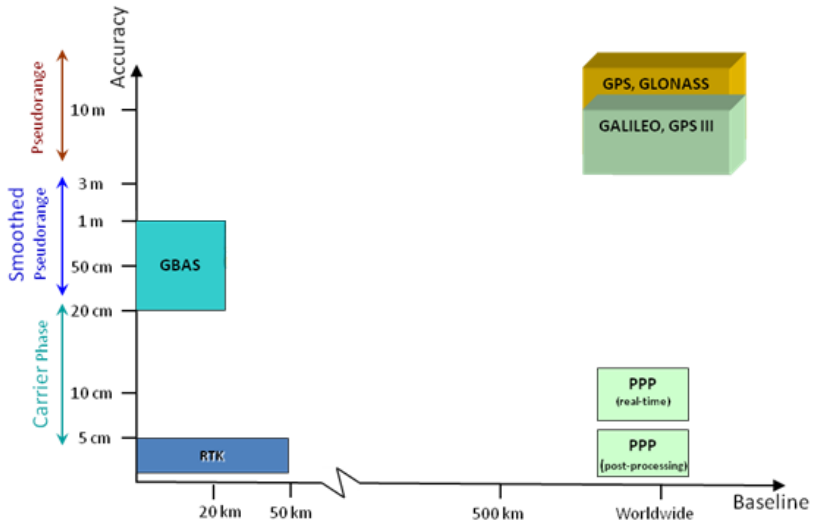

Figura 10: Comparación del error máximo obtenido por cada uno de los sistemas de aumento GNSS [4]

El sistema GBAS es un sistema de aumento que permite tener un error en la posición del receptor de entre 0,2 y 1 metros. Para conseguir esto, se basa en la asunción de que el receptor y una estación terrena se encuentran relativamente cercanas entre sí (a menos de 15 km) y por tanto, los errores ionosféricos y troposféricos a los que se ve sometida la señal son iguales [4].

El sistema funciona de la siguiente manera: una estación terrena de la que se conoce su posición exacta y que cuenta con al menos cuatro receptores GNSS de

gran exactitud separadas entre sí lo suficiente como para poder despreciar el efecto multicamino producido entre ellas, calcula su posición. A partir de las posiciones recibidas y conociendo la suya propia, puede calcular el error producido debido a los retrasos introducidos por la ionosfera y la troposfera. Una vez la estación terrena ha obtenido el error para cada uno de los satélites que tiene a la vista, estos son retransmitidos a todos los aviones cercanos al aeropuerto a través de un enlace de radio de VHF en la banda del localizador ILS (108-118 MHz) utilizado para la aproximación instrumental al aeropuerto [3].

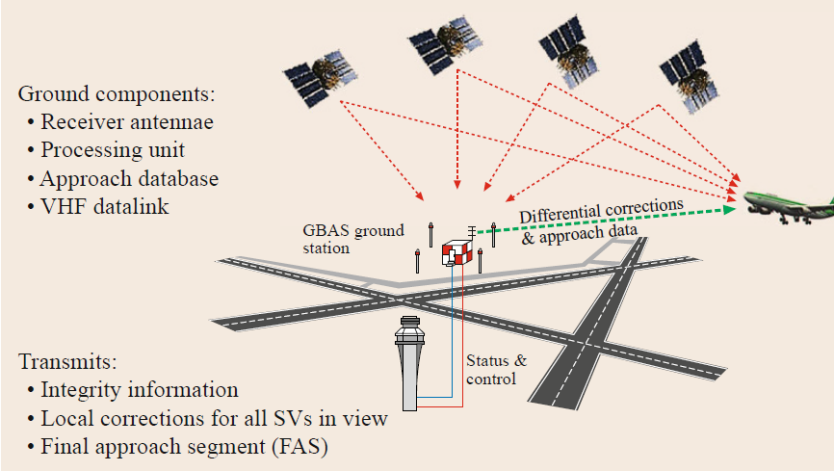

Figura 11: Esquema explicativo de la arquitectura de un sistema GBAS [3]

Además, como indica ICAO en [18], debido a la exactitud que ofrece en el cálculo de posiciones, GBAS es una tecnología que a mayores de una seguridad extra a la hora de aterrizar, tiene multitud de aplicaciones y beneficios en los aeropuertos. Algunas de estas aplicaciones son, la reducción de zonas críticas y sensibles en el aeropuerto, la posibilidad de realizar una maniobra de aproximación a pista siguiendo un patrón en curva, suministro de varios ángulos de planeo de aproximación o la posibilidad de realizar un guiado de aproximaciones finales frustradas.

2.3. Técnicas de conducción autónoma

En los últimos años y sobretodo en el ámbito automovilístico, se está produciendo una revolución tecnológica gracias a potentes tarjetas gráficas que permiten realizar procesado de imagen en tiempo real, y a los nuevos protocolos de comunicaciones

entre los vehículos y el resto de las infraestructuras que permiten el intercambio de grandes cantidades de información a altas velocidades. Esta revolución está dando sus primeros frutos y el más conocido y llamativo es el de la conducción autónoma.

Son muchos los fabricantes automovilísticos que se están volcando en su desarrollo para aumentar la seguridad en sus vehículos tanto para los pasajeros como para los transeúntes u otros vehículos. Tesla Inc ha sido una de las primeras marcas en ponerse a la vanguardia, con su disruptivo modelo de berlina tecnológica, el Model S, pero son muchos los fabricantes que quieren seguir su camino e ir avanzando cada vez más en el ámbito de la conducción autónoma. Algunos ejemplos de esto son Audi, Volvo o Renault, quienes ya permiten a sus clientes disfrutar de controles de velocidad adaptativos, frenadas de emergencia automática, seguimiento de carriles en autovías y autopistas o aparcamiento autónomo.

2.3.1. Los 6 niveles de automatización en la conducción

Según describe la NHTSA (National Highway Traffic Safety Administration) de los Estados Unidos [5], existen seis niveles de automatización en la conducción, del 0 al 5, por orden ascendente de automatización.

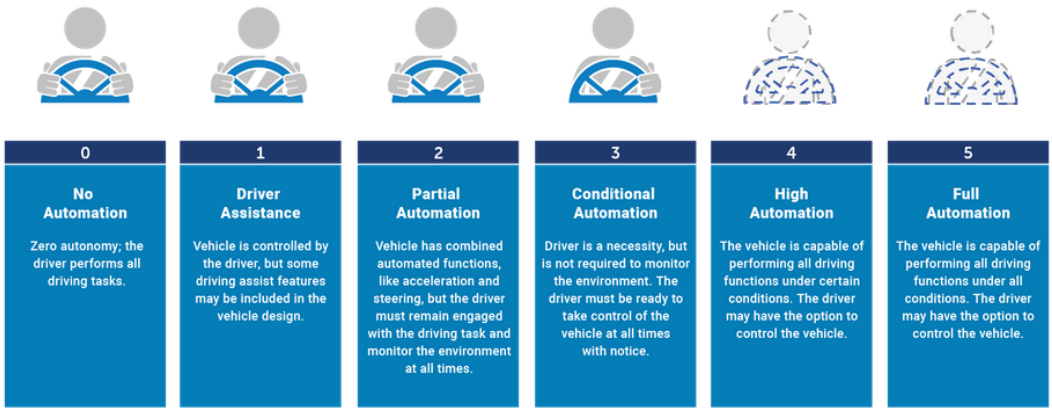

Figura 12: Gráfico explicativo de los 6 niveles de automatización existentes para vehículos [5]

En el primer nivel, el 0, se encuentra la conducción normal. No hay ningún tipo de asistencia y todo el trabajo lo debe de hacer el piloto. A medida que se va avanzando

en los niveles se llega hasta el nivel 5, en el que no es necesaria ningún tipo de intervención ni monitorización por parte del piloto. El sistema es completamente autónomo.

Actualmente los vehículos normales se encuentran en el nivel 1, con alguna asistencia como el control de crucero o en el nivel 2, donde el coche puede girar por si mismo y controla la aceleración o el frenado del coche. Este sería el caso del control de velocidad adaptativo, el aparcamiento autónomo o el seguimiento de carriles en autovías.

Algunos de los vehículos que se encuentran a la vanguardia de la conducción autónoma, se encuentran en el nivel 3, donde el piloto ya solo debe de estar presente para poder tomar el control si llega a ser necesario, pero durante la mayor parte del tiempo no tiene que monitorizar el sistema. Este sería el caso de la marca Tesla [6] o algunos de los sistemas en pruebas de la marca Nvidia [19], donde el vehículo es capaz de reconocer señales y situaciones y tomar decisiones más complejas por si mismo en función del entorno.

Por último, se encuentran los niveles 4 y 5 que son aquellos en los que no es necesaria la presencia del piloto en la mayoría de las situaciones o en ninguna, en función del nivel y que es hacia donde se dirigen las investigaciones al respecto.

Según la NHTSA [5], para el 2025, se contarán con vehículos completamente autónomos en las carreteras.

2.3.2. Tecnologías implicadas en la conducción autónoma

La conducción autónoma, pese a ser un gran avance tecnológico y una mejora innegable en la seguridad vial, necesita de una gran cantidad de recursos para poder llevarse a cabo. Requiere de una gran capacidad de procesamiento de imágenes, así como de toda la información proveniente de todos los sensores incorporados.

En el caso de Tesla Inc [6], como describen en su página web, cada uno de sus vehículos cuenta con 8 cámaras que envía información de forma constante al ordenador de abordo y es procesada en tiempo real permitiendo identificar personas, señales, líneas de la carretera y otros coches. Además, cuenta con una serie de

sensores de ultrasonidos que cubren todo el espacio al rededor del coche hasta una distancia máxima de 8 metros y con un radar frontal que permite identificar obstáculos hasta una distancia máxima de 160 metros por delante del vehículo.

Todo este conjunto de sensores necesita una gran capacidad de procesado que permita transformar toda esa información en decisiones. Para ello, se utilizan potentes tarjetas gráficas como las que está desarrollando Nvidia [19] o la propia Tesla , gracias las cuales cada vez se está más cerca de una conducción completamente autónoma.

Figura 13: Imagen de un coche autónomo de Tesla obtenida de un vídeo donde se ve que el conductor no manipula en ningún momento el vehículo y se ven algunas de las imágenes procesadas obtenidas mediante las cámaras del vehículo a la derecha de la imagen [6]

3. Diseño del prototipo

Para la correcta realización de las pruebas y la obtención de datos fiables, se ha diseñado y construido un prototipo que, a pequeña escala, con un número reducido de aviones y salvando las distancias de calidad entre sensores, permita extraer datos y conclusiones sobre los que fundamentar las conclusiones obtenidas del trabajo.

El prototipo consta, a grandes rasgos, de dos partes. La primera parte es la relacionada con el software, donde se encuentran la implementación de los algoritmos de corrección de posición y cálculo de rutas, el flujo de trabajo a seguir durante la maniobra de rodaje o la interfaz gráfica entre otros. La segunda parte es la implementación física del proyecto. Esto implica todo lo relacionado con el hardware del prototipo. En esta parte se encuentran los sensores GNSS, el ordenador que adopta el rol de torre de control y el robot a escala que tendrá el papel de representar un avión moviéndose por el aeropuerto.

A continuación se pasa a describir en detalle la arquitectura del software desarrollado y el montaje hardware realizado.

3.1. Implementación software

Toda la programación se ha realizado en el lenguaje de programación Python en su versión del intérprete 3.6. Se ha elegido este lenguaje debido a la versatilidad que ofrece y al gran número de recursos y librerías que existen en la red de forma completamente gratuita. Esto último es algo que a lo largo de todo el proceso de desarrollo ha sido de gran ayuda.

El paradigma de programación utilizado es el paradigma de programación orientado a objetos con un flujo de trabajo multihilo. Esto es debido a que cuando se plantearon todas las tareas que debía de hacer un mismo agente (entiéndase por agente el ordenador de abordo de una aeronave o el ordenador de la torre de control), se observó que varias debían ejecutarse de forma simultánea, algo que solo era posible de realizar si se implementaba una solución multiproceso o multihilo.

Las soluciones multiproceso son aquellas en las que para cada nueva tarea, se crea un nuevo proceso y todos los procesos creados se ejecutan de forma simultánea en

CPUs distintas del ordenador. Por esto, tienen como ventajas la sencillez tanto en la programación, como en la ejecución. Es como tener varios programas ejecutándose a la vez, cada uno con un intérprete de Python distinto. Pero es precisamente ese, su principal problema. Requiere de una gran cantidad de recursos en el ordenador, porque cada nuevo proceso que se crea, requiere una CPU para si mismo durante todo el tiempo que dure su ejecución, aunque esté largos periodos de tiempo esperando. Eso limita mucho el rendimiento del programa, además de que el sistema vería limitado el número de aviones a los que podría atender de forma simultánea, al número de CPUs del servidor que lo implementase. [7] y [20]

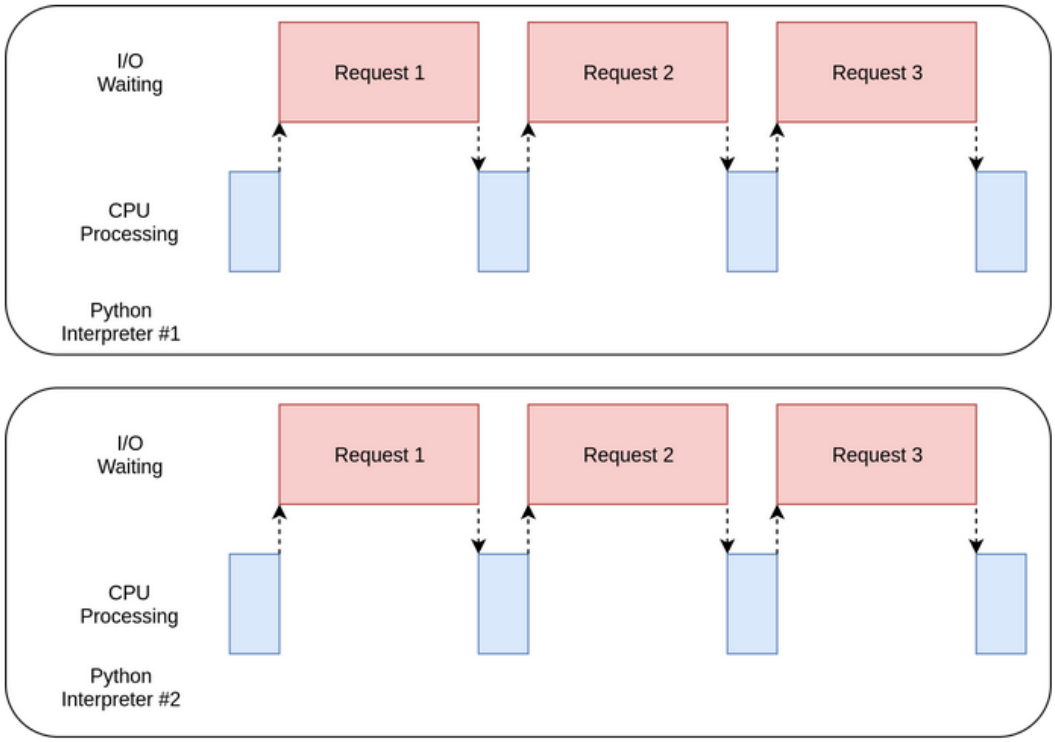

Figura 14: Esquema temporal de una solución multiproceso procesando varias solicitudes en paralelo [7]

Por el contrario, una solución multihilo es una solución en la que para cada nueva tarea, se crea un nuevo hilo. Los hilos son conjuntos de operaciones que no requieren de una CPU en exclusiva para trabajar y es el propio sistema operativo del ordenador quien decide qué hilo, cuándo y en qué CPU se ejecuta, para in-

tercalarlos a la vez con el resto de procesos del ordenador que son ejecutados en segundo plano. Su principal ventaja es la rapidez y la menor cantidad de recursos que necesita (tan solo una CPU es suficiente para implementar esta opción de computación paralela), pero por contra es el más complejo de utilizar, ya que el código se complica con respecto a la opción multiproceso. [7] y [20]

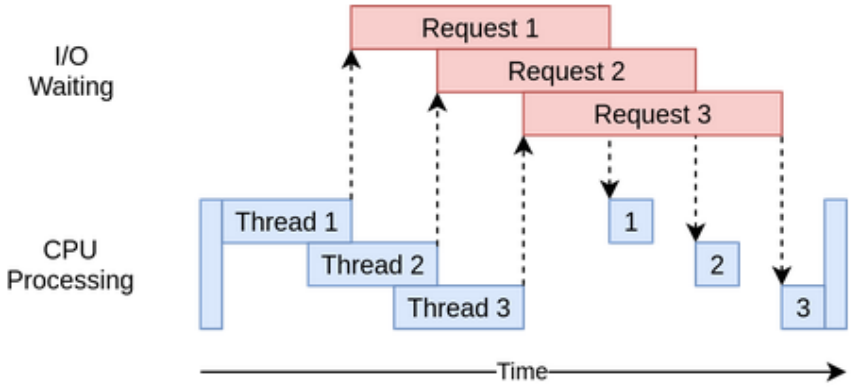

Figura 15: Esquema temporal de una solución multihilo procesando varias solicitudes en paralelo [7]

Finalmente, se terminó por elegir la solución multihilo, debido a que una implementación multiproceso desperdiciaba una mayor cantidad de tiempo entre una tarea y otra, y además era más exigente en cuanto a los recursos necesarios por parte del ordenador. Esto último fue el factor decisivo en la decisión final, ya que en la Raspberry (que es el ordenador con el que se creó el robot que hizo las veces de avión) se disponía de una capacidad de computación mucho menor, con una única CPU, por lo que no se podía implementar una solución multiproceso.

En cuanto al sistema de coordenadas utilizado, se utilizan dos. Los sensores GNSS utilizados captan señales del sistema GPS, por lo que obtenemos sus coordenadas en el datum WGS84. Sin embargo, para trabajar en el plano de la interfaz gráfica diseñada para el prototipo, se utiliza el sistema de coordenadas UTM. Se ha elegido este sistema de coordenadas, porque se trata de un sistema que hace proyecciones en base a husos esféricos que permiten hacer una proyección lineal, despreciando la curvatura del planeta. Además, la unidad fundamental del sistema de coordenadas UTM es el metro, por lo que se simplifican mucho los cálculos a realizar. En el

sistema UTM se trabaja en una cuadrícula orientada de forma sur-norte en el eje de ordenadas y oeste-este en el eje de abscisas, lo que para el diseño de una maqueta resulta muy cómodo. Basta con calcular la posición de una de las esquinas y sumar y restar cantidades en metros para hallar la posición del resto. Por estas razones, se eligió este sistema y por tanto, se deberá de realizar la transformación de coordenadas del sistema WGS84 al UTM (huso 30, al cual, pertenece Valladolid). Esta transformación se realiza utilizando una librería de Python llamada *python-utm* disponible en el portal de descarga libre de librerías de Python PyPi [21].

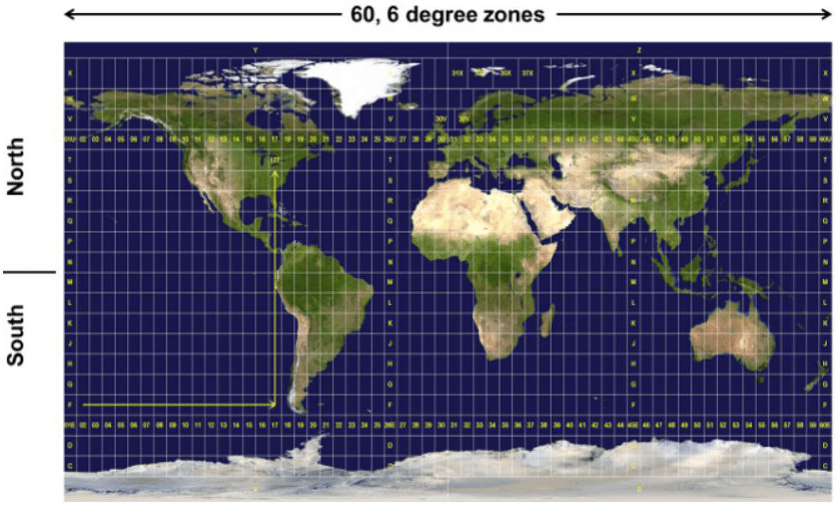

Figura 16: Representación según el sistema de coordenadas UTM de un mapamundi [8]

Por último, para poder comunicar la torre de control y la aeronave, se ha utilizado la librería de Python *socket*. Esta librería nos permite crear conexiones TCP/IP o UDP/IP entre dos puertos de una o más interfaces de red distintas. En este caso concreto se han utilizado las tarjetas de red inalámbricas (802.11ac - Wifi) disponibles tanto en la torre de control como en la aeronave.

Una vez establecidas las líneas generales para el diseño del código, se comentará el flujo de trabajo tanto de la torre de control, como de las aeronaves.

3.1.1. Software en la torre de control

En la torre de control el flujo de trabajo es el siguiente: Se comienza calibrando el sensor GNSS que se va a utilizar. Se calcula el error base que posee el sensor pidiendo al usuario introducir la posición real de la torre y restándola a la posición obtenida mediante un promediado de 10 muestras obtenidas a través del módulo GNSS. Habiendo calculado ya el error base del sensor, se procede a calcular la posición de las 4 esquinas del mapa en base a una relación de distancias predefinidas en el programa.

Ya con el dispositivo GNSS y las medidas del mapa calibrados, se inicializa la interfaz gráfica (desarrollada utilizando la librería *pygame* [22]) y se muestran con un asterisco la posición de la torre de control y de las 4 esquinas a modo de señal de que el programa está listo para usarse.

En este momento comienza el bucle principal del programa de la torre de control. Este bucle es el orquestador de los diferentes hilos de la torre. Contará con una serie de hilos encargados de gestionar todas las actividades de la torre de forma paralela. Cada uno de los hilos será un servidor TCP escuchando en un puerto distinto de la interfaz de red y que se encarga de gestionar una única conexión con una aeronave. Es el encargado de hacerle llegar toda la información necesaria al avión, como por ejemplo la información correspondiente a la ruta que debe seguir, su nivel de prioridad (número del 1 al 10) o su número de conexión (un número del 1 al 200) que le servirá como identificador dentro del aeropuerto. Dentro de esa información, se encuentra el error relativo percibido por el sensor GNSS. Aparte del error base que el sensor GNSS posee intrínsecamente y que ya se ha calculado en el proceso de calibrado del mismo, el sensor de la torre también percibirá una serie de errores relativos en función de las condiciones atmosféricas en el momento y lugar en los que se utilice el sistema, que serán comunes a todos los receptores GNSS del aeropuerto debido a su proximidad física. Cada uno de los hilos debe de calcular dicho error y transmitirlo al avión al otro lado del socket para que pueda hacer el cálculo diferencial y corregir su posición.

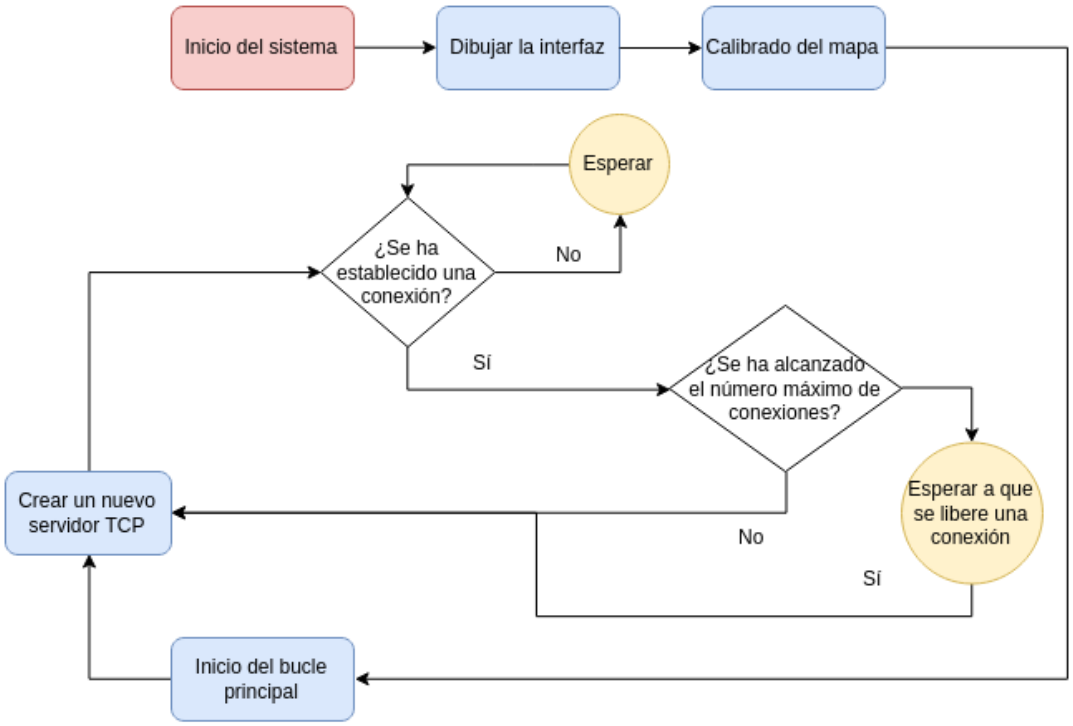

Figura 17: Diagrama de bloques donde se muestra el flujo de trabajo del software implementado en la torre de control

Aunque desde un punto de vista de eficiencia de recursos sería más simple que hubiese un único hilo encargado de hacer el cálculo del error y que los hilos servidores solamente tuviesen que transmitirlo, esto no se ha hecho así, ya que en Python no pueden acceder dos hilos de forma simultánea a una misma posición de memoria. Por eso debemos calcular el error de manera local, en cada uno de los hilos servidores. En cualquier caso, este error será común a todos los hilos porque todos obtienen los datos del mismo sensor GNSS.

En un primer momento, se crea solamente un servidor TCP en el puerto 65000 y en el momento en el que se recibe una solicitud de conexión y esta se ha establecido correctamente, se crea un nuevo servidor en el siguiente puerto disponible (65001, 65002, etc...) del ordenador. La creación de nuevos servidores tiene un límite máximo establecido a elección del usuario. Una vez se alcance este número máximo, no se podrán establecer más conexiones.

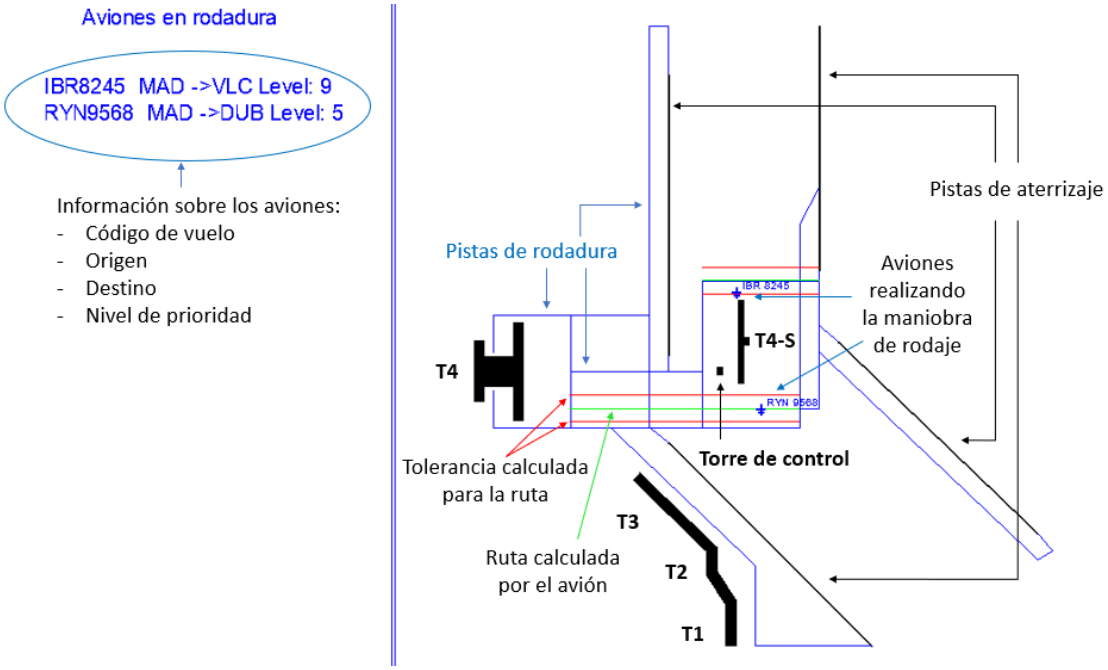

Figura 18: Interfaz desarrollada donde se muestran las posiciones y la información de los aviones en la maqueta a escala del aeropuerto

3.1.2. Software a bordo de la aeronave

En el caso del software utilizado en el avión, se sigue un flujo de trabajo muy parecido al establecido por la torre de control. Se trabaja de nuevo con una programación orientada a objeto y con un enfoque multihilo. Existen 4 hilos en cada avión.

El primer hilo es el encargado de, en primer lugar, realizar la calibración del error base del sensor GNSS utilizado de la misma manera en que lo hace la torre de control. En segundo lugar, también es responsable de mantener la conexión TCP con la torre de control. A través de esta conexión le solicita una ruta, un identificador y el error relativo que percibe la torre de control a través de su módulo GNSS. Además, una vez cuenta con toda esa información, corrige su posición en base al error relativo que recibe y transmite a la torre toda la información relativa al vuelo (compañía aérea, destino, posición corregida, nivel de prioridad y pista de rodaje en la que se encuentra).

El segundo hilo es el encargado de administrar un servidor UDP configurado para escuchar en el puerto 655001. Puerto al que todos los aviones transmiten en modo *broadcast* la misma información que intercambian con la torre de control. De esta forma, todos los aviones saben dónde está el resto de aeronaves en el aeropuerto en todo momento, y gracias a sus posiciones, saben si están cerca o lejos de ellos. Si están demasiado cerca (1,5 m en esta simulación), los aviones comprueban los niveles de prioridad y el que posea un menor nivel de prioridad (o un identificador más alto en el caso de igualdad de prioridades), se para y cede el paso al otro avión. Este proceso es explicado con detalle en el punto 4.4 de la memoria.

El tercer hilo es el encargado de asumir el papel de cliente UDP y envía de forma periódica en modo *broadcast* su información a través del puerto 65500 y hacia el 65501 para que el resto de aviones del aeropuerto conozcan de su existencia y sus datos.

El cuarto y último hilo es el encargado del movimiento de la aeronave. Siempre que tenga una distancia suficiente con el resto de aviones para poder avanzar, realiza el cálculo de la ruta entre los puntos especificados desde la torre de control. A mayores de la ruta calculada, se calculan unas paralelas a la misma que sirven de tolerancia. Si se detecta que la aeronave va a salirse por uno de los laterales de esta tolerancia, corrige automáticamente el rumbo a izquierda o derecha para volver a centrarse. Este proceso es explicado con detalle en el punto 4.3 de la memoria.

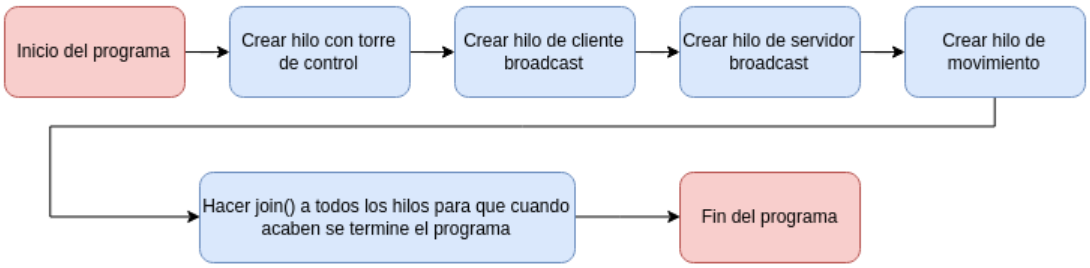

Figura 19: Diagrama de bloques donde se muestra el flujo de trabajo del software implementado en las aeronaves

3.2. Implementación hardware

3.2.1. Sensores GNSS utilizados

En primer lugar, el elemento común tanto a la torre de control como al avión, son los sensores GNSS. En concreto son unos módulos de la casa Stemedu que permiten recibir información tanto del sistema GLONASS como del sistema GPS, pero en el pack de compra solo se incluye una antena capaz de recibir los sistemas BeiDou y GPS, por lo que el sistema de posicionamiento que utiliza es el GPS con su datum WGS84.

Es un módulo GNSS de unas dimensiones bastante reducidas (48 x 28 x 8,5 mm) que cuenta con conectividad USB. Esta es la forma que tiene de conectarse con el ordenador, lo que simplifica mucho su interoperabilidad entre dispositivos. En un sistema operativo Linux es casi instantáneo de configurar, simplemente se debe de contar con el paquete *gpsd-client* instalado y al ejecutar el comando *cgps*, se comienza a recibir información de forma instantánea. Para la obtención de datos en un *script* de Python se debe de contar con el paquete *python-gps*. De esta forma ya se pueden crear objetos gps de los que ir obteniendo información tan solo con ejecutar el *script* y realizar una operación *getattr*.

Figura 20: Imagen del modelo de los sensores y las antenas utilizados en el proyecto

3.2.2. Torre de control

Para la implementación de la torre de control se utiliza un ordenador portátil Lenovo ideapad 300 con 8Gb de RAM y un procesador Intel i5 de 6ª generación. A este ordenador se le añade un sensor GNSS como el mencionado en el anterior punto que le permite conocer su posición y realizar los cálculos de los errores para enviarlos a las aeronaves a través de su conexión TCP.

Es necesario que este ordenador tenga instalada la versión 3.6 del intérprete de Python, así como las librerías *pygame*, *python-utm* y *python-gps*.

3.2.3. Robot a escalas adoptando el papel de avión

A modo de modelo a escala de un avión, se ha diseñado un robot que está formado por una Raspberry Pi 3B+, un sensor GNSS como el utilizado en la torre de control, el módulo controlador de motores L2980N, dos motores eléctricos, una batería portátil y una estructura de plástico.

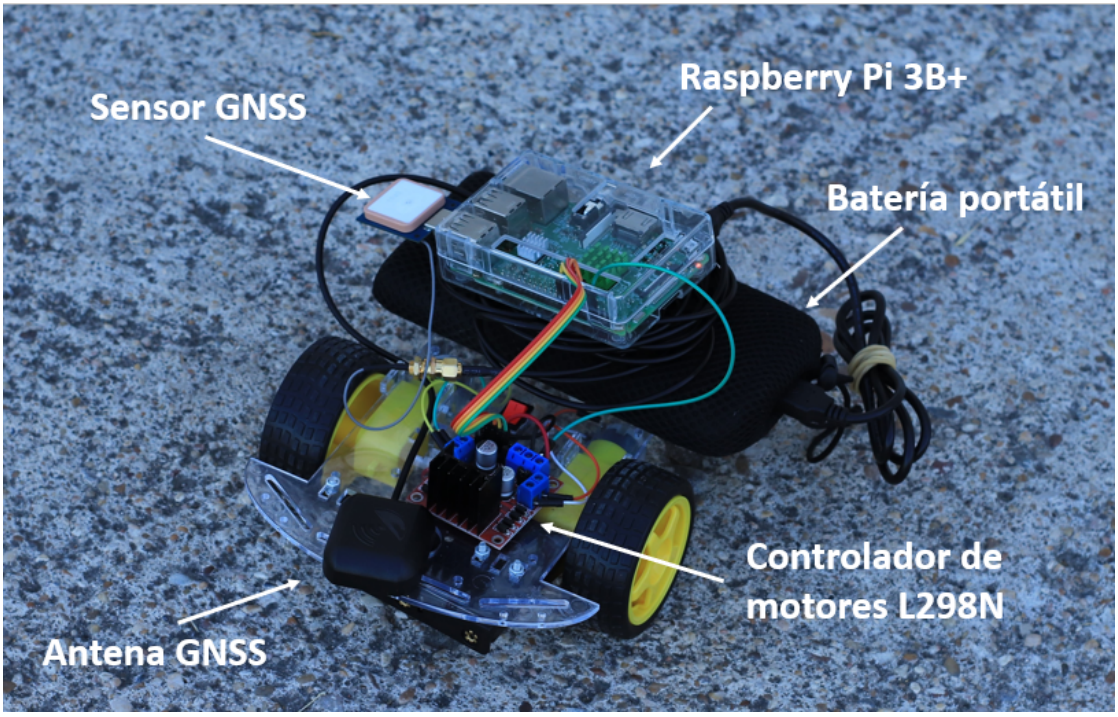

Figura 21: Imagen del robot diseñado para la realización de las pruebas del prototipo

De nuevo, es necesario que la Raspberry Pi tenga instalada la versión 3.6 del intérprete de Python, así como las librerías *pygame*, *python-utm* y *python-gps*.

Todos estos materiales nos permiten de una forma económica poder diseñar un prototipo con el que realizar pruebas para verificar la viabilidad o no, de escalar este sistema a un entorno de pruebas con aeronaves reales (pequeñas avionetas). Las especificaciones del hardware, así como de los costes asociados a todo el prototipo, pueden encontrarse detallados en los apéndices C y D respectivamente.

4. Pruebas realizadas

4.1. Objetivos

El objetivo que se pretendía alcanzar realizando diferentes pruebas era, en primer lugar, refinar las imperfecciones en el diseño del prototipo diseñado hasta alcanzar un prototipo con el que, finalmente, poder hacer una demostración funcional completa del mismo.

Una vez se alcanzó ese grado de madurez en el prototipo desarrollado, se ha buscado diseñar una prueba con la que poder comprobar y mostrar todas las funcionalidades del sistema de guiado en diferentes entornos y situaciones, para demostrar su capacidad de adaptación y versatilidad.

4.2. Exactitud de los sensores GNSS

En primer lugar, se necesitaba comprobar la exactitud de los sensores GNSS adquiridos para el prototipo. Si bien es cierto que la exactitud sin correcciones de ningún tipo del sistema GPS es de en torno a unos 14 metros (en un entorno despejado), se quería comprobar si el hecho de que fuesen unos sensores económicos (unos 18 € cada sensor) iba a afectar a la medida.

En las primeras pruebas, más enfocadas a la comprobación de la recepción de datos desde sensor en el ordenador, ya revelaron (en un entorno urbano) un error de unos 50 metros aproximadamente en la posición. Para poder comprobar esto, se llevó a cabo una prueba muy simple en la que se colocaba la antena GPS en el alféizar de la ventana y se ejecutaba el comando *cgps* perteneciente al paquete de Linux *gpsd-client*, con el que se muestra una tabla con toda la información ofrecida por el sensor (latitud, longitud, altitud y velocidad entre otras muchas, así como los satélites que tiene a la vista y que está utilizando para el cálculo de la posición)

Leyendo las posiciones obtenidas, representándolas sobre un mapa gracias a la herramienta Google Maps y comparándolas con la posición conocida de la ventana, se obtenía un error de unos 50 metros.

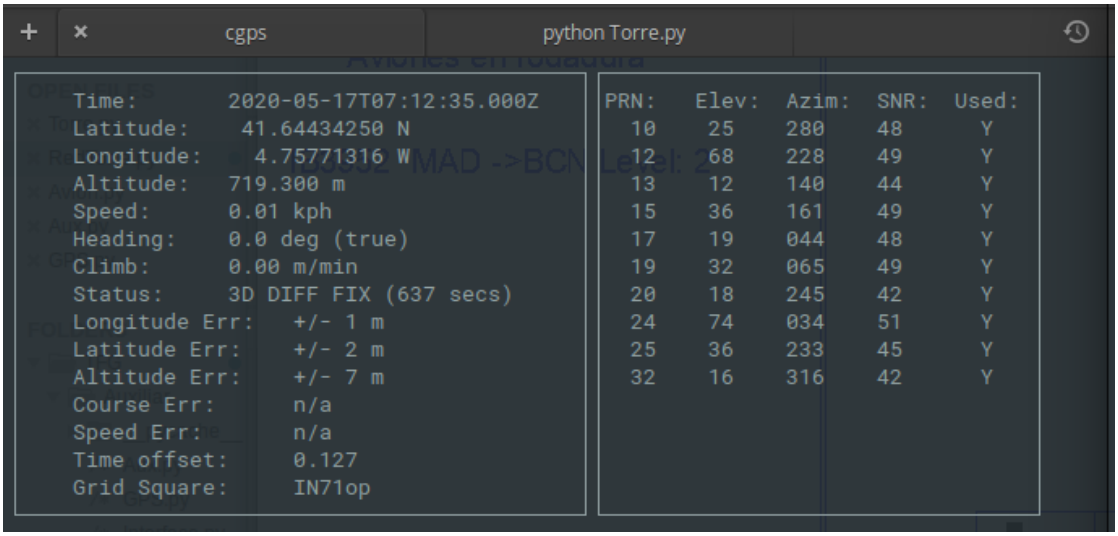

Figura 22: Información obtenida en tiempo real por el sensor GNSS utilizado

Dado que este primer experimento se realizó en un entorno urbano, se decidió repetir la prueba a un entorno mucho más despejado como es el aparcamiento del Centro Cultural Miguel Delibes donde poder contar con una mejor visibilidad de los satélites y un menor impacto del error multicamino provocado por los edificios circundantes.

En este caso, utilizando el mismo sistema, se obtuvo un error de unos 8 - 10 metros. Algo más aceptable, aunque no suficiente para una maqueta a escala de 8,5 x 6 metros, como se proponía en un inicio.

4.2.1. Calculo del error base del sensor y corrección de la posición combinándolo con un error relativo

En las primeras pruebas realizadas en un entorno abierto para comprobar la recepción de datos en el sensor GPS, se observó un fenómeno con el que no se contaba, y es que pese a trabajar con dos sensores, a priori, idénticos por ser el mismo modelo, de la misma marca y con la misma antena, si se ponía una antena en una posición, se registraban las coordenadas obtenidas y a continuación se cambiaba por la otra y se repetía el proceso, pese a dar un error de varios metros respecto de la posición verdadera, también daban un error de posición entre sí. Es decir, los sensores contaban con un error propio que era preciso corregir antes de hacer

ninguna corrección diferencial.

Este es un error más propio de un sensor de baja calidad, como los que se está utilizando, que de un sensor GNSS aplicado a la aeronáutica, pero en cualquier caso, los aviones no siempre van a utilizar el mismo fabricante y por tanto, tendrán pequeñas diferencias entre sí.

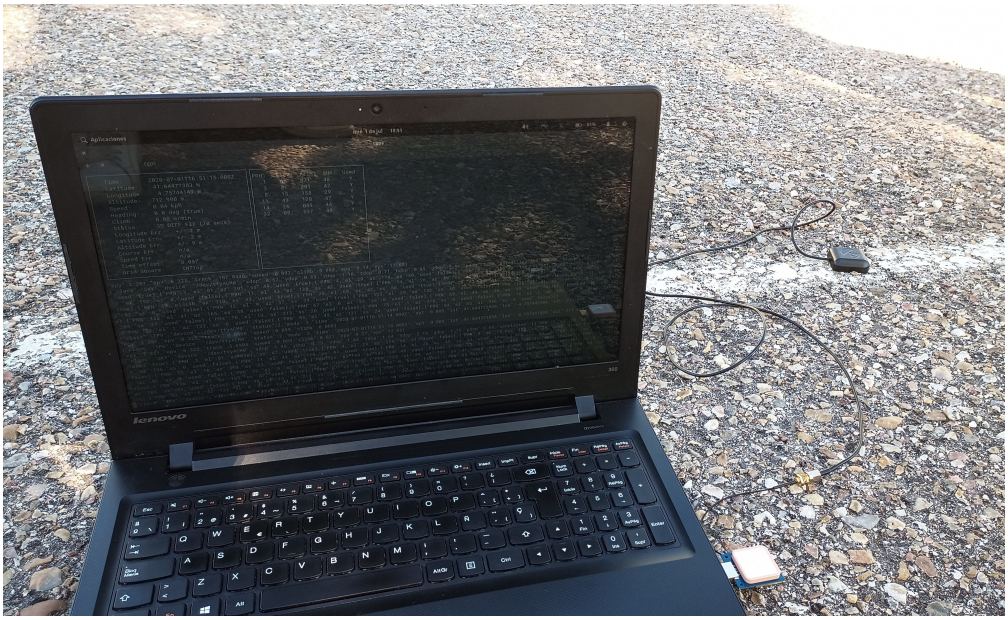

Figura 23: Imagen de la prueba en la que se percibió que había una excesiva diferencia entre la posición ofrecida por cada antena aun estando en la misma posición

Para abordar este problema, se decidió calibrar los sensores GPS antes de cada uso. Cuando el sistema se pone en marcha tanto en la torre de control como en la aeronave, se solicitan las coordenadas geográficas (datum WGS84) exactas y conocidas de la antena. A partir de ahí se realiza un promediado de las 10 primeras posiciones obtenidas por el sensor y a esta posición obtenida mediante promedio, se le resta la posición que se ha introducido como conocida y cierta (transformada por el programa a UTM). Se obtiene así, el error base del sensor. Este es un error denominado como error base del sensor, que se deberá de tener en cuenta en el caso de la torre para calcular el error relativo (el provocado por las fuentes de imprecisión ionosféricas y troposféricas), y en el caso de la aeronave, para realizar

el cálculo de la corrección de la posición al recibir el error relativo proveniente de la torre.

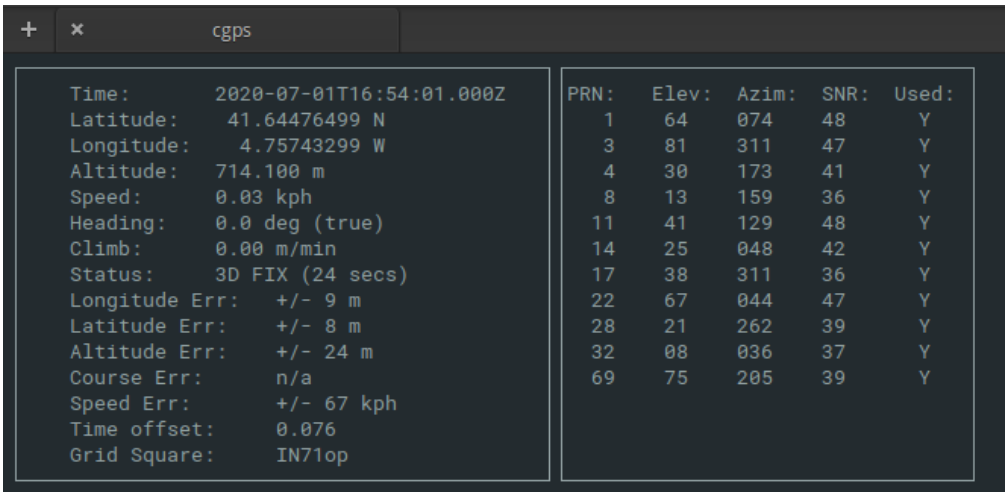

Figura 24: Captura de pantalla de las coordenadas calculadas con el primer sensor en la situación descrita en la figura 23

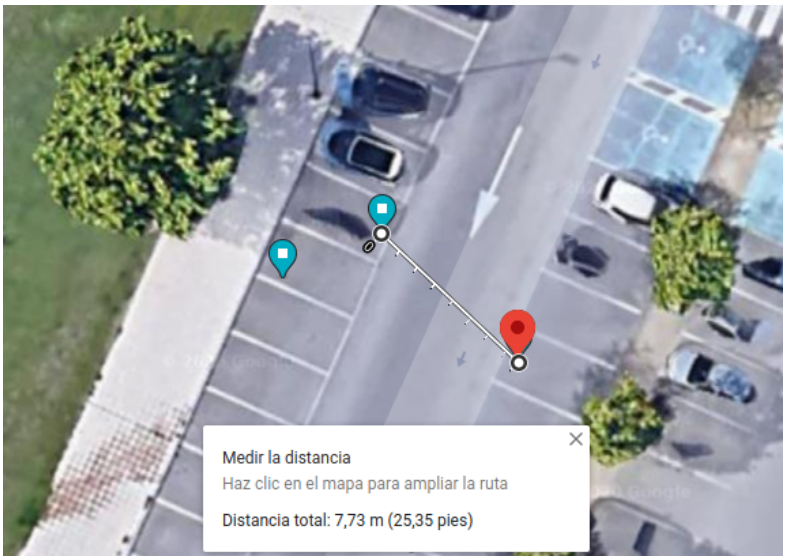

Figura 25: Captura de pantalla de la representación en google maps de coordenadas calculadas en la anterior figura. El punto rojo es la posición del ordenador y por tanto, la correcta. El error es de 7,73 metros. El segundo punto azul es la posición registrada por el segundo sensor.

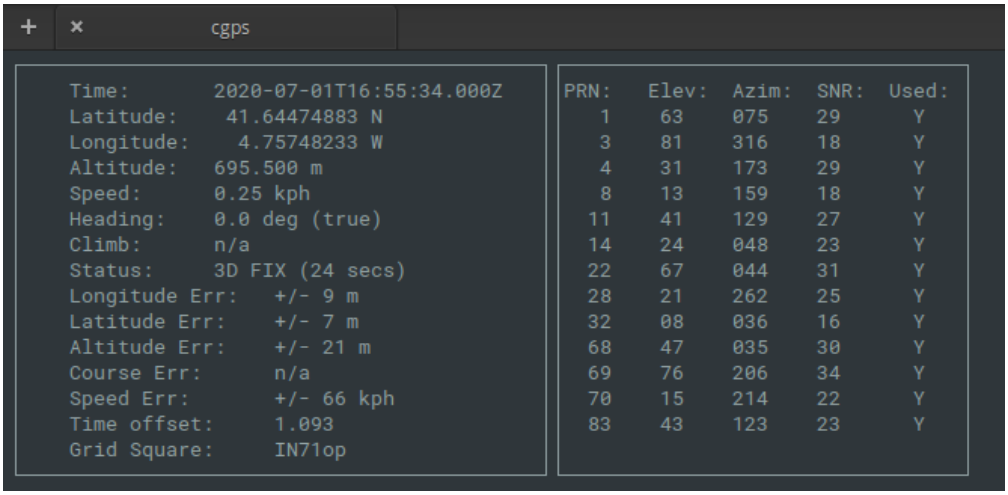

Figura 26: Captura de pantalla de las coordenadas calculadas con el segundo sensor en la situación descrita en la figura 23

Figura 27: Captura de pantalla de la representación en google maps de coordenadas calculadas en la anterior figura. El punto rojo es la posición del ordenador y por tanto, la correcta. El error es de 10,03 metros. El segundo punto azul es la posición registrada por el primer sensor.

Una vez se ha calculado este error base, se procede al cálculo del error relativo que como ya se ha mencionado, es el provocado por los efectos ionosféricos y

troposféricos. Este error se obtiene en la torre de control (ya que es el elemento fijo del prototipo) a partir de la posición que obtuvimos en un primer momento realizando el promediado de las 10 primeras posiciones obtenidas por el sensor. Esta posición es restada a la posición que va obteniendo el sensor GPS en cada iteración. Por tanto, este error relativo es actualizado y enviado a la aeronave en cada iteración del bucle.

Dado que es un error provocado por factores externos al sensor y que ambos sensores comparten por hallarse muy cerca el uno del otro (recordemos que las técnicas de GNSS diferencial eran válidas en un radio de unos 10/15 km desde la estación de corrección), se puede considerar que ambos sensores estarán calculando su posición bajo el mismo error.

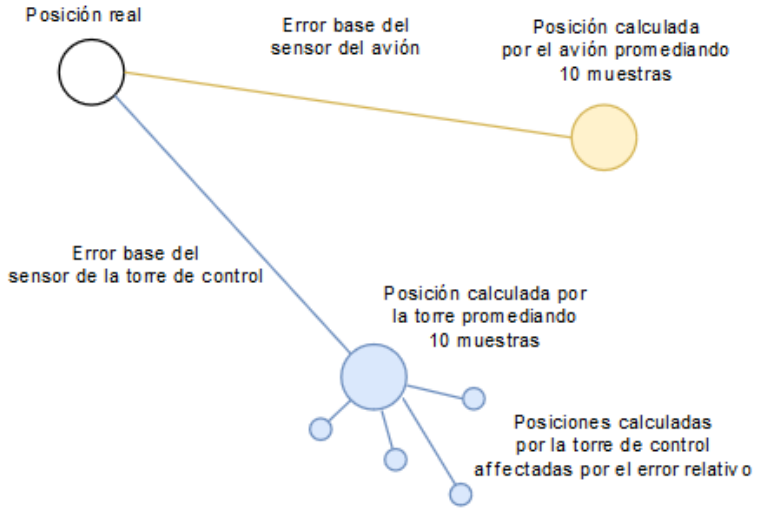

Figura 28: Esquema explicativo del método de corrección de la posición utilizado

Si se realiza la corrección de la posición con el error relativo en un sensor ya calibrado (ya se ha corregido el error base), deberíamos obtener la posición exacta de la aeronave. Esto no es así, pero sí que se consiguen posiciones bastante cercanas. El error obtenido con esta técnica de posicionamiento se halla entre 6 y 7 metros. Es un un error más aceptable para un sistema a escala real (8,3 x 5,4 km en el caso de Barajas con aviones de entre 30 y 70 metros de envergadura), pero bastante alto para una maqueta a escala.

Debido a esto, se decidió en este punto ampliar el tamaño inicial pensado para la maqueta sistema de 8,5 x 6 metros al tamaño final utilizado de 34 x 24 metros. Aun así, sigue siendo una maqueta muy reducida con respecto a la realidad donde, aunque los errores se han logrado reducir hasta imprecisiones en torno a 6 o 7 metros, siguen teniendo grandes variaciones entre una posición y la siguiente estando el móvil parado debido a las grandes variaciones que presentaba la posición calculada de una iteración a otra del bucle.

4.2.2. Implementación de filtro FIR paso bajo

Para solucionarlo se decidió implementar un filtro paso bajo que redujese esas grandes variaciones de la señal y solo mostrase los cambios reales y continuados cuando verdaderamente se moviese la aeronave.

Utilizando la aplicación *filterDesigner* disponible en MATLAB, se calculó un filtro FIR paso bajo siguiendo el diseño de mínimos cuadrados, ya que esta aproximación de diseño, daba la opción de indicar frecuencia de muestreo, frecuencia de corte y frecuencia de parada. Dado que el programa obtiene una posición GNSS cada 3 segundos aproximadamente, se fijó la frecuencia de muestro fs en 0,3333 Hz. La frecuencia de corte que se seleccionó fue de 0,14 Hz y la de parada de 0,15 Hz. Se fue variando el orden del filtro entre los órdenes 2, 4, 7 y 9 para obtener filtros compuestos por 3, 5, 8 y 10 coeficientes respectivamente. Al implementarlos en el código, se advirtió que alteraban la ganancia de la señal ya que la suma de sus coeficientes era distinta de 1. Lo que se decidió hacer para corregirlo, fue una aproximación de los coeficientes hasta obtener una suma total de 1 para que no aumentasen ni disminuyesen la posición al ponderar las diferentes muestras.

Para un filtro FIR paso bajo como el implementado en este prototipo, de orden k y k+1 muestras, su ecuación es:

$$y[n] = a_0 x[n] + a_1 x[n-1] + ... + a_k x[n-k] \tag{13}$$

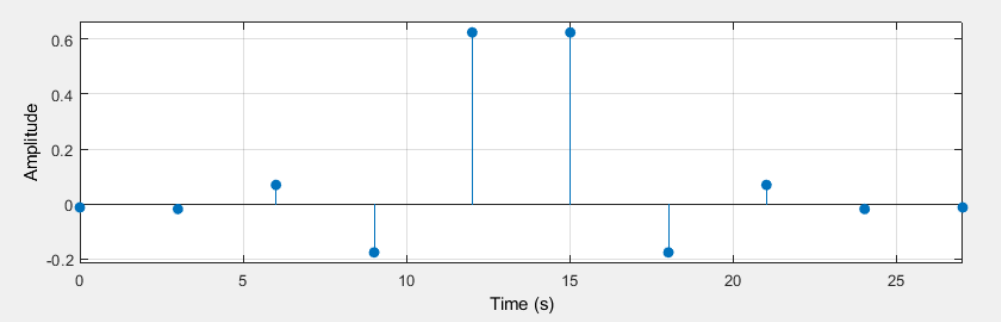

Figura 29: Ejemplo de respuesta al impulso en el dominio del tiempo del filtro FIR paso bajo de orden 9 utilizado en la torre de control

De esta forma la posición calculada era mucho más estable. Con órdenes de filtros altos (7 o 9), si el móvil no se movía, la posición obtenida era casi perfecta con un error de en torno a 50 cm, o 1 m en los peores casos, pero al iniciar el movimiento, el filtro introducía un retardo muy grande en la señal y aunque la posición seguía siendo muy exacta, tardaba demasiado en actualizarse la posición, algo que no era viable para esta aplicación.

Por eso, se decidió utilizar filtros de diferentes órdenes en la torre de control y en la aeronave. En la torre de control se utiliza un filtro de orden 9, ya que se desea que la señal sea lo más estable posible y no se va a mover nunca. En el avión, sin embargo, se utiliza un filtro con orden 2, que introduce mucho menos retardo y aunque la posición está afectada de un mayor error (unos 3 metros una vez se inicia el movimiento) y se nota una pequeña deriva del movimiento debido a las limitaciones del oscilador local del sensor, que introducen errores de reloj, sigue ofreciendo una posición bastante fiable que permite llevar a cabo las pruebas satisfactoriamente.

En el apéndice B.1 se pueden encontrar los valores de los coeficientes utilizados por el filtro FIR en cada uno de los casos, así como una representación gráfica tanto en tiempo como en frecuencia de las diferentes respuestas al impulso que posee cada uno de los filtros.

4.3. Creación, cálculo y representación de una ruta

Otro de los puntos clave del proyecto es el seguimiento de forma autónoma de una ruta por el aeropuerto. Para ello, se ha diseñado un sistema de cálculo de rutas que en el caso del avión, no necesita conocer el mapa del aeropuerto. Simplemente necesita conocer una serie de coordenadas que le serán enviadas desde la torre de control junto con el resto de información necesaria para el funcionamiento del sistema en el momento de solicitar el inicio de la maniobra de rodadura.

Quien sí necesita conocer el mapa del aeropuerto es la torre de control, ya que será la encargada de crear las rutas para cada uno de los aviones. Se ha simplificado el sistema de coordenadas del mapa del aeropuerto, en el que cada punto de referencia o intersección en el aeropuerto es denotado por un par de letras que representan la latitud y la longitud del mismo. De esta forma se simplifica el trabajo al controlador aéreo que cree la ruta ya que no debe conocer las coordenadas exactas, solo el par de letras latitud, longitud que lo identifican. En el caso de la latitud, puede tomar un valor cualquiera desde la A, hasta la O (de arriba a abajo) y en el caso de la longitud, desde la A hasta la K (de izquierda a derecha). El controlador aéreo tendrá que ir indicando pares de letras (latitud,longitud) que determinarán una serie de puntos en el mapa, es decir, la ruta que deberá de seguir el avión por el aeropuerto. Una vez haya terminado, se traducirán los pares de letras a coordenadas UTM de la forma que se indica en el apartado "Creación de rutas" del apéndice matemático (B.2) y se enviarán al avión que esté asociado a la conexión donde se haya creado la ruta.

Cuando el avión reciba esas coordenadas, se almacenarán en el atributo del objeto Avión *route* en forma de listas (cada pareja de coordenadas es una tupla). Las coordenadas representan puntos en el mapa entre los que se tiene que desplazar la aeronave. Para ello, se calculan, siguiendo los principios del álgebra lineal tal y como es descrito en el apéndice matemático B.2, las ecuaciones de las rectas que unen los puntos de coordenadas siguiendo el orden en el que fueron enviados. De esta forma, quedarán unidos mediante rectas el primer punto con el segundo, el segundo con el tercero, etc... Este conjunto de rectas será la ruta que seguirán los aviones para moverse de un punto a otro del aeropuerto.

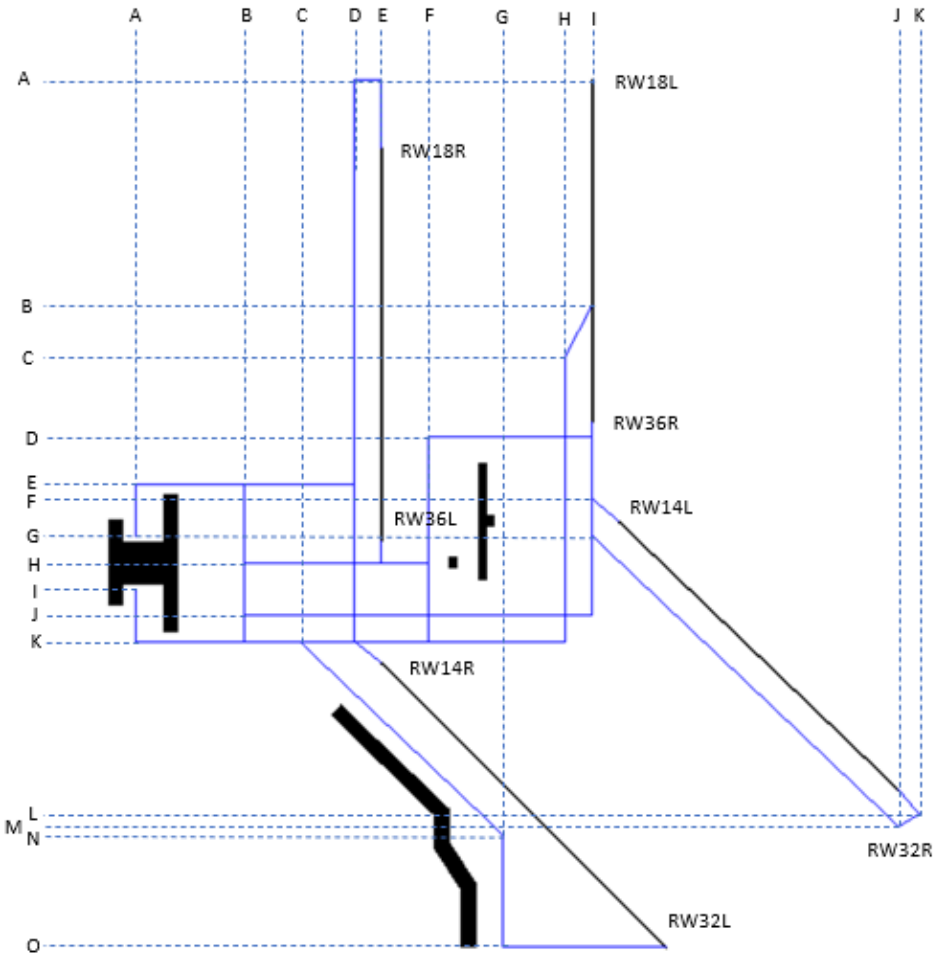

Figura 30: Mapa con los puntos del sistema de coordenadas diseñado para este aeropuerto referenciados a los laterales

Debido a esta forma de enfocar la solución para el guiado de la aeronave, para iniciar el sistema se deberá de alinear el avión con la pista de rodaje en la que se inicia la maniobra. Eso evita que el avión tenga que alinearse por si mismo simplificando mucho el inicio del sistema.

Aunque en el momento de iniciar la maniobra de rodaje se alinee la aeronave con la pista, de forma realista, es prácticamente imposible que un móvil se mueva completamente recto, o que la alineación sea perfecta, y más cuando se trata de rectas en un entorno real que rozarían las distancias de cientos de metros o incluso el kilómetro. Por ello, aplicando de nuevo el álgebra lineal tal y como se describe

en el apéndice matemático B.2, por cada recta calculada entre puntos, se calculan dos rectas paralelas, una a cada lado, que determinan una tolerancia de 4 metros por cada lado entorno al centro de la recta. Si el sistema detecta que el avión se está desviando de la ruta, corrige automáticamente el rumbo a izquierda o derecha para recentrar la aeronave.

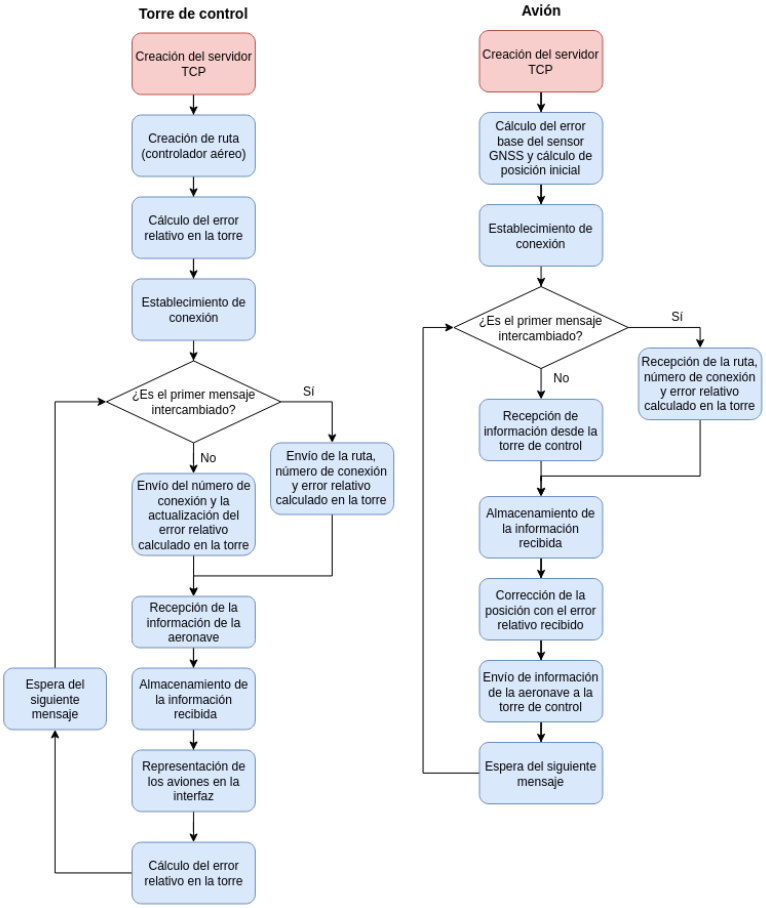

Figura 31: Diagrama de bloques donde se muestra el flujo de trabajo del software implicado en el intercambio de información entre la torre y una aeronave

Estas dos rectas, así como la recta central determinada por la ecuación entre cada uno de los puntos de la ruta, se envían a la torre de control, para que puedan ser representadas en la interfaz y permitan ver al controlador aéreo si cada uno de los aviones ha calculado bien su ruta y si la está siguiendo dentro de las tolerancias.

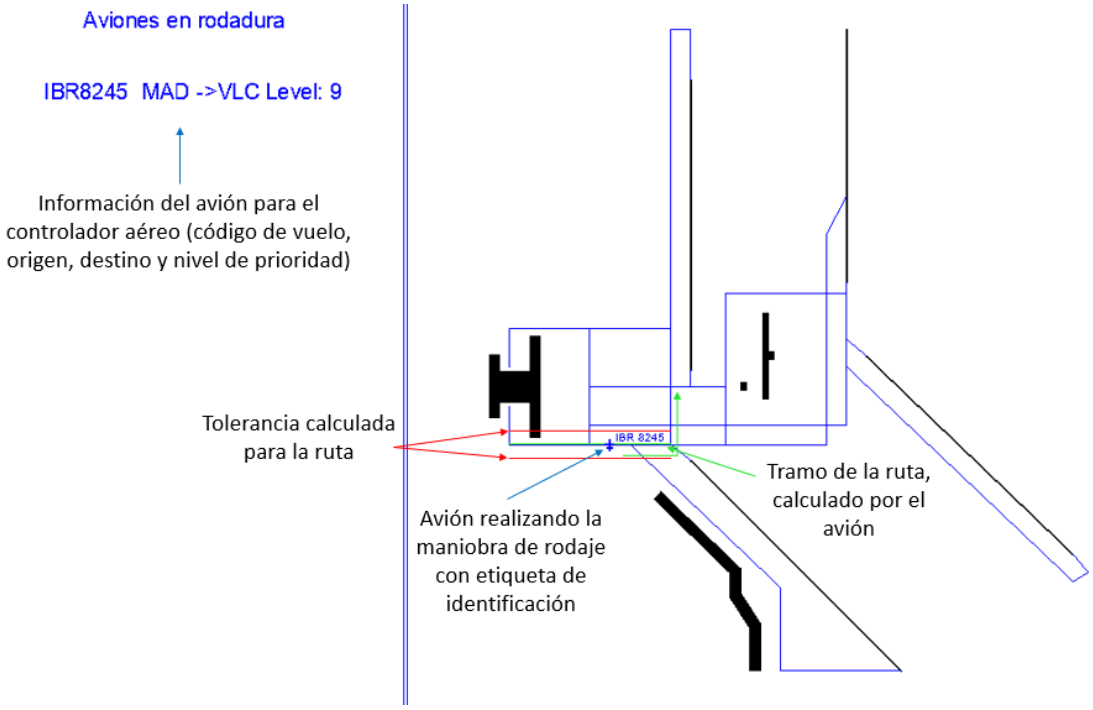

Figura 32: Interfaz del sistema con un avión realizando la maniobra de rodaje y sus rectas calculadas representadas en verde la recta principal y en rojo las tolerancias

El movimiento de la aeronave es un proceso que sucede de forma cíclica dentro de un bucle con una estructura muy sencilla. Primero se calcula la ecuación de la recta entre el primer par de puntos y si no hay aviones en las cercanías (como se explica en el próximo apartado con las pruebas de prevención de colisiones), se avanza. Después de un segundo se comprueba si se ha llegado al punto deseado (con una tolerancia de 1 metro) y si es así, se calcula la nueva recta entre los dos siguientes puntos de la ruta y el giro a izquierda o derecha que se debe de hacer para alinear el avión con el siguiente tramo de la ruta. Si el ángulo de giro es menor de 20^{o}, no se realinea el avión y se deja corregir la alineación con la pista de rodaje mediante tolerancias calculadas. Si no se ha llegado al punto deseado, se vuelve al punto en el que se comprueba si hay alguna aeronave en las inmediaciones y se continua avanzando.

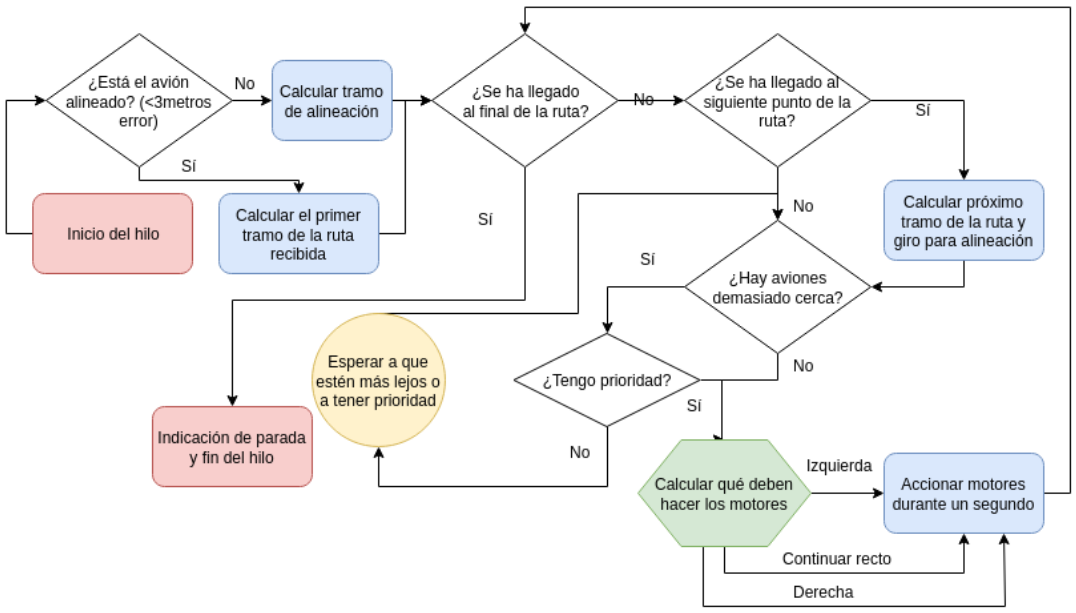

Figura 33: Diagrama de bloques donde se muestra el flujo de trabajo del hilo encargado del cálculo de rutas y movimiento en la maniobra de rodaje

4.3.1. Prueba en un entorno simulado

En un primer momento, y con el objetivo de poder comprobar que los algoritmos de creación, cálculo y representación de rutas funcionaban correctamente, se diseñó un entorno virtual en el que poder realizar estas operaciones con coordenadas perfectas, sin alteraciones debido a los errores de posicionamiento introducidos por el sensor GNSS.

Este entorno virtual era muy similar al programa real desarrollado, ya que seguía el mismo flujo de trabajo, el código correspondiente al avión se ejecutaba en un ordenador aparte y el código era prácticamente el mismo, pero obviaba los momentos en los que debían de recibir posiciones GNSS y corregir sus errores.

En su lugar, cuando en la aeronave recibía la ruta creada por la torre de control y se calculaban las rectas que le permitirían seguir dicha ruta, en vez de fijarse en las posiciones que recibía del sensor GNSS para saber si se estaba siguiendo la ruta o no, se hacía una llamada al método *simtaxiing()* de la clase *Plane*. Este método, calculaba la posición siguiente, en base a su posición anterior, una variable

llamada *vel* que simbolizaba la distancia que avanzaba en cada iteración del bucle y la ecuación de la recta que estaba siguiendo.

Una vez se había actualizado la posición y se habían hecho los cálculos para determinar si se había llegado o no al punto deseado, o si se encontraba demasiado cerca de otra aeronave, el avión enviaba mediante su cliente TCP toda su información a la torre de control, quien se encargaba de representar en la interfaz la recta calculada y la información del avión igual que en el entorno real.

En realidad, los cambios realizados para adaptar el sistema a un entorno virtual, solo afectaban al código de las aeronaves y para la torre de control era un proceso completamente transparente. Hacía las mismas tareas en el entorno real que en el virtual.

Dado que las pruebas tras varios ajustes y retoques en el código, resultaron un éxito al poder manejar incluso varias conexiones de forma simultánea representando las rectas de cada una de las rutas, así como a las aeronaves en sus caminos por el aeropuerto, se pudo concluir que los algoritmos de creación, generación y representación de rutas funcionaban sin problema y se podía pasar a realizar pruebas en un entorno real, con el robot a escala diseñado.

4.3.2. Prueba en un entorno real

Las pruebas planteadas en este caso fueron dos, una primera prueba en la que el objetivo era mover el avión robot de un punto a otro a lo largo de una línea recta sin desviarse más allá de la tolerancia establecida de 4 metros por cada lado, y una segunda prueba donde se persigue el mismo objetivo, pero al llegar al primer punto de la ruta se continuaba la ruta hacia un segundo punto que se hallaba en ángulo recto respecto del rumbo seguido hasta el momento (es el giro más común en una pista de rodaje, 90 grados a un lado o al otro).

Al llevar a cabo la primera prueba, se observó que la exactitud del sensor GNSS no era suficiente como para poder hacer esta operación con un móvil de 16 x 22 cm. El móvil era demasiado pequeño y ante los cambios repentinos del sensor GNSS (la posición tenía una oscilación de unos +/- 6 o 7 metros) y era muy difícil que el robot consiguiese seguir una línea más o menos recta por mucho tiempo.

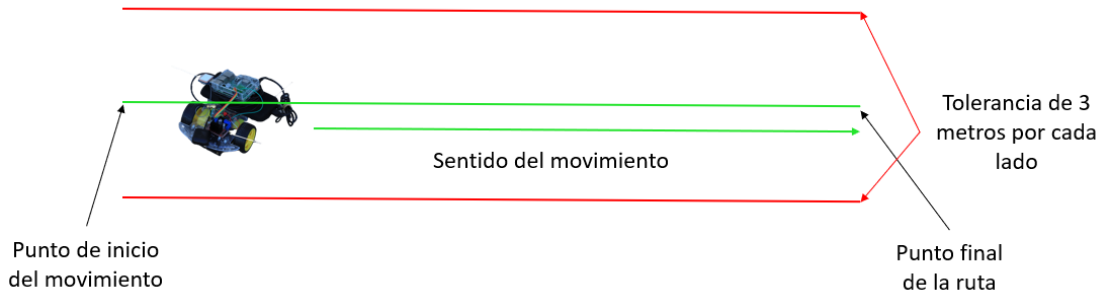

Figura 34: Esquema del movimiento a realizar por el robot en la primera prueba en un entorno real

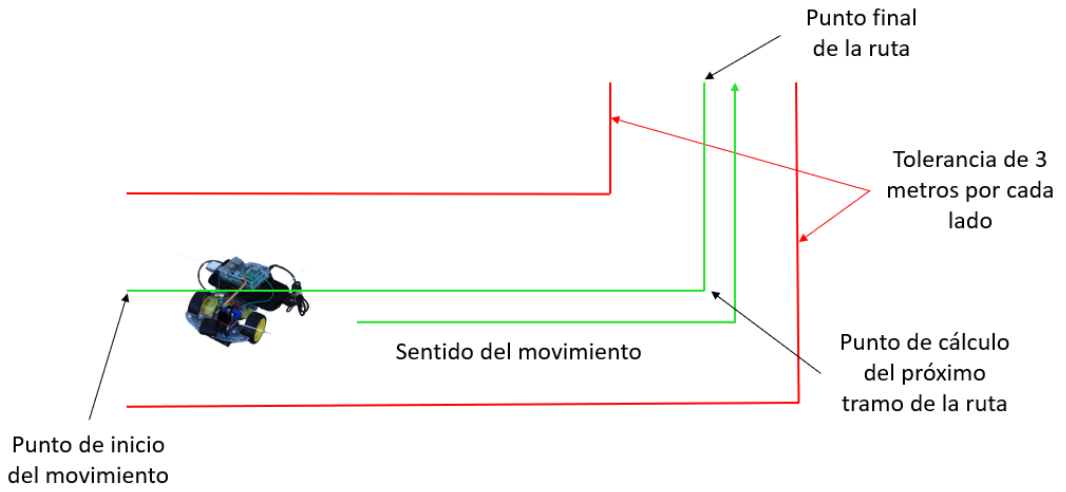

Figura 35: Esquema del movimiento a realizar por el robot en la segunda prueba en un entorno real. Incluye un giro y el cálculo de la ecuación que une los dos próximos puntos de la ruta

Para mejorar este resultado, se suavizaron las variaciones de la señal GNSS aplicando un filtro FIR paso bajo de orden 9 a las posiciones calculadas por la torre de control y un filtro FIR paso bajo de orden 3 a las posiciones calculadas por el avión.

Tras varios intentos buscando unas condiciones meteorológicas que favoreciesen la recepción de la señal GNSS y acortando las distancias entre los puntos de la ruta, se obtuvo un resultado aceptable teniendo en cuenta la calidad de los sensores utilizados en el prototipo, cumpliendo con los objetivos de las dos pruebas propuestas

en un inicio.

Figura 36: Imagen del robot diseñado entre unas líneas que simulan las líneas de tolerancia a 3 metros por cada lado calculadas por el programa

Gracias a las pruebas realizadas tanto en el entorno virtual, como en el real, se pudo determinar que el algoritmo desarrollado funcionaba y que podría ser viable seguir desarrollándolo realizando una mayor inversión en la tecnología de posicionamiento, que es el factor limitante de este prototipo.

4.4. Prevención de colisiones

Uno de los aspectos más importantes de este proyecto es la capacidad de las aeronaves para detectar a otros aviones en las proximidades y la decisión que tomen en función de la situación y de la aeronave con la que se vaya a encontrar. Para poder hacer esto, antes de enviar ninguna orden de movimiento (como izquierda, derecha o hacia delante) a los motores, se comprueba el valor del primer elemento del atributo del objeto Avión (creado a partir de la clase *Plane*), *route*. Si este atributo vale 0, todo está en orden y se puede continuar, si por el contrario, vale -1, se debe de parar inmediatamente hasta que vuelva a valer 0.

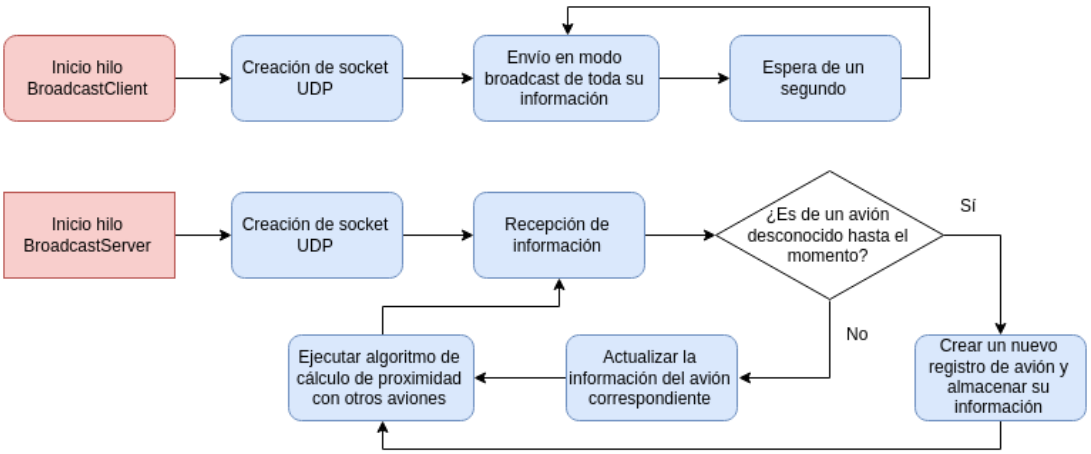

Figura 37: Diagrama de bloques donde se muestra el flujo de trabajo de los hilos encargados de los sockets UDP, *BroadcastServer* y *BroadcastClient*.

Este valor es actualizado periódicamente por el hilo *BroadcastServer* cada vez que se recibe información de alguna aeronave que se halle en la misma red inalámbrica y envía su información a través del hilo *BroadcastClient*. Cuando se recibe información de un avión, se almacena y se invoca al método de la clase *Plane*, *checkproximity()*.Este método se encarga de calcular la distancia tanto en latitud como en longitud entre sí mismo y el resto de aeronaves de las que se ha recibido información. Si esta distancia es inferior a 3 metros (en la realidad debería de aumentarse unas 10 veces, hasta 30 metros por lo menos), se deberá de tomar una decisión, seguir avanzando, o detenerse. Esta decisión se basa en el nivel de prioridad que posea cada aeronave (este es un valor que deberá ser indicado desde torre de control antes de comenzar el rodaje y se encuentra en el atributo *level* de la clase *Plane*). La aeronave que posea un nivel de prioridad superior, continuará, y la otra deberá detenerse hasta que se cumpla la distancia mínima de separación (3 metros). En el hipotético caso de que ambas aeronaves tuviesen el mismo nivel de prioridad, la decisión se tomaría en base al número de conexión más bajo. La aeronave con el número de conexión más bajo, llevará más tiempo en el sistema y por tanto, tendrá prioridad para continuar con el rodaje.

```
162     def checkproximity(self, Aviones):
163         """
164         En este método se calcula la distancia entre la aeronave y el resto de aeronaves en la red.
165         En función de la distancia calculada se tomará una decisión u otra.
166         Esto se verá reflejado a través del parámetro self.route[0]. Si adopta un valor igual a -1,
167         la aeronave deberá detenerse. Si adopta un valor igual a 0, la aeronave podrá continuar
168         con la maniobra de rodaje
169         """
170 
171         if len(Aviones) != 1:
172             for i in range(0,len(Aviones)-1):
173                 if abs(self.lat-Aviones[i].lat)<0.8 and abs(self.lon-Aviones[i].lon)<0.8:
174                     print('Situacion de bloqueo')
175                     self.route[0]=-1
176                     return
177                 elif abs(self.lat-Aviones[i].lat)<1.5 and abs(self.lon-Aviones[i].lon)<1.5:
178                     self.route[0]=-1
179                     if self.level<Aviones[i].level:
180                         self.route[0]=0
181                     elif self.level == Aviones[i].level and self.n < Aviones[i].n:
182                         self.route[0]=0
183                 elif abs(self.lat-Aviones[i].lat)<3 and abs(self.lon-Aviones[i].lon)<3:
184                     if self.level < Aviones[i].level:
185                         self.route[0]=-1
186                         return
187                     elif (self.level == Aviones[i].level) and self.n > Aviones[i].n:
188                         self.route[0]=-1
189                         return
190 
191         self.route[0]=0
192         return
```

Figura 38: Código creado para calcular la distancia al resto de aeronaves y tomar una decisión en función de la misma

En el caso de que las dos aeronaves estuviesen una a continuación de la otra en la misma pista de rodadura o en un cruce de pistas, aunque una se parase, la otra podría arrollar a la anterior, si la que se encuentra obstruyendo el paso es la aeronave con menor prioridad. Por eso, se ha diseñado el algoritmo de tal forma que si la aeronave con mayor prioridad sigue avanzando y se llega a una distancia mínima de seguridad de 1,5 metros, se detenga igualmente y se de paso a la aeronave con menor prioridad para que avance y desbloquee el paso.

Igualmente, podría darse el caso de que ambas aeronaves, por un error del controlador aéreo fuesen enviadas por la misma pista de rodadura en direcciones contrarias. En este caso, se ha diseñado el algoritmo de tal forma que si se detecta una distancia crítica de 0,8 metros se detienen ambas aeronaves por completo y muestran un mensaje de “Situación de bloqueo”. Se trata de 0,8 metros ya que el sistema primero piensa que que se halla en la situación anterior y por tanto, da paso al avión de menor prioridad para que desbloquee el paso. Al avanzar el avión con menor prioridad, como se encuentran en la misma pista, pero en direcciones opuestas, la distancia en vez de aumentar, disminuye aún más.

Para poder comprobar la efectividad del sistema teniendo en cuenta las variaciones en la posición del sensor GNSS, se ha diseñado una prueba en el mismo entorno virtual del anterior punto en la que se utilizan dos aviones "virtuales" que obtienen sus posiciones de la misma forma en que lo hacía el simulador de movimiento, siguiendo las rectas previamente calculadas por el objeto Avión.

4.4.1. Identificar la prioridad y ceder el paso

En la primera prueba, el avión que se ejecutaba en la Raspberry Pi (por simplificar, avión 1) tenía prioridad sobre el avión ejecutado en un tercer ordenador utilizado para esta prueba (avión 2). El avión 1 tenía una ruta definida por la torre de control y en su camino se debía de encontrar con el avión 2, que estaría parado en una intersección durante 30 segundos, a partir de los cuales arrancaba siguiendo su propia ruta liberando así la pista de rodaje.

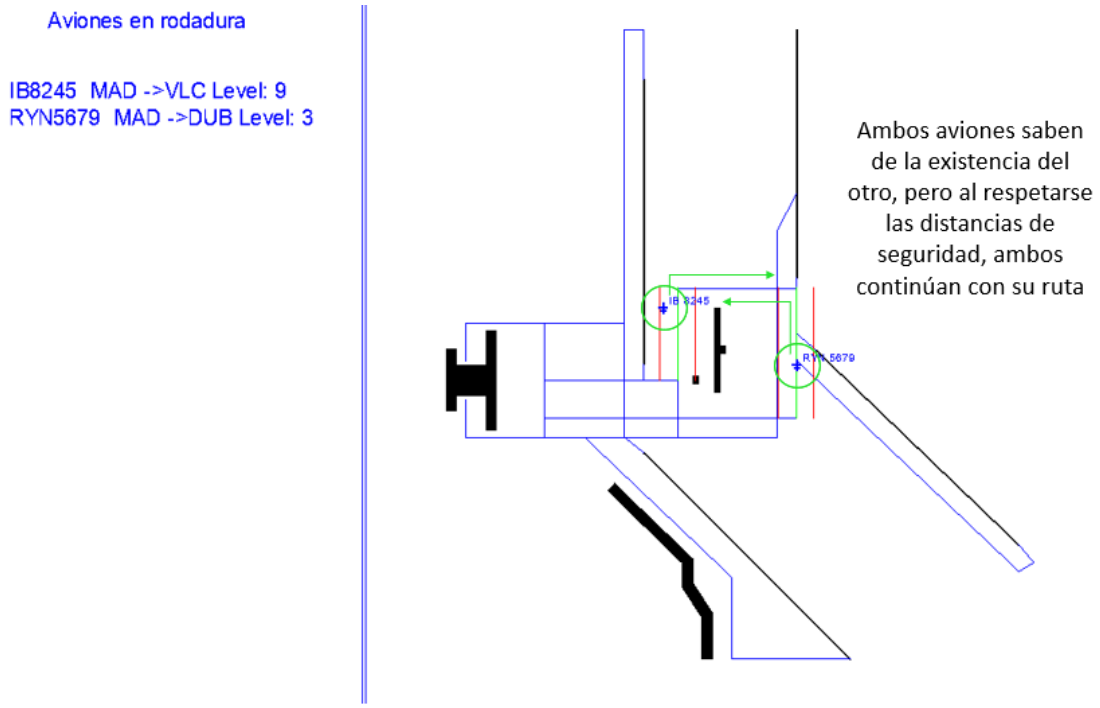

Figura 39: Situación provocada en la que dos aviones realizan la maniobra de rodaje de forma simultánea, pero no necesitan ceder la prioridad por existir suficiente distancia

El objetivo era que el avión 1 identificase que había un segundo avión en su camino y que tenía prioridad sobre él. Cuando siguiese avanzando y viese que aunque tenía prioridad, el otro avión no se movía y por tanto seguía recortándole distancia, se debía de detener por completo al pasar la barrera de 1,5 metros de distancia mínima de seguridad y esperar a que el avión 2 se moviese.

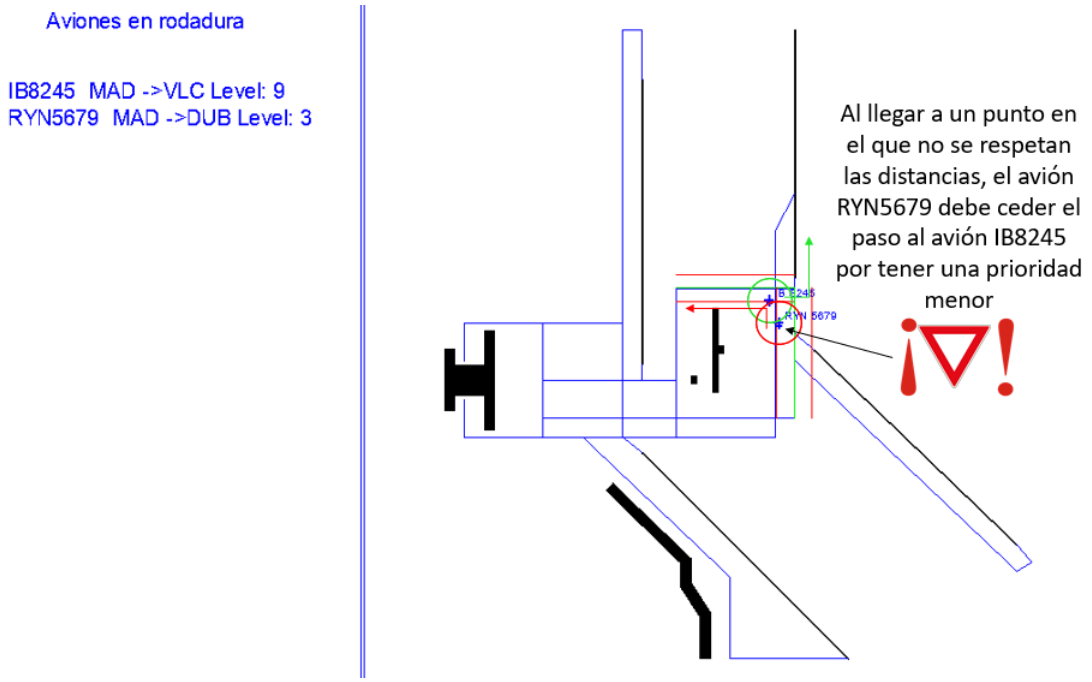

Figura 40: Situación provocada en la que dos aviones realizan la maniobra de rodaje de forma simultánea y debido a lo próximos que están, el avión de menor prioridad (RYN 5679) debe ceder el paso

El resultado de la prueba tras varios intentos de ensayo y error haciendo ajustes en el código, fue satisfactorio. Ambas aeronaves detectaron que había un segundo avión en el momento en el que se conectaba la segunda aeronave a la red, fueron calculando las distancias segundo a segundo, a medida que iban actualizando sus propias coordenadas y llegado el momento, seguían el protocolo marcado por el algoritmo. La primera aeronave en un primer momento prosiguió, detectando que tenía una mayor prioridad, pero al ver que la segunda aeronave no se movía, se detuvo y esperó a que esta liberase la pista.

En la segunda prueba, los roles se invirtieron y ahora era el avión 2 quien tenía

prioridad. En este caso el avión 2 se había programado de tal forma que cruzase la trayectoria del avión 1, por lo que este último, debería de detenerse, esperar a que pasase el robot 2 y una vez se cumpliese la distancia de seguridad de 3 metros, reanudar su marcha.

El sistema de control volvió a comportarse satisfactoriamente ante las condiciones planteadas, ambas aeronaves supieron interpretar la situación y no chocaron. Pero planteando esta prueba, surgió la duda de qué debería suceder en el caso de suceder un error humano por parte del controlador aéreo, que enviase a dos aeronaves por la misma pista de rodadura pero en sentidos opuestos.

4.4.2. Situación de bloqueo en la pista de rodaje

Si llegase a suceder la situación expuesta al final de la anterior subsección, según se había desarrollado el algoritmo hasta ese momento, una aeronave proseguiría con su camino con la intención de desbloquear la pisa y chocaría frontalmente con la otra.

Por eso se decidió incluir el último caso posible en este algoritmo: la distancia crítica de seguridad. Esta es una distancia ya muy reducida, en la que una aeronave ha decidido avanzar, pero se encuentra que al avanzar está incluso más cerca de la otra aeronave de lo que ya estaba en un principio. En ese caso, el sistema para inmediatamente ambas aeronaves y muestra un mensaje de "Sistema bloqueado". En ese momento, en una situación real, la tripulación debería contactar con torre de control y esperar instrucciones por parte del controlador aéreo.

Para comprobar este último caso, se planteó un tercer escenario, en el que se envió a las dos aeronaves por la pista que une los puntos D,I, D,F y H,F en sentidos contrarios. El resultado, tras un par de ajustes en el código, fue que ambos aviones se paraban y detectaban que se encontraban uno frente a otro sin poder avanzar, mostraron el mensaje de "Sistema bloqueado" por la consola de comandos y se detuvieron.

4.5. Capacidad del sistema

Cuando se despliega una aplicación informática nueva o se integra un sistema como este dentro de una infraestructura ya consolidada como es la de un aeropuerto, una de las cosas que más se deben de tener en cuenta es la cantidad de recursos que se utilizan. Esto determinará en gran medida la viabilidad y escalabilidad del proyecto a largo plazo y el interés que mostrarán los posibles inversores o promotores en la idea propuesta. Por eso, es preciso realizar una serie de pruebas mediante las que observar la carga computacional que supone la solución desarrollada para el equipo informático que lo albergue.

En este caso, se ha decidido observar el impacto del programa desarrollado mediante la aplicación que provee la familia de sistemas operativos Linux para monitorización y administración de procesos, *htop* (mostrada en la figura 41). Con esta herramienta se puede observar el porcentaje de uso de las CPUs del sistema, así como la cantidad de memoria RAM y memoria *swap* que se está utilizando. Además, proporciona un listado de procesos para los que, de forma individualizada, muestra su PID, el usuario que los ha puesto en marcha y el tiempo que llevan en activo, entre otros. Por último, ofrece una visión global del estado del sistema operativo mostrando la cantidad de tareas que hay esperando a ser procesadas por alguna de las CPUs del equipo, el número de hilos o *threads* que se encuentran en marcha y la cantidad de procesos que se están llevando a cabo de forma simultánea en el sistema.

Esta prueba, se ha diseñado en cuatro fases. Una primera donde se monitoriza el funcionamiento y exigencias a las que se ve sometido el ordenador en un estado usual, sin que se esté ejecutando ningún programa a parte de los básicos y esenciales para mantener el ordenador en funcionamiento (figura 41). Una segunda fase en la que se inicia el programa encargado de asumir el rol de torre de control, con todo lo que ello implica como corrección de errores, monitorización de aviones, posicionamiento en la interfaz gráfica, etc... (figura 42). Una tercera fase en la que se incorpora un único avión al sistema (figura 43) y una última, en la que se intenta incorporar el máximo número de aviones al sistema (figura 44). Este número máximo de aviones, solamente pudieron ser tres, debido a limitaciones de

material ya que solo se admite un avión por dirección IP en el sistema y solo se disponía de tres ordenadores.

Para medir el impacto de cada una de las fases respecto a la anterior, se tomarán como valores reseñables el porcentaje de uso de las CPUs y de la RAM del equipo.

Se presentan las diferentes imágenes de los resultados obtenidos.

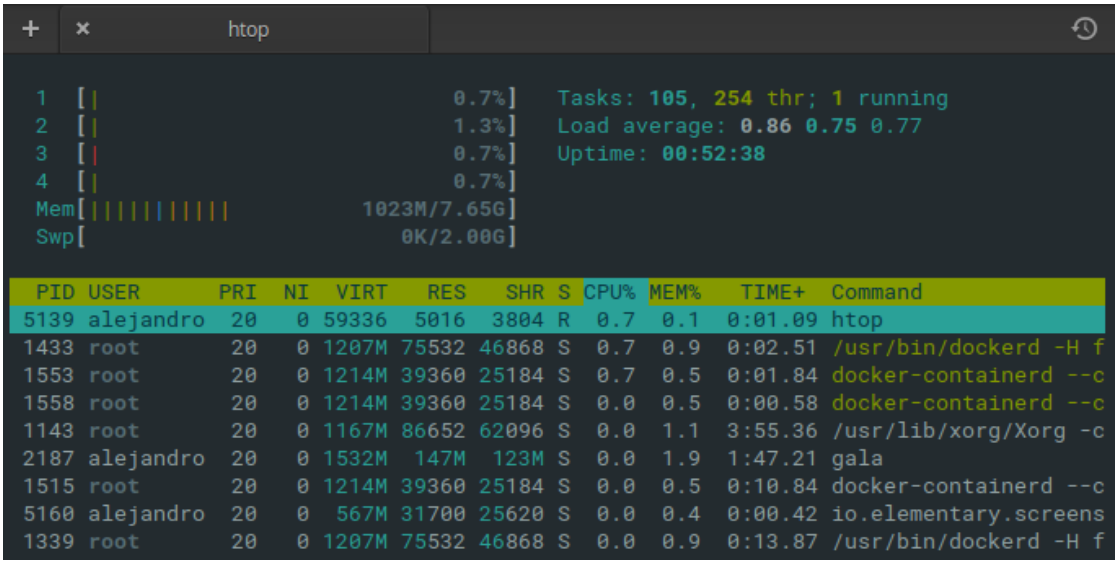

Figura 41: Situación registrada por htop cuando no hay ningún proceso ejecutándose en el ordenador aparte de los mínimos e indispensables

En un principio y sin iniciar el sistema, se observa como el consumo normal de recursos del ordenador no supera el Gb de RAM y el uso de las CPUs es muy bajo, de en torno al 1 %. Lo que se ha podido observar ha sido que desde el momento en el que se inicia el programa de la torre de control y se se comienzan a crear los hilos responsables de mantener una comunicación con los aviones, el sistema comienza a operar con 2 CPUs a tasas cercanas al 100 %. Esto es debido a los dos hilos existentes, el hilo principal de la torre de control y el primer servidor TCP disponible en el puerto 65000.

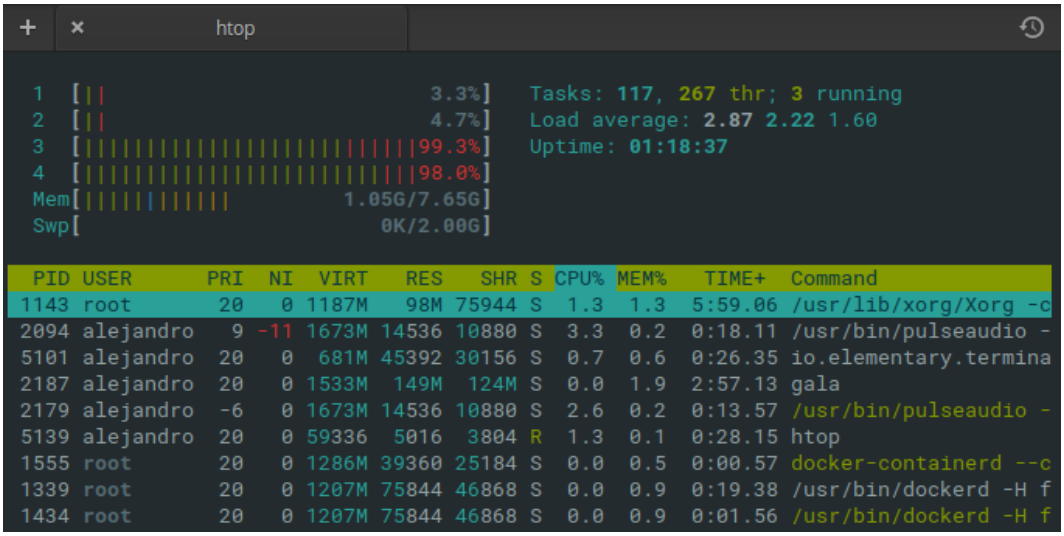

Figura 42: Situación registrada por htop cuando solo se está ejecutando Torre.py y no hay ningún avión conectado

A continuación, si se inicia una conexión con un avión, lo que se observa es que se añade un hilo más y ya son 3 las CPUs con un alto porcentaje de uso, pero solo una se mantiene al 100 % y de las otras dos, una baja hasta valores en torno al 70 % - 75 % y la otra tiene un consumo bastante bajo, en torno al 30 % o 35 %.

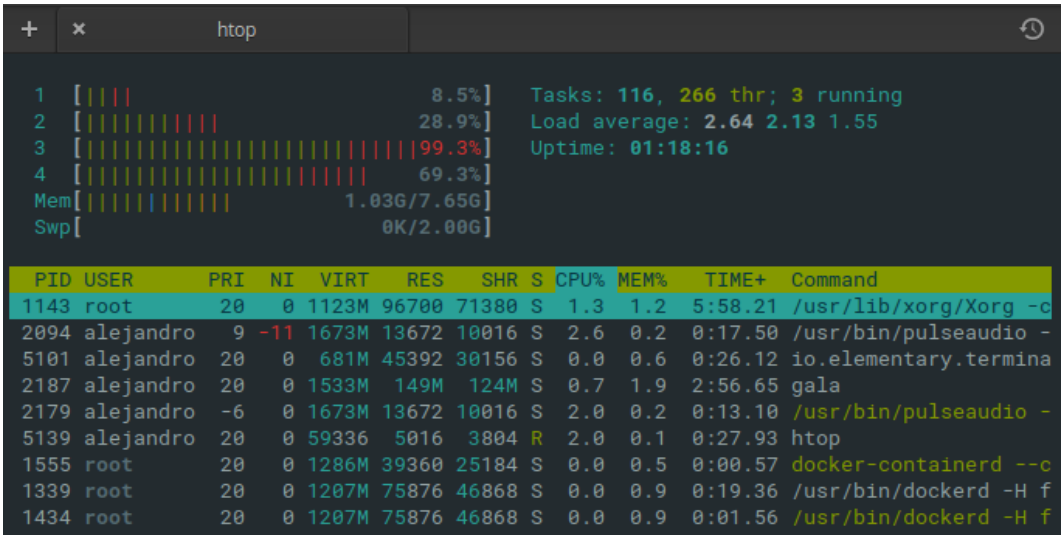

Figura 43: Situación registrada por htop cuando se está ejecutando Torre.py y se está manteniendo la conexión con un avión

Por último, si seguimos aumentando el número de peticiones al servidor, creando varios aviones, lo que se observa es que la exigencia de recursos pese a ser alta, no varía mucho entre uno y varios aviones. En este caso se han llegado a realizar un número máximo de tres conexiones simultáneas. Han sido únicamente tres conexiones, no por falta de recursos en el ordenador que asume el rol de torre de control, sino porque no se puede establecer más de una conexión por cada máquina y solo se disponía de 3 ordenadores para hacer la prueba. En este caso se hicieron pruebas con dos portátiles y la Raspberry Pi 3B+.

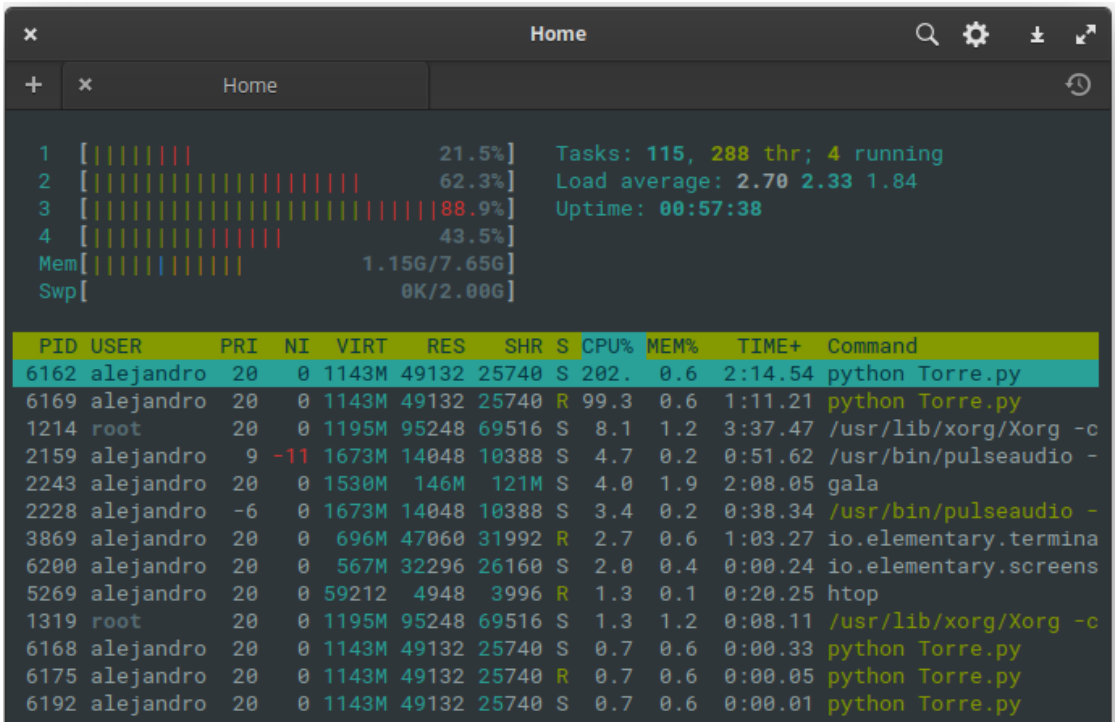

Figura 44: Situación registrada por htop cuando se está ejecutando Torre.py y se está manteniendo la conexión con tres aviones de forma simultánea

Lo que se puede observar, es que las exigencias de memoria RAM del sistema en cualquier caso, son muy bajas dado que superan el Gb por muy poco en cualquiera de los escenarios. Esto hace que el factor limitante del sistema no sea la cantidad disponible de memoria RAM, sino el número de CPUs disponibles y el número de sockets con el que se pueda acceder a cada CPU.

A la hora de escalar el sistema, este factor debería ser tenido en cuenta para

priorizar el número de CPUs del servidor, por encima de la memoria RAM o el almacenamiento en disco, como podría interesar en otro tipo de proyectos.

Aun así, para optimizar el uso de los recursos del sistema, en futuros trabajos, se debería de llevar a cabo una optimización del código en el que se minimizasen las variables y los bucles utilizados. También sería interesante el uso de tarjetas gráficas para todos los cálculos en paralelo que se deben realizar en el proceso de cálculo de rutas, conversión de coordenadas, etc... Todo esto podría aligerar en gran medida la carga de trabajo para el ordenador y reducir el consumo tanto energético, como de recursos informáticos, ajustando el presupuesto del proyecto.

4.6. Prueba final del sistema completo

Por último, para finalizar la fase de pruebas y a modo de prueba resumen de todo lo comprobado anteriormente, se quiso comprobar el correcto funcionamiento de todo el sistema con varios aviones a la vez, se propuso un escenario, en el que se iniciaba la torre de control, se iban conectando aviones a los que se les asignaban diferentes rutas y cada uno de ellos debía de llegar sin problema hasta su destino.

El objetivo era simular una situación cualquiera en el aeropuerto con varios aviones para poder observar si las maniobras de rodaje se llevaban a cabo de forma segura y sin incidentes.

Al contar solamente con un robot, se decidió realizar esta prueba en el entorno simulado utilizado para las primeras pruebas de movimiento y las pruebas de prioridad. Se crearon 3 aviones y se les asignó una ruta al azar. El sistema debía de ser capaz de llevar a cabo todo el proceso de maniobras de rodaje sin problema.

El resultado fue satisfactorio ya que se hicieron varias pruebas con diferentes rutas y en cualquiera de los casos en los que los aviones coincidían, eran capaces de cederse el paso entre sí e identificar correctamente en qué situación se encontraban. Todos los aviones llegaron en todo momento a sus pistas de despegue sin problema.

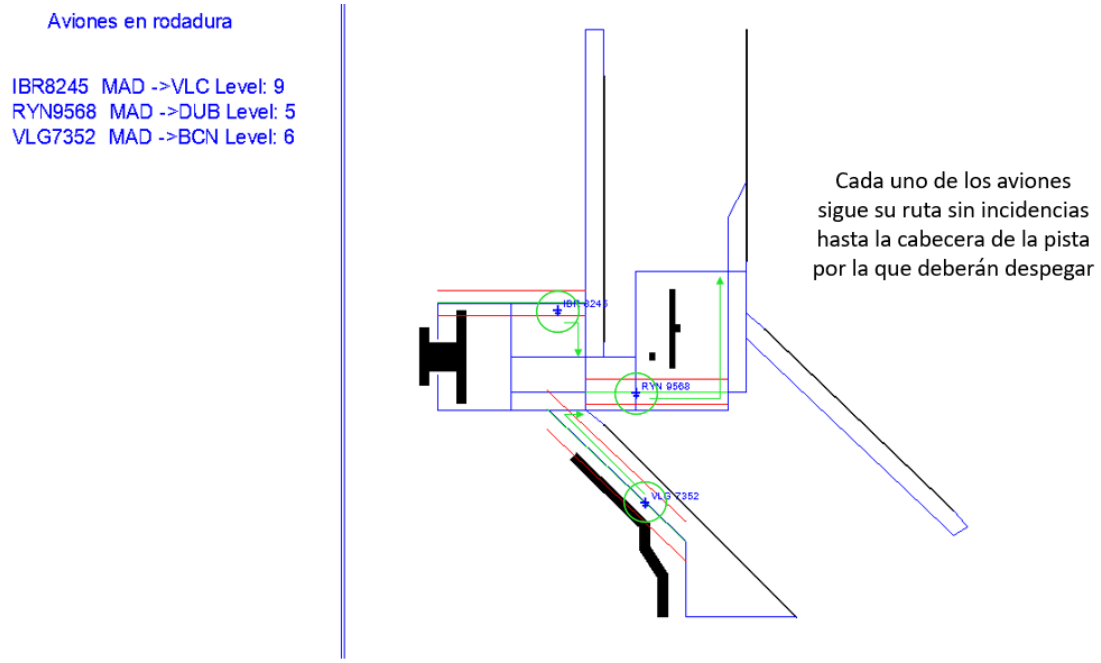

Figura 45: Situación provocada en la prueba final con 3 aviones realizando la maniobra de rodaje de forma simultánea

5. Conclusiones y líneas de trabajo futuras

Con todos los datos recopilados en los puntos anteriores, se puede concluir que el sistema diseñado y puesto a prueba mediante este prototipo, funciona y es un sistema que, realizando una serie de mejoras en aspectos como el sensor GNSS o la eficiencia de los recursos informático utilizados, podría llegar a ser un sistema desplegable en entornos reales de aeropuertos, simplificando así la labor de pilotos y controladores aéreos y minimizando los riesgos durante la maniobra de rodadura.

Se proponen ahora, una serie de aspectos que se deberían tener en cuenta en un futuro si se decidiese continuar con este trabajo, con el objetivo de avanzar hacia una serie de pruebas más realistas con las que probar que el sistema podría ser eficaz en una pequeña aeronave como una avioneta o un ultraligero.

5.1. Problemas observados en el prototipo y soluciones propuestas

5.1.1. Imprecisiones en el posicionamiento GNSS

Como ya se ha comentado, el principal problema del prototipo ha sido la baja precisión a la hora de obtener las coordenadas tanto de la torre de control como del avión robot.

La solución propuesta para este aspecto es utilizar un receptor GNSS que nos proporcione, no una mayor precisión, sino un sistema de corrección DGNSS en tiempo real como los ya mencionados RTK o PPP.

Otra de las cosas que mejoraría la precisión en la posición es aumentar la tasa de recepción de posiciones de una cada tres segundos, a una por segundo por lo menos, aunque lo ideal sería una cada medio segundo. Esto, combinado con una técnica de predicción de posiciones entre muestra y muestra, que ayudase a hacer una estimación de la distancia que recorrerá el móvil durante el tiempo que pase entre el cálculo de una posición y otra, mejoraría mucho la seguridad del sistema y permitiría una mayor exactitud en las posiciones de la aeronave en todo momento.

Además, será preciso utilizar, si no una avioneta o un ultraligero, un móvil a una escala similar y que avance aproximadamente a su misma velocidad.

Con una posición más exacta y una menor relación entre el error del sensor GNSS y el tamaño del móvil, se podrán obtener resultados mucho más relevantes de los aquí obtenidos y más cercanos a la realidad para la que está pensado el sistema.

5.1.2. Gran consumo de recursos informáticos

Otro de los mayores problemas a los que se ha enfrentado el prototipo, ha sido la gran cantidad de recursos informáticos que consume el sistema cuando está en marcha.

Debido a la exigencia del programa, se fuerza de forma constante al servidor que lo alberga y además convierte el número de CPUs en un factor limitante a la hora de atender nuevas peticiones. Todo esto hace que el despliegue y el mantenimiento

del sistema se prevea más caro y por tanto, menos competitivo en el mercado.

Para que esto no sea un problema, en primer lugar se debe de optimizar el código creado, minimizando el número de bucles y variables utilizadas, así como la cantidad de CPU reclamada por cada uno de los hilos en marcha. Una vez se haya atajado este aspecto, si se plantease implantar el sistema en un entorno real, se debería de realizar una inversión en equipos potentes con una gran capacidad multiproceso y características técnicas que contengan, previsiblemente, un elevado número de CPUs y varias unidades de GPUs implicadas en el procesado en paralelo.

Con todo esto solucionado, sería viable un despliegue del sistema que, pese a tener una mayor inversión en un inicio, luego pudiese mantenerse a un menor coste y trabajase con un menor número de averías y errores, ya que al consumir menos recursos y forzar menos el ordenador, se alargará su vida útil.

5.1.3. Mejora del algoritmo de cálculo de rutas

Aunque el algoritmo de creación y cálculo de rutas funciona bien en una situación en la que todos los movimientos a realizar son en línea recta, que es el escenario más común en la mayoría de las maniobras de rodaje, hay muchos aeropuertos que cuentan con algún tramo de sus pistas de rodadura que describe algún tipo de curva o de arco y eso ahora mismo, no es algo que pueda hacer el algoritmo.

Para mejorarlo, una opción sería aplicar el símil de la circunferencia con el polígono de infinitas aristas. Definiendo una gran cantidad de puntos muy cercanos a lo largo de toda la curva a seguir, se podrían realizar curvas de diferente radio y longitud, e incluso con una curvatura cambiante. Esto nos lleva al segundo punto a mejorar en cuanto al algoritmo de planificación de rutas.

Tal y como se presenta en este prototipo, el algoritmo está preparado para recibir información del controlador aéreo, quien debe indicar una a una las intersecciones o puntos de referencia a través de los cuales debe de pasar la ruta del avión.

Lo ideal, para descargar a los controladores aéreos de trabajo sería desarrollar un algoritmo de planificación de rutas que tuviese en cuenta las distancias al resto

de aviones actualmente y en el futuro, ya que se conocen sus rutas si también se hallan realizando la maniobra de rodaje. También deberá tener en cuenta la distancia a recorrer y las prioridades de los aviones con los que deberá cruzarse una aeronave hasta llegar a su destino para minimizar al máximo tanto el riesgo de colisión, como el tiempo empleado en la maniobra.

5.2. Aspectos a tener en cuenta en un despliegue real

5.2.1. Ciberseguridad

Para las comunicaciones inalámbricas llevadas a cabo por este prototipo se ha utilizado el estándar 802.11ac (Wifi en las bandas de 2,4GHz y 5 GHz), pero aunque para un entorno de desarrollo supone una gran comodidad, cualquier persona con un smartphone, tablet, o portátil podría intentar atacar a la red mediante un ataque *man in the middle* o incluso *DoS - Denial of Service* o *DDoS - Distributed Denial of Service* si se llegase a conocer la contraseña de la red. Esto se debe a que el estándar 802.11ac, es utilizado por prácticamente todos los dispositivos con conectividad inalámbrica del mercado y cualquier persona en un aeropuerto poseerá si no uno, varios de ellos.

Por esta razón, si se decidiese implementar el prototipo en un aeropuerto, al menos, se debería de desplazar la frecuencia a una frecuencia fuera de las bandas de recepción y transmisión de este u otros estándares de redes inalámbricas. Una de las mejores opciones, dado que se trata de una comunicación en un entorno muy despejado y se podrían instalar antenas en puntos bastante elevados, sería una banda disponible entre 1 y 3 GHz.

Por último, además de este tipo de medidas de seguridad, habría que contar, por supuesto, con la redundancia tanto física como geográfica (en otro edificio del aeropuerto, por ejemplo) de todos los servidores del sistema para poder ofrecer en todo momento un servicio sin interrupciones ni fisuras. Además de la redundancia en cuanto a servidores, se debería de contar con redundancia del sistema en las aeronaves, contando con por lo menos dos tarjetas de red. Una de ellas debería de comunicarse con la torre de control, y la otra, realizar la función de *BroadcastServer* y *BroadcastClient* en una red ad-hoc en la que la comunicación no dependa de

ningún nodo central y permita a los aviones comunicarse entre sí directamente a través de una red creada por sus propios dispositivos.

5.2.2. Impacto en la actual infraestructura de los aeropuertos y certificación

Incluir una nueva tecnología en el mundo aeronáutico siempre es un gran reto. En primer lugar por la gran exigencia requerida en el proceso de certificación del sistema para calificarlo como seguro en un entorno aeronáutico.

Para poder implantar el sistema en un aeropuerto real, es necesario que ICAO lo certifique y que tanto las organizaciones aéreas nacionales (AESA - Agencia Estatal de Seguridad Aérea, FAA - Federal Aviation Administration) como regionales (EASA - European Aviation Safety Agency) lo ratifiquen y lo acepten en sus aeropuertos.

Esto conlleva una inmensa cantidad de pruebas, revisiones y puestas a punto, por lo que la simplicidad del sistema es clave. Por ello, se debe de reducir al mínimo el código y las funcionalidades del mismo, siempre que se cumplan los requisitos mínimos del sistema en lo que a seguridad y disponibilidad se refiere.

Además, utilizar sistemas y tecnologías ya certificados, simplifica mucho el proceso, y abarata enormemente los costes al no tener que certificar una nueva tecnología desde cero.

Es por eso, el sistema que se propone, en un caso real, debería de utilizar GBAS, el sistema de aumento GNSS ya mencionado previamente en el punto 2.2.2 de esta memoria. Esta tecnología se encuentra ya implantada en la mayoría de los aeropuertos comerciales del mundo (aeropuertos internacionales, los principales beneficiados de este sistema por volumen de tráfico). Realizar la certificación para este tipo de tecnología sería mucho más simple que certificar un nuevo sistema de aumento desde cero.

Además, al estar ya desplegada, la inversión que deberían de hacer los aeropuertos es mucho menor, ya que cuentan con toda la infraestructura de un sistema GBAS y solo se deberían de instalar los sistemas informáticos necesarios para el control

del software. Lo mismo se aplica a la inversión en hardware para las aeronaves, se podría utilizar el mismo hardware que ya ha sido certificado para el uso de un sistema GBAS y añadir funcionalidades software en el ordenador de a bordo.

6. Conclusión final del proyecto

En base a todo lo expuesto en este trabajo, se concluye que el sistema de guiado de aeronaves mediante GNSS propuesto para este Trabajo Fin de Grado, es un sistema que aunque está en una fase muy prematura de desarrollo, podría llegar a aportar una mayor seguridad en los aeropuertos y aliviar la carga tanto de pilotos, como de controladores aéreos durante la realización de la maniobra de rodaje.

Para poder realizar esta afirmación, en primer lugar se ha realizado una investigación sobre el estado de la técnica de las diferentes tecnologías involucradas en el posicionamiento de móviles, a partir de dicha información se diseñó un prototipo con el que poder realizar una serie de pruebas que, finalmente, permitieron observar la eficacia de los algoritmos diseñados, pero también las debilidades del diseño y por tanto, los puntos más críticos del sistema en el hipotético caso de una implementación real. Habiendo observado esas debilidades, se proponen una serie de líneas de trabajo futuras con las que seguir desarrollando y mejorando el sistema, con el objetivo de poder realizar pruebas en un entorno más realista y que permita una nueva evolución del sistema.

A. Apéndice matemático

A.1. Filtros FIR paso bajo empleados

Como ya se ha comentado, se calcularon filtros de orden 2, 5, 7 y 9 para obtener filtros de 3, 5, 8 y 10 muestras respectivamente.

A continuación se muestran los coeficientes utilizados por cada uno de ellos y su respuesta al impulso en tiempo y frecuencia:

Filtro de orden 2

Los coeficientes calculados para este filtro con $f_s = 0{,}3333Hz$, $f_c = 0{,}14Hz$ y $f_stop = 0{,}15Hz$ han sido 0.1158, 0.8810 y 0.1158, pero su suma es de 1.1125, por lo que se han ajustado a 0.1, 0.8 y 0.1.

Sus respuestas al impulso en tiempo y en frecuencia son:

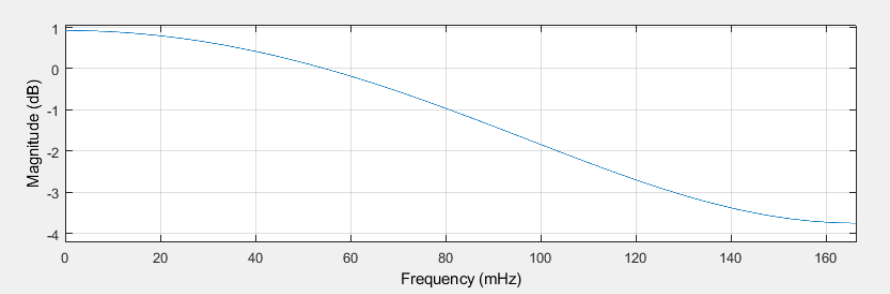

Figura 46: Respuesta al impulso en el dominio de la frecuencia del filtro FIR paso bajo de orden 2

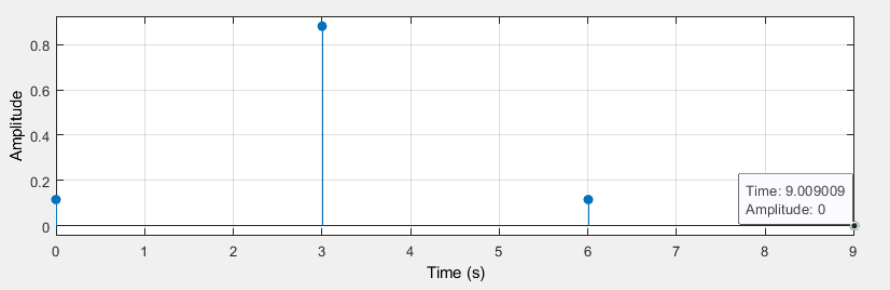

Figura 47: Respuesta al impulso en el dominio del tiempo del filtro FIR paso bajo de orden 2

Filtro de orden 4

Los coeficientes calculados para este filtro con $f_s = 0{,}3333Hz$, $f_c = 0{,}14Hz$ y $f_stop = 0{,}15Hz$ han sido -0.1142, 0.1261, 0.8698, 0.1261, -0.1142, pero su suma es de 0.8935, por lo que se han ajustado a -0.125, 0.175, 0.9, 0.175 y -0.125.

Sus respuestas al impulso en tiempo y en frecuencia son:

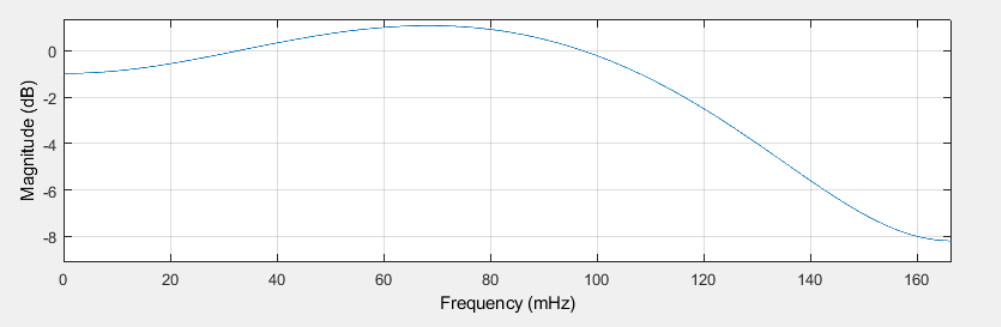

Figura 48: Respuesta al impulso en el dominio de la frecuencia del filtro FIR paso bajo de orden 4

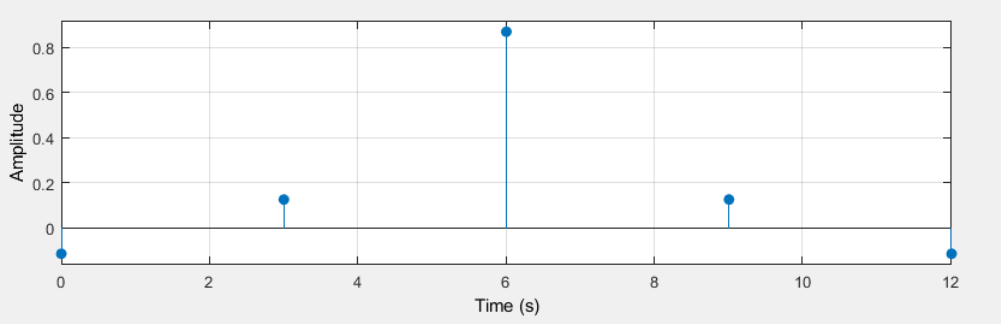

Figura 49: Respuesta al impulso en el dominio del tiempo del filtro FIR paso bajo de orden 4

Filtro de orden 7

Los coeficientes calculados para este filtro con $f_s = 0{,}3333Hz$, $f_c = 0{,}14Hz$ y $f_stop = 0{,}15Hz$ han sido -0.0195, 0.0713, -0.1764, 0.6243, 0.6243, -0.1764, 0.0713 y -0.0195, pero su suma es de 0.9994, por lo que se han ajustado a -0.0195, 0.0715, -0.1765, 0.6245, 0.6245, -0.1765, 0.0715 y -0.0195.

Sus respuestas al impulso en tiempo y en frecuencia son:

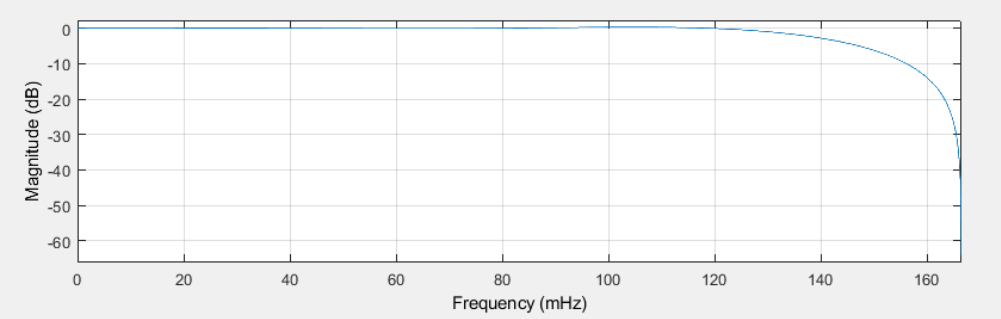

Figura 50: Respuesta al impulso en el dominio de la frecuencia del filtro FIR paso bajo de orden 7

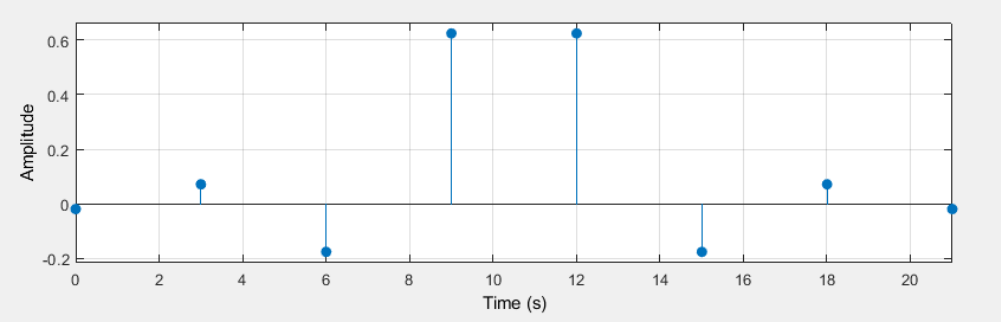

Figura 51: Respuesta al impulso en el dominio del tiempo del filtro FIR paso bajo de orden 7

Filtro de orden 9

Los coeficientes calculados para este filtro con $f_s = 0{,}3333Hz$, $f_c = 0{,}14Hz$ y $f_stop = 0{,}15Hz$ han sido -0.0119, -0.0178, 0.0698, -0.1754, 0.6240, 0.6240, -0.1754, 0.0698, -0.0178 y -0.0119, pero su suma es de 0.9772, por lo que se han ajustado a -0.012, -0.0178, 0.0798, -0.1754, 0.6254, 0.6254, -0.1754, 0.0798, -0.0178 y -0.012.

Sus respuestas al impulso en tiempo y en frecuencia son:

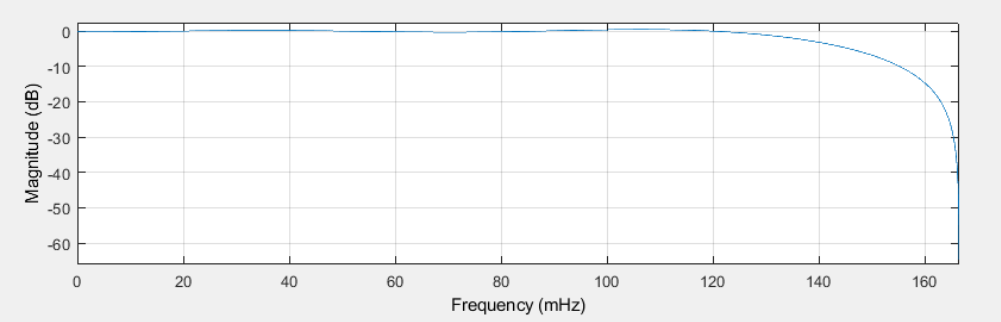

Figura 52: Respuesta al impulso en el dominio de la frecuencia del filtro FIR paso bajo de orden 9

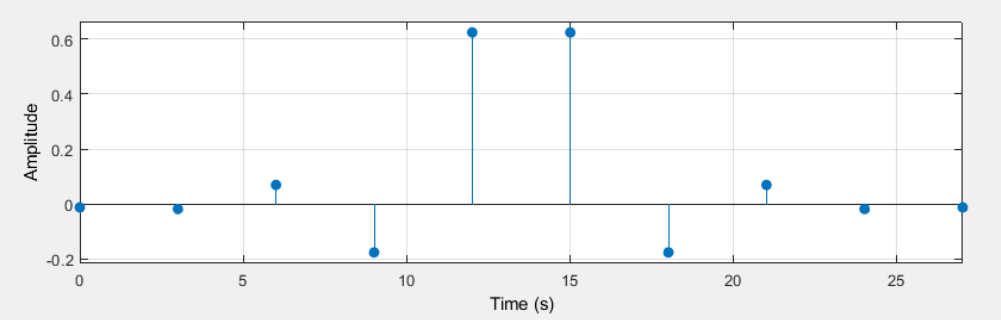

Figura 53: Respuesta al impulso en el dominio del tiempo del filtro FIR paso bajo de orden 9

A.2. Cálculo de ruta y rectas paralelas de tolerancia

En primer lugar se identifica el tipo de recta, para ello se miran los tres casos posibles: que las coordenadas y (latitud en UTM) del punto de partida y de llegada sean iguales *self.route[i][0] == self.route[i+1][0]* (caso de recta horizontal, $y = n$), que las x (longitud en UTM) del punto de partida y llegada sean iguales *self.route[i][1] == self.route[i+1][1]* (caso de recta vertical, $x = cte$) y por último, si ninguno de los casos anteriores es el del tramo a calcular, es una recta genérica de la forma $y = mx+n$.

Una vez se ha identificado el tipo de recta que se utilizará para el cálculo de la ruta, se procede a dar valor a los parámetros m,n,n_izq y n_dcha que devuelve este método. La variable m es la pendiente de la recta, $m = \Delta y/\Delta x$, n es el

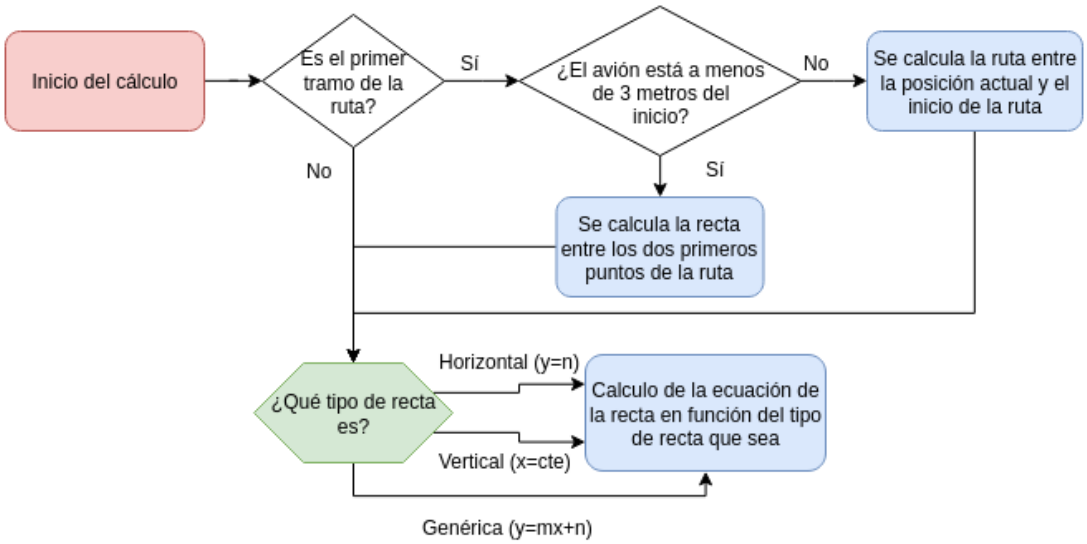

Figura 54: Diagrama de bloques donde se muestra el flujo de trabajo del algoritmo encargado del cálculo de las rutas de la aeronave

término independiente de la recta. A partir de un punto conocido $P = (x_1, y_1)$ $n = -mx_1 + y_1$. Por último, n_izq y n_dcha son los términos independientes de las rectas paralelas separadas a una distancia d de la recta principal. Al compartir la pendiente, solo se necesita conocer este término independiente.

Caso de recta horizontal

En este caso, la ecuación de la recta obedece a la forma $y = n$, dado que $\Delta y = 0$, la pendiente m será nula igualmente. Por eso, y conociendo los puntos de inicio y final de la recta, $y = self.route[i][0]$ siendo $self.route[i][0]$ la latitud (coordenada y en UTM) del punto i-ésimo de la recta.

Para poder determinar el valor de n_izq y n_dcha se debe de conocer el sentido de avance, si es de izquierda a derecha o de derecha a izquierda. Para ello se comparan las coordenadas de longitud (x en UTM) de los puntos de llegada y de salida y se determina el sentido del movimiento.

En el caso de moverse hacia la derecha el valor de m, n_izq y n_dcha será: $m = 0$, $n\_izq = n + d$ y $n\_dcha = n - d$.

En el caso de moverse hacia la izquierda, el valor de m, n_izq y n_dcha será: $m = 0$, $n\_izq = n - d$ y $n\_dcha = n + d$.

Caso de recta vertical

En este caso, la ecuación de la recta obedece a la forma $x = cte$, dado que $\Delta x = 0$, la pendiente m será infinita y adoptará el valor de una *string* $m =$ "inf". Por eso, y conociendo los puntos de inicio y final de la recta, $x =$ *self.route[i][1]* siendo *self.route[i][1]* la longitud (coordenada x en UTM) del punto i-ésimo de la recta.

En este caso, no tiene sentido hablar de un término independiente, por lo que $n =$ $x =$*self.route[i][1]*. Matemáticamente no es correcto, pero a la hora de programar sirve para identificar el valor que adopta la coordenada x en este tipo de rectas. Lo mismo sucede con n_izq y n_dcha. No es correcto matemáticamente, pero en el programa sirve para identificar las posiciones en la coordenada x de las rectas paralelas.

Para poder determinar el valor de n_izq y n_dcha, de nuevo se debe de conocer el sentido del movimiento, si es de abajo a arriba o de arriba a abajo. Para ello se comparan las coordenadas de latitud (y en UTM) de los puntos de llegada y de salida y se determina el sentido del movimiento.

En el caso de moverse de abajo a arriba, el valor de m, n_izq y n_dcha será: $m =$ "inf", $n\_izq = n - d$ y $n\_dcha = n + d$.

En el caso de moverse de arriba a abajo, el valor de m, n_izq y n_dcha será: $m =$ "inf", $n\_izq = n + d$ y $n\_dcha = n - d$.

Ecuación de recta genérica

Por último, en el caso de que la aeronave se encuentre con una recta genérica donde no se cumpla que las coordenadas x o y sean iguales a las del próximo punto de la ruta (lo cual será el caso más genérico), deberá de calcular la ecuación de una recta que obedezca a la forma $y = mx + n$ y calcular las rectas paralelas a ella que cumplan una cierta distancia d llamada tolerancia.

En este caso obtendremos los siguientes valores para m y n: $m = \frac{\Delta y}{\Delta x}$ y dado un punto $P = (x_1, y_1)$ conocido (punto de inicio de la ruta), $n = -mx_1 + y_1$.

En el caso del cálculo del término independiente de las rectas paralelas, se aplica la fórmula de la distancia entre un punto y una recta:

$$d = \frac{|Ax_1 + By_1 + C|}{\sqrt{A^2 + B^2}} \tag{14}$$

Siendo A, B y C los términos de la recta a la que se van a trazar las paralelas descrita de la forma $Ax + By + C = 0$.

Relacionando esto, con lo que se conoce de la recta calculada (m y n) en la forma $y = mx + n$, se obtiene que $y = mx + n = \frac{-A}{B} - \frac{C}{B}$. Por lo que $m = -\frac{A}{B}$ ($A = \Delta y$ y $B = -\Delta x$) y $n = -\frac{C}{B}$.El elemento desconocido es C, por lo que despejando de la ecuación de la distancia entre un punto y una recta se obtiene:

$$d\sqrt{A^2 + B^2} = |Ax_1 + By_1 + C| \tag{15}$$

si por simplificación se asume que,

$$Ax_1 + By_1 = D \tag{16}$$

se obtiene,

$$d\sqrt{A^2 + B^2} = |D + C| \tag{17}$$

y por tanto,

$$C = d\sqrt{A^2 + B^2} - D \tag{18}$$

$$C = -d\sqrt{A^2 + B^2} - D \tag{19}$$

Con las dos últimas expresiones se obtienen los valores de C que junto con $B = \Delta x$ permiten conocer los valores de n_izq y n_dcha. Para saber a cual corresponde cada valor, se debe de estudiar de nuevo la dirección del movimiento.

Si se avanza hacia y (latitudes) mayores respecto de la latitud del punto de inicio:

$$n\_izq = -\frac{-d\sqrt{A^2 + B^2} - D}{B} \tag{20}$$

$$n\_dcha = -\frac{d\sqrt{A^2+B^2}-D}{B} \tag{21}$$

Si por el contrario, se avanza hacia y (latitudes) menores respecto de la latitud del punto de inicio:

$$n\_izq = -\frac{d\sqrt{A^2+B^2}-D}{B} \tag{22}$$

$$n\_dcha = -\frac{-d\sqrt{A^2+B^2}-D}{B} \tag{23}$$

Por último, se deja el código diseñado para la consulta del lector:

```
def calcularruta(self, i):
"""
    Método del objeto Plane.py encargado del algoritmo de cálculo de
    rutas
"""
        d=2

        if self.route[i+1][0]==self.route[i][0]:
            #Caso de recta horizontal
                m=0
                n=self.route[i][0]
                if self.route[i+1][1]>self.route[i][1]:
                    #Va hacia la derecha
                        n_izq=self.route[i][0]+d
                        n_dcha=self.route[i][0]-d
                else: #Va hacia la izquierda
                        n_izq=self.route[i][0]-d
                        n_dcha=self.route[i][0]+d

        elif self.route[i+1][1]==self.route[i][1]:
            #Caso de recta vertical
                m="inf"
                n=self.route[i][1]
```

```
            if self.route[i+1][0]>self.route[i][0]:
                #Va hacia arriba
                    n_izq=self.route[i][1]-d
                    n_dcha=self.route[i][1]+d
            else: #Va hacia abajo
                    n_izq=self.route[i][1]+d
                    n_dcha=self.route[i][1]-d

    else:
            delta_x=self.route[i+1][1]-self.route[i][1]
            delta_y=self.route[i+1][0]-self.route[i][0]

            m=(delta_y)/(delta_x)
            n=-m*self.route[i][1]+self.route[i][0]

            A=delta_y
            B=-delta_x
            D=A*self.route[i][1]+B*self.route[i][0]

            if self.route[i+1][0]>self.route[i][0]:
                    #Caso de y positivas (vamos hacia arriba)
                    n_izq=(D-d*sqrt((A**2)+(B**2)))/B
                    n_dcha=(D+d*sqrt((A**2)+(B**2)))/B

            else:
                    #Caso de y negativas (vamos hacia abajo)
                    n_izq=(D+d*sqrt((A**2)+(B**2)))/B
                    n_dcha=(D-d*sqrt((A**2)+(B**2)))/B

    self.rectas=[m,n,n_izq,n_dcha]

    return
```

A.3. Comprobación de llegada

En este punto del apéndice se explica brevemente el cálculo realizado para determinar si se ha llegado o no a un punto del mapa, entendiendo por punto del mapa el punto de partida de la ruta, cualquier punto intermedio o el punto final. Se utiliza para saber cuando se ha llegado al final de una de las rectas calculadas.

La fórmula aplicada es la fórmula para el cálculo de la distancia entre dos puntos:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{24}$$

siendo x_1 , y_1 y x_2 e y_2 las coordenadas de los puntos entre los que se desea calcular la distancia, $P_1 = (x_1, y_1)$ y $P_2 = (x_2, y_2)$.

```
def checkarrival(self, i):
    #Si es el primer punto de la ruta, se determina si el avión está
    #alineado a menos de 3 metros con el centro de la ruta
        if i==1:
                if sqrt(((self.lat-self.route[i+1][0])**2)+
                ((self.lon-self.route[i+1][1])**2))<=3:
                        return True
                else:
                        return False
        #Si se trata de un punto normal de la ruta, necesita estar
        #a menos de 1.5 metros
        elif sqrt(((self.lat-self.route[i+1][0])**2)+
        ((self.lon-self.route[i+1][1])**2))<=1.5:
                return True
        else:
                return False
```

B. Características técnicas del hardware utilizado

B.1. Sensor GNSS y antena utilizados

Sensor GNSS de la marca Stemedu comprado a través del portal web Amazon [9].

Concepto	Características
Tamaño	28 x 48 x 8,5 mm
Frecuencia de trabajo	1561 MHz a 1575,42 MHz
Sistemas GNSS soportados por el sensor	GPS / GLONASS
Sistemas GNSS soportados por la antena	GPS / BeiDou
Frecuencia de actualización	1 a 10 Hz
Velocidad de transmisión	4800,9600,19200,38400,57600,112500 bps
Canales	56
Temperatura de funcionamiento	-40ºC a 85ºC
Potencia mínima de rastreo	-164 dBm
Potencia mínima de captura	-159 dBm
Potencia mínima de arranque en frío	-147 dBm
Tiempo de inicio en frío	26 segundos (promedio)
Tiempo de inicio con arranque cálido	24 segundos (promedio)
Tiempo de inicio en caliente	1 segundo (promedio)
Posición horizontal	2,0 m de promedio
Señal de tiempo	RMS 30 ns
Altura máxima	50000 metros
Velocidad máxima	500 m/s
Aceleración máxima	4G

Cuadro 1: Tabla de especificaciones del módulo GNSS y su antena [9]

B.2. Hardware utilizado por la torre de control

Ordenador portátil de la marca Lenovo. Modelo Ideapad 300-150 [10].

Concepto	Características
Tamaño	265 x 384 x 22,9 mm
Peso	2,3 Kg
Sistema Operativo	Elementary OS (Linux) / Windows 10
CPU	Intel® Core™ i5 de 6° generación y bajo consumo
GPU	AMD Radeon Graphics
RAM	8GB
Red cableada	Gigabit Ethernet
Red inalámbrica	2.4GHz and 5GHz 802.11b/g/n/ac Wi-Fi
Bluetooth	Bluetooth 4.0
Almacenamiento	1 TB HDD
GPIO	No
Puertos	HDMI, VGA, 3.5mm analogue audio-video jack, 1x USB 3.0, 2x USB 2.0, lector de tarjetas SD, SDHC, SDXC, MMC

Cuadro 2: Tabla de especificaciones de Lenovo Ideapad 300 [10]

B.3. Hardware utilizado por el robot

Mini ordenador de la marca Raspberry Pi [13], sensor de la marca STMicroelectronics [11] y batería de la marca RavPower [12].

Concepto	Características
Tamaño	155 x 215 x 67 mm
Peso total	470 g
Controlador de motores L298N	
Vin	4,5 VDC min / 7 VDC max
I salida max	2 A (DC Operacion) / 3 A (DC pico)
Vin motores	2,5 VDC min / 46 VDC max
Vin pines de control	-1 V DC min / 2,3 V DC max
Potencia disipada (a 75ºC)	25 W
Temperatura de operación	-25ºC a 130ºC
Batería portátil	
Tamaño	83 x 165 x 21 mm
Peso	380 g
Capacidad	22000 mAh
Vout	5 VDC
I out	2,4 A max

Cuadro 3: Tabla de especificaciones del robot diseñado [11] y [12]

Concepto	Características
Tamaño	82 x 56 x 19.5 mm
Peso	50 g
Sistema Operativo	Raspbian (Linux)
CPU	Broadcom BCM2837B0 quad-core A53 (ARMv8) 64-bit @ 1.4GHz
GPU	Broadcom Videocore-IV
RAM	1GB
Red cableada	Gigabit Ethernet
Red inalámbrica	2.4GHz and 5GHz 802.11b/g/n/ac Wi-Fi
Bluetooth	Bluetooth 4.2, Bluetooth Low Energy (BLE)
Almacenamiento	Micro-SD
GPIO	40-pin GPIO header
Puertos	HDMI, 3.5mm analogue audio-video jack,4x USB 2.0, CSI, DSI

Cuadro 4: Tabla de especificaciones de Raspberry Pi 3B+ [13]

C. Presupuesto del prototipo

En este apéndice se detalla el coste total asociado al desarrollo del prototipo. Esto incluye el hardware utilizado y el coste de las horas de ingeniería empleadas (60€/hora).

Se incluyen 2 Raspberrys Pi 3B+ no porque se hayan creado dos robots, sino porque se precisaba de un tercer equipo que adoptase el papel de segundo avión en el sistema para poder realizar la prueba de predicción de colisiones.

No se encuentran incluidos los costes del ordenador portátil que adopta el rol de torre de control ya que se trataba de un ordenador personal.

Concepto	Precio	Cantidad	Total
Horas de trabajo	60 €	195	11700 €
Raspberry Pi 3B+	44,83 €	2	89,66 €
Pack motores + estructura robot	12,99 €	1	12,99 €
Pack antena + sensor GNSS Stemedu ST3199A	17,99 €	2	35,98 €
Cables Dupont	3,99 €	1	3,99 €
Controlador de motor DC L298N	6,99 €	1	6,99 €
Banco de batería 20000 mAh	19,99 €	1	19,99 €
Total			11869,5 €

Cuadro 5: Presupuesto del prototipo

D. Código elaborado

D.1. Código elaborado para el funcionamiento de la torre de control

D.1.1. Torre.py

Aquí se presenta el código troncal de la torre de control. Se encarga de generar la interfaz gráfica, calibrar el mapa, llevar registro de cada uno de los aviones que entran en el sistema y de ir generando servidores TCP a medida que se van ocupando los ya creados hasta llegar al número de conexiones máximas admitidas por el sistema.

```
import sys, pygame, time, threading
from os import system
import RedTorre
sys.path.append('/home/alejandro/Documentos/TFG/Auxiliar/')
import Aux, Plane, Interface


if __name__ == "__main__":
    #Variable definition
    maxconex=10
    numconex=1
    initport=65000
    new=True
    twr_pos=[0, 0]
    Aviones = []
    Threads = []
    Error_base=[]

    Avion=Plane.Plane('**',0000,'***','*',0,0)
    Aviones.append(Avion)
```

```
#Inicializamos pygame y generamos la interfaz
pygame.init()
pygame.font.init()
size = 1050, 650
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Aeropuerto Adolfo Suarez Madrid Barajas")
width, height = 1200, 650
myfont=pygame.font.SysFont('Arial',25)
screen.fill((255,255,255))
Aux.text("Aviones en rodadura", 55,5, myfont, screen,(0,0,255))
myfont=pygame.font.SysFont('Arial',20)
#Dibujamos el aeropuerto sorbre la interfaz generada
Interface.draw_interface(screen)
pygame.display.flip() #Refresca la interfaz

#Se calibra el mapa a partir de la posición conocida de la torre
cornercoord, twr_pos =Aux.callibratemap(Error_base, twr_pos)
#Se representan en el mapa las 4 esquinas y la torre de control
for i in range(0,5):
    Aux.showinmap(cornercoord[1],cornercoord[i],myfont,screen)

#Se muestra por consola las coordenadas UTM de las esquinas y la torre
print('Coord Torre'+str(cornercoord[0]))
print('Primera esquina'+str(cornercoord[1]))
print('Segunda esquina'+str(cornercoord[2]))
print('Tercera esquina'+str(cornercoord[3]))
print('Cuarta esquina'+str(cornercoord[4]))
pygame.display.flip()#Refresca la interfaz

# Comenzamos el bucle principal de la torre de control
run=True
while run and numconex<=maxconex:
```

```
        #Capturamos los eventos que se han producido
        for event in pygame.event.get():
        #Si el evento es salir de la ventana, terminamos
            if event.type == pygame.QUIT:
                run = False

        #En el caso de ser la primera iteración, se crea el primer servidor
        if new == True:
            Torre_Thread=threading.Thread(name='Torre_Thread',
            target=RedTorre.tcpserver, args=(numconex, initport
            +numconex-1,Aviones, cornercoord[1], twr_pos,
            cornercoord[0], myfont,screen, stop), daemon=True)
            Torre_Thread.start()
            Threads.append(Torre_Thread)
            print('Nuevo servidor disponible en el puerto '
            + str(initport+numconex-1))

        if Aviones[numconex-1].n != 0 and numconex<maxconex:
            #Cuando recibimos una nueva conexión, se crea un nuevo
            #objeto avión vacío y se incrementa en uno el número de
            #conexión, lo que incrementará a su vez el número de
            #puerto
            print('Nueva conexion iniciada en el puerto '
            +str(initport+numconex-1))
            numconex=numconex+1
            Avion = Plane.Plane('**',0000,'***','*',0,0)
            Aviones.append(Avion)
            new=True
        else:
            new=False

    #Una vez se haya salido del bucle porque el usuario lo haya
```

```
    #solicitado se cierran todos los hilos pendientes
    for i in range(0,len(Threads)-1):
        Threads[i].join()
        print ('He cerrado el hilo '+ str(i+1))

    # Salgo de pygame
    pygame.quit()
```

D.1.2. RedTorre.py

El código mostrado bajo estas líneas es el encargado de establecer y mantener la conexión TCP con la aeronave desde que se inicia hasta que finaliza su maniobra de rodaje. Además, se encarga de calcular el error relativo percibido por la torre de control, de generar la ruta individualizada para esa aeronave y de representar la información recibida a través de la interfaz gráfica que incorpora el sistema.

```
import socket,sys,time,pickle,pygame, numpy
from gps import *
from math import sqrt
sys.path.append('/home/alejandro/Documentos/TFG/Auxiliar/')
import Plane, Aux

def tcpserver(numconex, port, Aviones, ref_coordenates, twr_coord,
twr_coorde_exactas, myfont, screen, stop):
    """
    Función encargada de gestionar una de las conexiones recibidas.
    Recibimos como argumentos el numero de conexión que corresponde a
    nuestro servidor, el puerto en el que las escuchara, un objeto avión
    en el que almacenaremos los datos, unas coordenadas de
    referencia
    que nos servirán para posicionar el avión en el mapa y los
    datos necesarios para mostrar por pantalla la información.
    """
    ruta=[]
```

```
Error=[0,0]
nummuestras=10
posprev=[]
#Iniciamos el objeto encargado de recibir información del sensor GNSS
gpsd = gps(mode=WATCH_ENABLE|WATCH_NEWSTYLE)
#Iniciamos un socket de tipo Internet y en modo stream (TCP)
server=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
#Con la siguiente opción permitimos al socket reutilizar la dirección
#si se ha quedado colgado en algún momento del programa y no se
#ha cerrado correctamente
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
#Hacemos el servidor visible desde el exterior en el puerto port
server.bind(('192.168.1.160',port))
#Numero máximo de peticiones que queremos encolar antes de rechazar
#una conexión. En este caso, solo admitimos un avión por puerto
server.listen(0)

#Se pide al controlador aéreo que cree una ruta para la aeronave
ruta=Aux.crearruta(ref_coordenates)

#Se llena el vector de posiciones previas con las coordenadas de la
#torre obtenidas al calibrar el mapa promediando
for i in range(0,nummuestras-1):
    posprev.append(list(twr_coord))

#Tenemos que hacer el cálculo del error relativo de la posición en la
#misma función porque si lo ponemos en otro hilo, se bloquea la
#dirección de la memoria que tiene el hueco de "Error" y no
#podemos acceder a él.
Error=getErrorrelativo(gpsd,posprev,twr_coord, nummuestras)
#Esperamos a que haya una solicitud de conexión y la aceptamos
conexion, addr = server.accept()
```

```
while True:
    try:
        #Le mandamos el numero de conexión, el error relativo
        #calculado y la ruta creada
        conexion.send(pickle.dumps([numconex, Error, ruta]))
    except:
        print('Conexion perdida con el avión '+str(numconex-1))
        Aviones[numconex-1].stop=True
        Aux.updatescreen(Aviones,ref_coordenates, myfont,screen)
        return

    try:
        #Recibimos los datos desde el avión. Un máximo de 4096
        #bytes
        data=conexion.recv(4096)
    except:
        print('Conexion perdida con el avión '+str(numconex))
        Aviones[numconex-1].stop=True
        Aux.updatescreen(Aviones,ref_coordenates, myfont,screen)
        return

    #Devolvemos los datos obtenidos de formato binario a
    #un objeto Avión
    Aviones[numconex-1]=pickle.loads(data)
    #Representamos por la interfaz toda la información del Avión
    Aux.updatescreen(Aviones,ref_coordenates, myfont,screen)

    if Aviones[numconex-1].stop==True:
        break

    Aviones[numconex-1].n=numconex
```

```
            print(twr_coord)
            print([Aviones[numconex-1].lat,Aviones[numconex-1].lon])
            distanciaatorre=sqrt(((twr_coord_exactas[0]
            -Aviones[numconex-1].lat)**2)+((twr_coord_exactas[1]
            -Aviones[numconex-1].lon)**2))
            print('Distancia a la torre: '+str(distanciaatorre))

            #Se vuelve a calcular el error relativo para el próximo ciclo
            Error=getErrorrelativo(gpsd,posprev,twr_coord,nummuestras)

        conexion.close()

    def getErrorrelativo(gpsd,posprev, twr_coord, nummuestras):
        while True:
            report = gpsd.next()
            if report['class'] == 'TPV':
                coordenadas_gps=list(Aux.convert2UTM([getattr(report,'lat',0.0),
                getattr(report,'lon',0.0)]))
                print('Coordenadas recibidas por GPS '+str(coordenadas_gps))

                #En función del número de muestras que se hayan decidido utilizar
                #para el filtro FIR, se utilizarán unos coeficientes u otros
                if nummuestras==3:
                    coordenadas_gps[0]=0.1*coordenadas_gps[0]+0.8*posprev[0][0]
                    +0.1*posprev[1][0]
                    coordenadas_gps[1]=0.1*coordenadas_gps[1]+0.8*posprev[0][1]
                    +0.1*posprev[1][1]
                elif nummuestras==5:
                    coordenadas_gps[0]=-0.125*coordenadas_gps[0]+0.175*posprev[0][0]
                    +0.9*posprev[1][0]+0.175*posprev[2][0]-0.125*posprev[3][0]
                    coordenadas_gps[1]=-0.125*coordenadas_gps[1]+0.175*posprev[0][1]
                    +0.9*posprev[1][1]+0.175*posprev[2][1]-0.125*posprev[3][1]
```

```
        elif nummuestras== 8:
            coordenadas_gps[0]=-0.0195*coordenadas_gps[0]+0.0715*
            posprev[0][0]-0.1765*posprev[1][0]+0.6245*posprev[2][0]+0.6245*
            posprev[3][0]-0.1765*posprev[4][0]+0.0715*posprev[5][0]-0.0195*
            posprev[6][0]
            coordenadas_gps[1]=-0.0195*coordenadas_gps[1]+0.0715*
            posprev[0][1]-0.1765*posprev[1][1]+0.6245*posprev[2][1]+0.6245*
            posprev[3][1]-0.1765*posprev[4][1]+0.0715*posprev[5][1]-0.0195*
            posprev[6][1]
        elif nummuestras==10:
            coordenadas_gps[0]=-0.012*coordenadas_gps[0]-0.0178*
            posprev[0][0]+0.0798*posprev[1][0]-0.1754*posprev[2][0]+0.6254*
            posprev[3][0]+0.6254*posprev[4][0]-0.1754*posprev[5][0]+0.0798*
            posprev[6][0]-0.0178*posprev[7][0]-0.012*posprev[8][0]
            coordenadas_gps[1]=-0.012*coordenadas_gps[1]-0.0178*
            posprev[0][1]+0.0798*posprev[1][1]-0.1754*posprev[2][1]+0.6254*
            posprev[3][1]+0.6254*posprev[4][1]+0.1754*posprev[5][1]+0.0798*
            posprev[6][1]-0.0178*posprev[7][1]-0.012*posprev[8][1]

        print('Coordenadas filtradas '+str(coordenadas_gps))
        Error= Aux.getError(twr_coord,coordenadas_gps)
        print("Error relativo torre="+str(Error))

        for i in range(1,nummuestras-1):
            posprev[i]=posprev[i-1]

        posprev[0]=coordenadas_gps
    return Error
```

D.1.3. Interface.py

Este fichero Python recoge el código encargado de dibujar el plano del aeropuerto sobre la interfaz creada en Torre.py. Lo hace utilizando las funciones *pygame.draw.line* y *pygame.draw.polygon* de la librería de Python *pygame*.

```
import pygame

BASE_X=425
BASE_Y=25

def draw_interface(screen):
        pygame.draw.line(screen, (0,0,255), (347,0), (347,650), 1)
        pygame.draw.line(screen, (0,0,255), (350,0), (350,650), 1)

        #Pistas verticales
        pygame.draw.line(screen, (0,0,0), (BASE_X+183,BASE_Y+46),
        (BASE_X+183,BASE_Y+310), 2)
        pygame.draw.line(screen, (0,0,0), (BASE_X+325,BASE_Y+0),
        (BASE_X+325,BASE_Y+230), 2)

        #Lateral de la pista vertical izquierda
        pygame.draw.line(screen, (0,0,255), (BASE_X+165,BASE_Y+0),
        (BASE_X+165,BASE_Y+378), 1)
        pygame.draw.line(screen, (0,0,255), (BASE_X+165,BASE_Y+0),
        (BASE_X+183,BASE_Y+0), 1)
        pygame.draw.line(screen, (0,0,255), (BASE_X+183,BASE_Y+0),
        (BASE_X+183,BASE_Y+46), 1)

        #Cuadro de la T4
        pygame.draw.line(screen, (0,0,255), (BASE_X+183,BASE_Y+310),
        (BASE_X+183,BASE_Y+325), 1)
        pygame.draw.line(screen, (0,0,255), (BASE_X+165,BASE_Y+272),
        (BASE_X+18,BASE_Y+272), 1)
```

```
pygame.draw.line(screen, (0,0,255), (BASE_X+91,BASE_Y+325),
(BASE_X+215,BASE_Y+325), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+91,BASE_Y+360),
(BASE_X+325,BASE_Y+360), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+91,BASE_Y+272),
(BASE_X+91,BASE_Y+378), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+18,BASE_Y+272),
(BASE_X+18,BASE_Y+307), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+18,BASE_Y+378),
(BASE_X+18,BASE_Y+343), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+18,BASE_Y+378),
(BASE_X+307,BASE_Y+378), 1)

#Rodadura T4S
pygame.draw.line(screen, (0,0,255), (BASE_X+215,BASE_Y+240),
(BASE_X+325,BASE_Y+240), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+215,BASE_Y+240),
(BASE_X+215,BASE_Y+378), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+307,BASE_Y+187),
(BASE_X+307,BASE_Y+378), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+325,BASE_Y+230),
(BASE_X+325,BASE_Y+360), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+307,BASE_Y+187),
(BASE_X+325,BASE_Y+152), 1)

#Pista diagonal derecha
pygame.draw.line(screen, (0,0,0), (BASE_X+343,BASE_Y+297),
(BASE_X+531,BASE_Y+477), 2)
pygame.draw.line(screen, (0,0,255), (BASE_X+325,BASE_Y+282),
(BASE_X+343,BASE_Y+297), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+325,BASE_Y+307),
(BASE_X+531,BASE_Y+502), 1)
```

```
pygame.draw.line(screen, (0,0,255), (BASE_X+531,BASE_Y+502),
(BASE_X+545,BASE_Y+494), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+545,BASE_Y+494),
(BASE_X+531,BASE_Y+477), 1)

#Pista diagonal izquierda y rodadura T1,T2,T3
pygame.draw.line(screen, (0,0,0), (BASE_X+183,BASE_Y+392),
(BASE_X+375,BASE_Y+583), 2)
pygame.draw.line(screen, (0,0,255), (BASE_X+128,BASE_Y+378),
(BASE_X+265,BASE_Y+508), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+265,BASE_Y+508),
(BASE_X+265,BASE_Y+583), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+265,BASE_Y+583),
(BASE_X+375,BASE_Y+583), 1)
pygame.draw.line(screen, (0,0,255), (BASE_X+165,BASE_Y+378),
(BASE_X+183,BASE_Y+392), 1)


#T4S

pygame.draw.polygon(screen, (0,0,0), [(BASE_X+249,BASE_Y+258),
(BASE_X+254,BASE_Y+258), (BASE_X+254,BASE_Y+336), (BASE_X+249,
BASE_Y+336)], 0)
pygame.draw.polygon(screen, (0,0,0), [(BASE_X+254,BASE_Y+293),
(BASE_X+259,BASE_Y+293), (BASE_X+259,BASE_Y+300), (BASE_X+254,
BASE_Y+300)], 0)

# #T4
pygame.draw.polygon(screen, (0,0,0), [(BASE_X+0,BASE_Y+296),
(BASE_X+9,BASE_Y+296), (BASE_X+9,BASE_Y+353), (BASE_X+0,
BASE_Y+353)], 0)
pygame.draw.polygon(screen, (0,0,0), [(BASE_X+9,BASE_Y+311),
```

```
(BASE_X+37,BASE_Y+311), (BASE_X+37,BASE_Y+339), (BASE_X+9,
BASE_Y+339)], 0)
pygame.draw.polygon(screen, (0,0,0), [(BASE_X+37,BASE_Y+279),
(BASE_X+46,BASE_Y+279), (BASE_X+46,BASE_Y+371), (BASE_X+37,
BASE_Y+371)], 0)


#T1/T2/T3
pygame.draw.polygon(screen, (0,0,0), [(BASE_X+150,BASE_Y+427),
(BASE_X+156,BASE_Y+420), (BASE_X+229,BASE_Y+490),
(BASE_X+229,BASE_Y+512), (BASE_X+247,BASE_Y+540),
(BASE_X+247,BASE_Y+583), (BASE_X+237,BASE_Y+583),
(BASE_X+237,BASE_Y+544), (BASE_X+219,BASE_Y+516),
(BASE_X+219,BASE_Y+494)], 0)

#Torre
pygame.draw.polygon(screen, (0,0,0),[(BASE_X+229,BASE_Y+321),
(BASE_X+234,BASE_Y+321),(BASE_X+234,BASE_Y+328),
(BASE_X+229,BASE_Y+328)])
```

D.2. Código involucrado en el funcionamiento de una aeronave

D.2.1. Avion.py

Aquí se presenta el código troncal de las aeronaves. Se encarga de crear y mantener todos los hilos encargados de la conexión TCP con la torre de control, enviar su información en *broadcast* para el resto de aviones a través de un cliente UDP, de recibir al mismo tiempo la información de los otros aviones a través del servidor UDP, coordinar el movimiento de la aeronave y calcular la ruta que debe de seguir

```
import sys, time, threading
import RedAvion, Taxiing
sys.path.append('/home/pi/Documents/TFG/Auxiliar/')
import Plane, Aux

if __name__ == "__main__":
    """
    Esta es la función principal del avión, donde se orquestan todos
    los hilos.
    Existen 4 hilos:
        - El encargado de mantener la conexión con la torre de control
        - El encargado de emitir en broadcast su información al resto
        de aviones de la red
        - El encargado de recibir la información del resto de aviones
        de la red y de procesarla para saber si están demasiado cerca
        - El encargado del movimiento del avión, calcular las rutas que
        debe de seguir por el aeropuerto y saber si se esta desviando
    Todos los hilos tienen un parámetro stop, que sirve para indicar
    cuando deben de parar sus bucles y cerrar
    """

    Threads=[]
    twr_IP='192.168.1.160'
```

```
#Información del avión
Avion=Plane.Plane('IB',3952,'BCN','A',2,0)

#Se crean los hilos
Tower_Thread=threading.Thread(name='Tower_Thread',
target=RedAvion.towerconnection,args=(twr_IP, Avion), daemon=True)
BroadcastClient_Thread=threading.Thread(name='BroadcastClient_Thread',
target=RedAvion.broadcastclient, args=(Avion,twr_IP), daemon=True)
BroadcastServer_Thread=threading.Thread(name='BroadcastServer_Thread',
target=RedAvion.broadcastserver, args=(Avion, twr_IP), daemon=True)
Taxiing_Thread=threading.Thread(name='Taxiing_Thread',
target=Taxiing.taxiing, args=(Avion, Threads), daemon=True)

#Se inician los hilos
Tower_Thread.start()
BroadcastClient_Thread.start()
BroadcastServer_Thread.start()
Taxiing_Thread.start()

#Se añaden los hilos a una lista que luego se usara para cerrarlos
Threads.append(Tower_Thread)
Threads.append(BroadcastServer_Thread)
Threads.append(BroadcastClient_Thread)
Threads.append(Taxiing_Thread)

#Se cierran los hilos
for i in range(0,len(Threads)):
    Threads[i].join()
    print('He cerrado el hilo '+str(i+1))
```

D.2.2. RedAvion.py

El script que está debajo de estas líneas es el encargado de todas las comunicaciones de la aeronave. Tanto de la conexión TCP con la torre de control, como del servidor y cliente UDP.

```
import socket,sys, pickle, time
sys.path.append('/home/pi/Documents/TFG/Auxiliar/')
import Aux, Plane

def towerconnection(twrIP, Avion):

    #En primer lugar, se calibra el GPS. Una vez se ha hecho esto,
    #se intenta establecer una conexión con la torre de control.
    #Si se consigue, se inicia la secuencia de trabajo de este hilo.
    Avion.callibrate_gps()
    conexion=initconexion(twrIP,65000)
    first=True

    while True:
        if Avion.stop==True:
            try:
                Mensaje=rx(conexion)
                tx(conexion, Avion)
            except:
                close(conexion)
                return
        elif first==True:
            #Si es la primera vez que se intercambia un mensaje con la
            #torre, además del error, se deberá de almacenar el número
            #de conexión y la ruta a seguir por el aeropuerto.

            #El avión obtiene su posición y la almacena en los parámetros
            #del objeto avión lat y lon
```

```
        Avion.getposition()
        #Se recibe el primer mensaje desde la torre de control
        Mensaje=rx(conexion)
        #Se muestra el error recibido
        print("Error recibido="+str(Mensaje[1]))
        #Se corrige la posición del avión teniendo en cuenta el error
        #base y el error relativo recibido
        Avion.correctposition(Mensaje[1])
        #Se almacena la ruta en coordenadas UTM en el vector route
        #del objeto avión
        Avion.getroute(Mensaje[2])
        print(Avion.route)
        #Se almacena el numero de conexión con la torre
        Avion.n=Mensaje[0]
        #Se transmite toda la información del avión actualizada con
        #los datos que le acaban de llegar
        tx(conexion, Avion)
        print(Avion.n)
        first=False
    else:
        #En este tramo del bucle se repiten las operaciones anteriores
        #de recepción y transmisión, pero solo se actualiza el error
        #relativo y se corrige la posición del avión.
        Avion.getposition()
        Mensaje=rx(conexion)
        print("Error recibido="+str(Mensaje[1]))
        Avion.correctposition(Mensaje[1])
        tx(conexion, Avion)
        print(Avion.n)
```

```
def initconexion(IP,port):
    """
    Recibimos como argumentos la IP y el puerto del servidor al que queremos
    conectarnos.Se crea un socket que va probando puertos de la torre hasta
    que se da con uno que acepta la conexión o hasta que se llegue a un número
    máximo de intentos
    """
    nummaxtry=200
    numtry=0
    success=False

    #Se inicia el socket TCP con la configuración deseada
    conexion=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    conexion.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
    print ('Trying to connect...')

    #Mientras no se alcance un número máximo de intentos se intenta conectar
    #con la torre de control. Si un intento es rechazado, se avanza al siguiente
    while numtry<nummaxtry and success==False:
        try:
            conexion.connect((IP,port+numtry))
            print ('Connection success in port '+ str(port+numtry)+'!')
            success=True
        except socket.error as er:
            if numtry==nummaxtry-1:
                print ('Error al crear el socket')
                break
            else:
                numtry=numtry+1
                print(numtry)

    return conexion
```

```
def tx(conexion, Avion):
    """
    Se envía a la torre toda la información del avión en formato binario a
    través de un socket haciendo uso de la función dumps() de la librería
    pickle
    """
    conexion.send(pickle.dumps(Avion))

def rx(conexion):
    """
    Se recibe información de la torre (un máximo de 2500 bytes) y se
    devuelven del formato binario al formato original (una lista de 3
    elementos). Esta lista es devuelta para su posterior uso.
    """
    answer=conexion.recv(2500)
    return pickle.loads(answer)

def close(conexion):
    """
    Se cierra la conexión con la torre de control
    """
    conexion.close(

def broadcastclient(Avion, twr_IP):
    """
    Se crea un socket cliente UDP que emite en modo broadcast toda la
    información del avión para que el resto de aviones en la red, sepan
    su posición y su nivel de prioridad. Con esta información, se podrá
    tomar una decisión respecto que avión deja pasar al otro en caso
    de encuentro.
    """
```

```
    #Se inicia el socket UDP con la configuración deseada
    server=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
    server.bind(('',65501))

    while True:
        if Avion.stop==True:
            server.sendto(pickle.dumps(Avion),('<broadcast>',65500))
            return
        else:
            server.sendto(pickle.dumps(Avion),('<broadcast>',65500))
            time.sleep(1)

def broadcastserver(Avion, twr_IP):
    """
    Se crea un socket cliente UDP que escucha en el puerto 65501, que
    es el puerto al que se envía toda la información de los aviones en
    modo broadcast. Se encarga de almacenar esa información y determinar
    si existe riesgo de colisionar con otros aviones debido a la cercanía.
    """

    addresses=[]
    Aviones=[]
    #Se almacena la IP de la torre de control para evitar almacenar lo
    #recibido por la torre de control
    addresses.append(twr_IP)
    AvionNuevo=Plane.Plane('**',0000,'***','*',0,0)
    Aviones.append(AvionNuevo)
    #Se averigua la IP propia estableciendo una conexión momentánea con
    #el servidor DNS de Google (8.8.8.8)
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
s.connect(("8.8.8.8", 80))
myIP=s.getsockname()[0]
s.close()

#Se inicia un socket UDP en el puerto 65000
client=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
client.bind(('',65500))

while True:
    if Avion.stop==True:
        return
    else:
        data, addr = client.recvfrom(4096)
        if addr[0]!=myIP:
            if addr[0] not in addresses:
                #Si la información recibida no es de si mismo ni de
                #un avión ya registrado, se almacena y se crea un
                #nuevo objeto avión para futuros nuevos contactos
                print('Recibida informacion de un nuevo avion!!')
                addresses.append(addr[0])
                Aviones[-1]=pickle.loads(data)
                AvionNuevo = Plane.Plane('**',0000,'***','*',0,0)
                Aviones.append(AvionNuevo)
            else:
                Aviones[addresses.index(addr[0])-1]=pickle.loads(data)

            print(str(Aviones[addresses.index(addr[0])-1].name)+' '
            +str(Aviones[addresses.index(addr[0])-1].id))

        Avion.checkproximity(Aviones)
```

D.2.3. Taxiing.py

Bajo estas líneas se encuentra el código responsable del movimiento del avión durante la maniobra de rodaje. En este punto del sistema se calculan las rectas entre cada par de puntos de la ruta, si se está dentro del espacio delimitado por las rectas de tolerancia y si se ha llegado al destino deseado.

```
import sys, time
from gpiozero import Motor

def taxiing(Avion, threads):
    """
    Esta es la función encargada de todo lo relacionado con el movimiento
    del avión por el aeropuerto. Desde aquí se hacen llamadas a las
    funciones que calculan la ruta, averiguan si se esta siguiendo
    correctamente y si ya se ha llegado al punto deseado. En función de la
    información que devuelvan estas funciones, se decidirá en que sentido
    mover cada motor.
    """

    Motorizq= Motor(23,24)
    Motordcha = Motor(17,27)
    i=2

    print("Taxiing iniciado!")

    while True:
        if i<len(Avion.route)-2:
            #Mientras no se reciba una ruta se espera de segundo en
            #segundo hasta recibir una
            if Avion.checkarrival(i-1)==False:
                #Cuando se recibe una ruta, se comprueba que el avión
                #esté lo suficientemente alineado. Si no lo está, se
                #calcula un tramo hasta el punto de inicio
```

```
            Avion.route.insert(i,[Avion.lat, Avion.lon])

        print("Calculando ruta...")
        #Calculamos el primer tramo de la ruta que nos han dado
        Avion.calcularruta(i)
        print("Ruta calculada!")
        break
    else:
        time.sleep(1)

    while i<len(Avion.route)-1:
    if Avion.checkarrival(i)==True:
        #Si se entra aquí es que se ha llegado al final de la recta
        #anterior y se debe actualizar a la siguiente recta
        if i<len(Avion.route)-2:
            i=i+1
            Avion.route[1]=i
            print('Calculando proximo tramo de la ruta...')
            #Se calcula el siguiente tramo de la ruta que nos han dado
            Avion.calcularruta(i)
            print('Ruta calculada!')
        else:
            print('Maniobra de rodaje finalizada')
            Avion.stop=True
            for i in range(0,len(threads)):
                threads[i].join()
            return

    if Avion.route[0]==-1:
        #Si se ha detectado peligro, el primer elemento del vector
        #ruta se ha convertido en un -1 se para el avión y se muestra
        #un mensaje de peligro
```

```
            Motorizq.stop()
            Motordcha.stop()
            print('Peligrooooo!')
            time.sleep(1)
        else:
            #Si no se ha detectado peligro, se continua. En primer lugar,
            #se comprueba si se está siguiendo correctamente el camino que
            #se ha calculado y se decide si se continua recto o si se hace
            #alguna corrección

            instruccion=Avion.checkdirection(i)
            if instruccion==1:
                Avion.fwd(Motorizq,Motordcha)
                print('Avante tooda')
            elif instruccion==2:
                Avion.right(Motorizq,Motordcha)
                print('Derechaaa!!')
            elif instruccion==3:
                Avion.left(Motorizq,Motordcha)
                print('Izquierdaaa!!')
            else:
                Motorizq.stop()
                Motordcha.stop()

            time.sleep(1)
```

D.3. Código auxiliar

D.3.1. Clase Plane.py

Aquí se presenta la clase Plane. Es la clase en torno a la que gira todo el código desarrollado. Cada aeronave está representada por un objeto de esta clase. Este objeto incluye toda su información, como todos los métodos que hacen posible llevar a cabo cualquiera de las acciones relacionadas con un avión.

```
import sys, socket, time
from gps import *
from statistics import mean
from math import sqrt
from random import uniform
import Aux

NOVGRAD=math.pi/2 #90 grados
ALPHAMIN=math.pi/9 #20 grados
t_novgrad=0.5 #tiempo de giro para llevar a cabo 90 grados

class Plane():

    def __init__ (self, name, id, dest, txway, level, n):
        self.name=name
        self.id=id
        self.dest=dest
        self.txway=txway
        self.level=level
        self.n=0
        self.lat=0.0
        self.lon=0.0
        self.alt=0
        self.speed=0
        self.climb=0
```

```
        self.route=[0,2]
        self.rectas=[0,0,0,0]
        self.error_base=[0, 0]
        self.nummuestras=3
        self.posprev=[]
        self.stop=False

    def showinfo(self, myfont, screen):
        Aux.text(self.name+str(self.id)+"  MAD ->"+self.dest+ " Level: "
        +str(self.level),20,25*self.n+45,myfont,screen,(0,0,255))

    def callibrate_gps(self):
        """
        Método de la clase Plane.py que calibra el GPS del avión
        """
        #Se solicitan las coordenadas exactas de la aeronave
        print("Introduzca latitud del self (WGS84): ")
        self.lat=float(input())
        print("Introduzca longitud del self (WGS84): ")
        self.lon=float(input())

        #En base a esas coordenadas, se calcula el error base del sensor
        self.error_base=list(Aux.convert2UTM((self.lat, self.lon)))
        print("Coord avion reales UTM: "+str(self.error_base))

        #Se almacenan en la lista de posiciones previas la posición exacta
        #de la aeronave
        for i in range(0,self.nummuestras-1):
            self.posprev.append(self.error_base)

        self.getmeanpos()
        self.error_base=Aux.getError(self.error_base, [self.lat, self.lon])
```

```
        print("Error base del avion: "+str(self.error_base))

    def getmeanpos(self):
        """
        Método de la clase Plane.py que calcula la posición promediada
        de 10 posiciones GPS
        """

        gpsd = gps(mode=WATCH_ENABLE|WATCH_NEWSTYLE)
        latitude=[]
        longitude=[]
        i=0

        print('Calibrando GPS...')
        while i<10:
            report = gpsd.next()
            if report['class'] == 'TPV':
                latitude.append(getattr(report,'lat',0.0))
                longitude.append(getattr(report,'lon',0.0))
                i=i+1
            time.sleep(1)

        self.lat=mean(latitude)
        self.lon=mean(longitude)
        self.lat, self.lon=Aux.convert2UTM([self.lat,self.lon])
        print('Coordenadas UTM obtenidas por GPS:'+str([self.lat,self.lon]))

    def getposition(self):
        """
        Método de la clase Plane.py que obtiene una posición GPS
        """
        check=True
```

```
        gpsd = gps(mode=WATCH_ENABLE|WATCH_NEWSTYLE)

        while True:
            report = gpsd.next()
            if report['class'] == 'TPV':
                self.lat=getattr(report,'lat',0.0)
                self.lon=getattr(report,'lon',0.0)
                self.alt=getattr(report,'alt','nan')
                self.speed=getattr(report,'speed','nan')
                self.climb=getattr(report,'climb','nan')
                self.lat, self.lon=Aux.convert2UTM((self.lat,self.lon))
                print("Posicion recibida por GPS: "+"\n"+'Latitud:
                '+str(self.lat)+"\n"+'Longitud: '+str(self.lon))
                self.correctbaseerror()
                return

            time.sleep(1)

    def getroute(self, ruta):
        """
        Método de la clase Plane.py que almacena la ruta indicada por la
        torre de control
        """
        for i in range(0, len(ruta)):
            self.route.append(ruta[i])

    def correctposition(self,Error):
        """
        Método de la clase Plane.py que se encarga de las correcciones
        de la posición recibida del sensor GPS
        """
```

```
#Se corrige el error relativo recibido por la torre de control
self.lat=self.lat-Error[0]
self.lon=self.lon-Error[1]
print('Error relativo corregido: '+str([self.lat,self.lon]))

#Se aplican los coeficientes del filtro FIR seleccionado en
#función del atributo nummuestras
if self.nummuestras==3:
    self.lat=0.1*self.lat+0.8*self.posprev[0][0]+0.1*self.posprev[1][0]
    self.lon=0.1*self.lon+0.8*self.posprev[0][1]+0.1*self.posprev[1][1]
elif self.nummuestras==5:
    self.lat=-0.125*self.lat+0.175*self.posprev[0][0]+0.9*
    self.posprev[1][0]+0.175*self.posprev[2][0]-0.125*self.posprev[3][0]
    self.lon=-0.125*self.lon+0.175*self.posprev[0][1]+0.9*
    self.posprev[1][1]+0.175*self.posprev[2][1]-0.125*self.posprev[3][1]
elif self.nummuestras== 8:
    self.lat=-0.0195*self.lat+0.0715*self.posprev[0][0]-0.1765*
    self.posprev[1][0]+0.6245*self.posprev[2][0]+0.6245*self.posprev
    [3][0]-0.1765*self.posprev[4][0]+0.0715*self.posprev[5][0]-0.0195*
    self.posprev[6][0]
    self.lon=-0.0195*self.lon+0.0715*self.posprev[0][1]-0.1765*
    self.posprev[1][1]+0.6245*self.posprev[2][1]+0.6245*self.posprev
    [3][1]-0.1765*self.posprev[4][1]+0.0715*self.posprev[5][1]-0.0195*
    self.posprev[6][1]
elif self.nummuestras==10:
    self.lat=-0.012*self.lat-0.0178*self.posprev[0][0]+0.0798*
    self.posprev[1][0]-0.1754*self.posprev[2][0]+0.6254*self.posprev
    [3][0]+0.6254*self.posprev[4][0]-0.1754*self.posprev[5][0]+0.0798*
    self.posprev[6][0]-0.0178*self.posprev[7][0]-0.012*self.posprev[8][0]
    self.lon=-0.012*self.lon-0.0178*self.posprev[0][1]+0.0798*
    self.posprev[1][1]-0.1754*self.posprev[2][1]+0.6254*self.posprev
    [3][1]+0.6254*self.posprev[4][1]-0.1754*self.posprev[5][1]+0.0798*
```

```
            self.posprev[6][1]-0.0178*self.posprev[7][1]-0.012*self.posprev[8][1]

        for i in range(1,self.nummuestras-1):
            self.posprev[i]=self.posprev[i-1]

        self.posprev[0]=[self.lat,self.lon]
        print('Posición ponderada: '+str([self.lat,self.lon]))

    def correctbaseerror(self):
        """
        Método de la clase Plane.py que corrige el error base del sensor GPS
        """
        self.lat=self.lat-self.error_base[0]
        self.lon=self.lon-self.error_base[1]
        print('Error base corregido: '+str([self.lat,self.lon]))

    def calcularruta(self, i):
        """
        Método de la clase Plane.py que calcula la ecuación de la recta
        que une cada uno de los pares de puntos de la ruta y sus rectas
        paralelas
        """

        d=3 #Distancia a la que se calculan las rectas paralelas

        if self.route[i+1][0]==self.route[i][0]:
            #Caso de recta horizontal
            m=0
            n=self.route[i][0]
            if self.route[i+1][1]>self.route[i][1]:
                #Va hacia la derecha
                n_izq=self.route[i][0]+d
```

```
            n_dcha=self.route[i][0]-d
        else:
            #Va hacia la izquierda
            n_izq=self.route[i][0]-d
            n_dcha=self.route[i][0]+d
    elif self.route[i+1][1]==self.route[i][1]:
        #Caso de recta vertical
        m="inf"
        n=self.route[i][1]
        if self.route[i+1][0]>self.route[i][0]:
            #Va hacia arriba
            n_izq=self.route[i][1]-d
            n_dcha=self.route[i][1]+d
        else:
            #Va hacia abajo
            n_izq=self.route[i][1]+d
            n_dcha=self.route[i][1]-d
    else:
        #Caso de recta genérica (y=mx+n)
        delta_x=self.route[i+1][1]-self.route[i][1]
        delta_y=self.route[i+1][0]-self.route[i][0]

        m=(delta_y)/(delta_x)
        n=-m*self.route[i][1]+self.route[i][0]

        A=delta_y
        B=-delta_x
        D=A*self.route[i][1]+B*self.route[i][0]

        n_izq=(D-d*sqrt((A**2)+(B**2)))/B
        n_dcha=(D+d*sqrt((A**2)+(B**2)))/B
```

```
        self.rectas=[m,n,n_izq,n_dcha]

    def checkproximity(self, Aviones):
        """
        En este método se calcula la distancia entre la aeronave y
        el resto de aeronaves en la red. En función de la distancia
        calculada se tomará una decisión u otra. Esto se verá reflejado
        a través del parámetro self.route[0]. Si adopta un valor igual
        a -1, la aeronave deberá detenerse. Si adopta un valor igual a
        0, la aeronave podrá continuar con la maniobra de rodaje
        """

        if len(Aviones) != 1:
            for i in range(0,len(Aviones)-1):
                difflat=abs(self.lat-Aviones[i].lat)
                difflon=abs(self.lon-Aviones[i].lon)
                if difflat<0.8 and difflon<0.8:
                    print('Situacion de bloqueo')
                    self.route[0]=-1
                    return
                elif difflat<1.5 and difflon<1.5:
                    self.route[0]=-1
                    if self.level<Aviones[i].level:
                        print('Tengo prioridad')
                        self.route[0]=0
                    elif self.level == Aviones[i].level and
                    self.n < Aviones[i].n:
                        print('Tengo prioridad')
                        self.route[0]=0
                    else:
                        print('Tengo que ceder la prioridad')
                elif difflat<3 and difflon<3:
```

```
                    if self.level < Aviones[i].level:
                        print('Tengo que ceder la prioridad')
                        self.route[0]=-1
                        return
                    elif self.level == Aviones[i].level and
                    self.n > Aviones[i].n:
                        print('Tengo que ceder la prioridad')
                        self.route[0]=-1
                        return
                    else:
                        print('Tengo prioridad')

        self.route[0]=0

    def checkdirection(self, i):
        """
        Método de la clase Plane.py que calcula si el avión se encuentra
        dentro del límite marcado por las rectas paralelas de tolerancia.
        En función de si se encuentra dentro o fuera, y del lado por el
        que se esté saliendo, se corregirá hacia un lado u otro el movimiento
        del avión.
        """

        if self.rectas[0]==0:
            #Caso de recta horizontal
            if (self.lat>self.rectas[2] and self.route[i][1]<self.
            route[i+1][1]) or (self.lat<self.rectas[2] and
            self.route[i][1]>self.route[i+1][1]):
                #Derecha
                return 2
            elif (self.lat<self.rectas[3] and self.route[i][1]<self.
            route[i+1][1]) or (self.lat>self.rectas[3] and
```

```
            self.route[i][1]>self.route[i+1][1]):
                #Izquierda
                return 3
            else:
                #Recto
                return 1
        elif self.rectas[0]=="inf":
            #Caso de recta vertical
            if (self.lon<self.rectas[2] and self.route[i][0]<self.
            route[i+1][0]) or (self.lon>self.rectas[2] and
            self.route[i][0]>self.route[i+1][0]):
                #Derecha
                return 2
            elif (self.lon>self.rectas[3] and self.
            route[i][0]<self.route[i+1][0]) or (self.lon<self.rectas[3] and
            self.route[i][0]>self.route[i+1][0]):
                #Izquierda
                return 3
            else:
                #Recto
                return 1
        else:
            #Resto de casos de self.rectas diagonales
            if (self.lat>(self.rectas[0]*self.lon+self.rectas[2]) and
            self.route[i][1]<self.route[i+1][1]) or
            (self.lat<(self.rectas[0]*self.lon+self.rectas[2]) and
            self.route[i][1]>self.route[i+1][1]):
                #A la derecha
                return 2

            elif (self.lat<(self.rectas[0]*self.lon+self.rectas[3]) and
            self.route[i][1]<self.route[i+1][1]) or
```

```
            (self.lat>(self.rectas[0]*self.lon+self.rectas[3]) and
            self.route[i][1]>self.route[i+1][1]):
                #A la izquierda
                return 3
            else:
                #Recto
                return 1

    def calculogiro(self,Avion, m_anterior,i, Motorizq, Motordcha):
        """
        Método de la clase Plane.py que calcula el giro que debe realizar
        una aeronave al llegar a un punto de la ruta, para alinearse
        con el siguiente tramo de la misma. Además calcula hacia
        qué lado debe girar.
        """

        if m_anterior==0:
            if (Avion.route[i][0]<Avion.route[i+1][0] and
            Avion.route[i-1][1]<Avion.route[i][1]) or
            (Avion.route[i][0]>Avion.route[i+1][0] and
            Avion.route[i-1][1]>Avion.route[i][1]):
                if Avion.rectas[0]=="inf":
                    print('90º izquierda!')
                    self.left_turn(Motorizq,Motordcha,t_novgrad)
                else:
                    alpha=abs(math.atan(Avion.rectas[0]))
                    print(str(math.degrees(alpha))+' º a la izquierda!')
                    if alpha > ALPHAMIN:
                        self.left_turn(Motorizq,Motordcha,alpha*
                        t_novgrad/NOVGRAD)
            elif (Avion.route[i][0]>Avion.route[i+1][0] and
            Avion.route[i-1][1]<Avion.route[i][1]) or
```

```
        (Avion.route[i][0]<Avion.route[i+1][0] and
        Avion.route[i-1][1]>Avion.route[i][1]):
            if Avion.rectas[0]=="inf":
                print('90º derecha!')
                self.rigth_turn(Motorizq,Motordcha,t_novgrad)
            else:
                alpha=abs(math.atan(Avion.rectas[0]))
                print(str(math.degrees(alpha))+' º a la derecha!')
                if alpha > ALPHAMIN:
                    self.rigth_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)
    elif m_anterior=="inf":
        if (Avion.route[i][1]<Avion.route[i+1][1] and
        Avion.route[i-1][0]<Avion.route[i][0]) or
        (Avion.route[i][1]>Avion.route[i+1][1] and
        Avion.route[i-1][0]>Avion.route[i][0]):
            if Avion.rectas[0]==0:
                print('90º derecha!')
                self.rigth_turn(Motorizq,Motordcha,t_novgrad)
            else:
                alpha=NOVGRAD-math.atan(Avion.rectas[0])
                print(str(math.degrees(alpha))+' º a la derecha!')
                if alpha > ALPHAMIN:
                    self.rigth_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)
        elif (Avion.route[i][1]>Avion.route[i+1][1] and
        Avion.route[i-1][0]<Avion.route[i][0]) or
        (Avion.route[i][1]<Avion.route[i+1][1] and
        Avion.route[i-1][0]>Avion.route[i][0]):
            if Avion.rectas[0]==0:
                print('90º izquierda!')
                self.left_turn(Motorizq,Motordcha,t_novgrad)
```

```
            else:
                alpha=NOVGRAD+math.atan(Avion.rectas[0])
                print(str(math.degrees(alpha))+' º a la izquierda!')
                if alpha > ALPHAMIN:
                    self.left_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)
    else:
        if Avion.rectas[0]!="inf":
            delta_m=Avion.rectas[0]-m_anterior
            alpha=abs(math.atan(delta_m))
            if alpha<ALPHAMIN:
                return
        if Avion.rectas[0]=="inf":
            alpha=math.atan(m_anterior)
            print(math.degrees(alpha))
            if abs(alpha)<ALPHAMIN:
                return

            if (Avion.route[i-1][1]>Avion.route[i][1] and
            Avion.route[i][0]>Avion.route[i+1][0]):
                #Vengo por la izquierda y quiero bajar
                #Giro a izquierdas
                alpha=(math.pi/2)-alpha
                print(str(math.degrees(alpha))+' º a la izquierda!')
                self.left_turn(Motorizq,Motordcha,alpha*
                t_novgrad/NOVGRAD)
            elif (Avion.route[i-1][1]>Avion.route[i][1] and
            Avion.route[i][0]<Avion.route[i+1][0]):
                #Vengo por la izquierda y quiero subir
                #Giro a derechas
                alpha=(math.pi/2)+alpha
                print(str(math.degrees(alpha))+' º a la derecha!')
```

```
                self.rigth_turn(Motorizq,Motordcha,alpha*
                t_novgrad/NOVGRAD)
            elif (Avion.route[i-1][1]<Avion.route[i][1] and
            Avion.route[i][0]>Avion.route[i+1][0]):
                #Vengo por la derecha y quiero bajar
                alpha=(math.pi/2)-alpha
                print(str(math.degrees(alpha))+' º a la derecha!')
                self.rigth_turn(Motorizq,Motordcha,alpha*
                t_novgrad/NOVGRAD)
            elif  (Avion.route[i-1][1]<Avion.route[i][1] and
            Avion.route[i][0]<Avion.route[i+1][0]):
                #Vengo por la derecha y quiero bajar
                #Giro a izquierda
                alpha=(math.pi/2)+alpha
                print(str(math.degrees(alpha))+' º a la izquierda!')
                self.left_turn(Motorizq,Motordcha,alpha*
                t_novgrad/NOVGRAD

        elif m_anterior>0:
            if Avion.rectas[0]>0:
                if delta_m>0:
                    #Giro a izquierdas
                    print(str(math.degrees(alpha))+' º a la izquierda!')
                    self.left_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)
                elif delta_m<0:
                    #Giro a derechas
                    print(str(math.degrees(alpha))+' º a la derecha!')
                    self.rigth_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)
            else:
                if (Avion.route[i][1]<Avion.route[i+1][1] and
```

```
                Avion.route[i-1][1]<Avion.route[i][1]) or
                ( Avion.route[i][1]>Avion.route[i+1][1] and
                Avion.route[i-1][1]>Avion.route[i][1]):
                #Giro a derechas
                    print(str(math.degrees(alpha))+' º a la derecha!')
                    self.rigth_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)
                else:
                    alpha=math.pi-alpha
                    #Giro a izquierdas
                    print(str(math.degrees(alpha))+' º a la izquierda!')
                    self.left_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)
    else:
        if Avion.rectas[0]<0:
            if delta_m>0:
                #Giro a izquierdas
                print(str(math.degrees(alpha))+' º a la izquierda!')
                self.left_turn(Motorizq,Motordcha,alpha*
                t_novgrad/NOVGRAD)
            elif delta_m<0:
                #Giro a derechas
                print(str(math.degrees(alpha))+' º a la derecha!')
                self.rigth_turn(Motorizq,Motordcha,alpha*
                t_novgrad/NOVGRAD)
        else:
            if (Avion.route[i][1]<Avion.route[i+1][1] and
            Avion.route[i-1][1]<Avion.route[i][1]) or
            ( Avion.route[i][1]>Avion.route[i+1][1] and
            Avion.route[i-1][1]>Avion.route[i][1]):
                #Giro a izquierdas
                print(str(math.degrees(alpha))+' º a la izquierda!')
```

```
                    self.left_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)
                else:
                    alpha=math.pi-alpha
                    #Giro a derechas
                    print(str(math.degrees(alpha))+' º a la derecha!')
                    self.rigth_turn(Motorizq,Motordcha,alpha*
                    t_novgrad/NOVGRAD)

def checkarrival(self, i):
    """
    Método de la clase Plane.py que calcula se si ha llegado ya al punto
    de la ruta deseado
    """

    #Si se trata del primer tramo de la ruta, se da una mayor tolerancia
    if i==1:
        if sqrt(((self.lat-self.route[i+1][0])**2)+((self.lon
        -self.route[i+1][1])**2))<=3:
            return True
        else:
            return False
    elif sqrt(((self.lat-self.route[i+1][0])**2)+((self.lon
    -self.route[i+1][1])**2))<=1.5:
        return True
    else:
        return False

    def simtaxiing(self, i, accion):
        """
    Método de la clase Plane.py que simula el movimiento de un avión
    realizando una maniobra de rodaje siguiendo las ecuaciones de
```

```
las rectas calculadas a una velocidad constante
"""

vel=0.4 #m/s -> 1.44 km/h (a escala) || 20 knots (nudos) = 37 km/h

if self.rectas[0]==0 and self.route[i][1]<self.route[i+1][1]:
    #Caso de recta horizontal hacia la derecha
    if accion==1: # Recto
        self.lat=self.lat
        self.lon=self.lon+vel
    elif accion==2: #Corregir hacia la derecha
        #Se esta moviendo hacia x positivas y hay que corregir
        #hacia abajo
        self.lat=self.lat-vel
        self.lon=self.lon+vel
    elif accion==3:
        #Se esta moviendo hacia x positivas y hay que corregir
        #hacia arriba
        self.lat=self.lat+vel
        self.lon=self.lon+vel
elif self.rectas[0]==0 and self.route[i][1]>self.route[i+1][1]:
    #Caso de recta horizontal hacia la izquierda
    if accion==1: # Recto
        self.lat=self.rectas[1]
        self.lon=self.lon-vel
    elif accion==2: #Corregir hacia la derecha
        #Se esta moviendo hacia x negativas y hay que corregir
        #hacia arriba
        self.lat=self.lat+vel
        self.lon=self.lon-vel
    elif accion==3:
        #Se esta moviendo hacia x negativas y hay que corregir
```

```
            #hacia abajo
            self.lat=self.lat-vel
            self.lon=self.lon-vel
        elif self.rectas[0]=="inf" and self.route[i][0]<self.route[i+1][0]:
            #Caso de recta vertical hacia arriba
            if accion==1: # Recto
                self.lat=self.lat+vel
                self.lon=self.lon
            elif accion==2: #Corregir hacia la derecha
                #Se esta moviendo hacia y positivas y hay que corregir
                #hacia la derecha
                self.lat=self.lat+vel
                self.lon=self.lon+vel
            elif accion==3:
                #Se esta moviendo hacia y positivas y hay que corregir
                #hacia la izquierda
                self.lat=self.lat+vel
                self.lon=self.lon-vel
        elif self.rectas[0]=="inf" and self.route[i][0]>self.route[i+1][0]:
            #Caso de recta vertical hacia abajo
            if accion==1: # Recto
                self.lat=self.lat-vel
                self.lon=self.lon
            elif accion==2: #Corregir hacia la derecha
                #Se esta moviendo hacia y negativas y hay que corregir
                #hacia la izquierda
                self.lat=self.lat-vel
                self.lon=self.lon-vel
            elif accion==3:
                #Se esta moviendo hacia y negativas y hay que corregir
                #hacia la derecha
                self.lat=self.lat-vel
```

```
            self.lon=self.lon+vel
    else:
        b_1=+sqrt((vel**2)/((self.rectas[0]**2)+1))
        b_2=-sqrt((vel**2)/((self.rectas[0]**2)+1))

        if self.route[i][1]<self.route[i+1][1]:
            #Caso de recta diagonal hacia x positivas
            self.lon=self.lon+b_1
            if accion==1: # Recto
                self.lat=self.rectas[0]*self.lon+self.rectas[1]
            elif accion==2: #Corregir hacia la derecha
                #Se esta moviendo hacia x positivas y hay que corregir
                #hacia abajo
                self.lat=self.rectas[0]*self.lon+self.rectas[1]-vel
            elif accion==3:
                #Se esta moviendo hacia x positivas y hay que corregir
                #hacia arriba
                self.lat=self.rectas[0]*self.lon+self.rectas[1]+vel
        elif self.route[i][1]>self.route[i+1][1]:
            #Caso de recta diagonal hacia x negativas
            self.lon=self.lon+b_2
            if accion==1: # Recto
                self.lat=self.rectas[0]*self.lon+self.rectas[1]
            elif accion==2: #Corregir hacia la derecha
                #Se esta moviendo hacia x negativas y hay que corregir
                #hacia arriba
                self.lat=self.rectas[0]*self.lon+self.rectas[1]+vel
            elif accion==3:
                #Se esta moviendo hacia x negativas y hay que corregir
                #hacia abajo
                self.lat=self.rectas[0]*self.lon+self.rectas[1]-vel
```

```
#Instrucciones de control de los motores
def fwd(self,Motorizq,Motordcha):
    Motorizq.forward(1)
    Motordcha.forward(1)
def bckwd(self,Motorizq,Motordcha):
    Motorizq.backward(1)
    Motordcha.backward(1)
def left(self,Motorizq, Motordcha):
    Motorizq.forward(0.5)
    Motordcha.forward(1)
def right(self,Motorizq, Motordcha):
    Motorizq.forward(1)
    Motordcha.forward(0.5)
def left_turn(self,Motorizq,Motordcha,tiempo):
    Motorizq.backward(1)
    Motordcha.forward(1)
    time.sleep(tiempo)
    Motorizq.stop()
    Motordcha.stop()
def rigth_turn(self, Motorizq,Motordcha,tiempo):
    Motorizq.forward(1)
    Motordcha.backward(1)
    time.sleep(tiempo)
    Motorizq.stop()
    Motordcha.stop()
```

D.3.2. Aux.py

Por último, se muestra el código elaborado para las funciones denominadas "auxiliares" entre las que se halla la conversión de coordenadas, actualización de la interfaz o calibración del mapa, entre otras.

```
import pygame, sys, time, utm
from gps import *
from statistics import mean
sys.path.append('/home/alejandro/Documentos/TFG/Torre')
sys.path.append('/home/pi/Documents/TFG/Torre')
import Interface


LONG_LAT=34 #Longitud vertical del mapa (metros)
LONG_LON=24 #Longitud horizontal del mapa (metros)
LONG_MAP_LAT=600 #Número de píxeles verticales del mapa
LONG_MAP_LON=550 #Número de píxeles horizontales del mapa
CTE_MAP_LAT=LONG_LAT/LONG_MAP_LAT #Relación m/píxel en el eje vertical
CTE_MAP_LON=LONG_LON/LONG_MAP_LON #Relación m/píxel en el eje horizontal
BASE_X=425 #Desplazamiento base en el eje horizontal (píxeles)
BASE_Y=25 #Desplazamiento base en el eje vertical (píxeles)


def text(text, x, y, myfont, screen, color):
    """
    Función que se encarga de mostrar texto en una posición concreta
    de la pantalla de pygame que se esté utilizando
    """

    texto=myfont.render(text, False, color)
    screen.blit(texto,(x,y))

def convert2UTM(coordenatesGeo):
```

```
    """
    Conversor de coordenadas geográficas decimales a coordenadas UTM
    utilizando las constantes calculadas a fin de simplificar los cálculos
    """

    UTMcoord=utm.from_latlon(coordenatesGeo[0], coordenatesGeo[1])
    return (UTMcoord[1], UTMcoord[0])

def convert2geo(coordenatesUTM):
    """
    Conversor de coordenadas UTM a coordenadas geográficas decimales
    utilizando las constantes calculadas a fin de simplificar los cálculos
    """

    latloncoord=utm.to_latlon(coordenatesUTM[1], coordenatesUTM[0])
    return (latloncoord[0],latloncoord[1])

def callibratemap(error_base, tower_pos):
    """
    Función que calibra el mapa sobre el que se va a trabajar.
    Calcula la posición de la torre de control (UTM_0)
    y en base a esa posición, establece las coordenadas de las
    esquinas del mapa.

    La primera es la superior izquierda (UTM_1) y continua en el
    sentido de las agujas del reloj hasta terminar en UTM_4

    Se trabaja con coordenadas UTM ya que permiten trabajar
    directamente con metros.

    Utilizando las constantes LONG_LAT y LONG_LON podemos variar el
    tamaño del mapa cambiando su valor. El resto del programa
```

```
    esta hecho con proporciones para que el cambio se inmediato.
    """
    cornercoord=[]

    print("Introduzca latitud de la torre (WGS84): ")
    tower_pos[0]=float(input())
    print("Introduzca longitud de la torre (WGS84): ")
    tower_pos[1]=float(input())

    UTM_0=convert2UTM(tower_pos)
    print('Coordenadas UTM reales torre:'+"\n"+'Latitud: '
    +str(UTM_0[0])+"\n"+'Longitud: '+str(UTM_0[1]))

    tower_pos=gettowerpos()
    error_base=getError(UTM_0, tower_pos)
    print("Error base torre: "+str(error_base))

    UTM_1=(UTM_0[0]+(9/17)*LONG_LAT,UTM_0[1]-(5/12)*LONG_LON)
    UTM_2=(UTM_1[0],UTM_1[1]+LONG_LON)
    UTM_3=(UTM_2[0]-LONG_LAT,UTM_2[1])
    UTM_4=(UTM_3[0],UTM_3[1]-LONG_LON)

    cornercoord.append(UTM_0)
    cornercoord.append(UTM_1)
    cornercoord.append(UTM_2)
    cornercoord.append(UTM_3)
    cornercoord.append(UTM_4)

    return (cornercoord, tower_pos)

def showinmap(ref_coordenates, coordenadas, myfont,screen):
    """
```

```
    Función que a partir de unas coordenadas de referencia y unas
    coordenadas (ambas en UTM), hace una llamada a la función
    text para que muestre con un * la posición indicada.

    Hace un ajuste de las coordenadas al mapa, utilizando las de
    referencia y las constantes del mapa en cada una
    de las direcciones (latitud y longitud)
    """

    text('*',((coordenadas[1]-ref_coordenates[1])/CTE_MAP_LON)+BASE_X,
    ((ref_coordenates[0]-coordenadas[0])/CTE_MAP_LAT)+BASE_Y,myfont,
    screen,(0,0,255))

def updatescreen(Aviones, ref_coordenates, myfont, screen):
     """
    Función que a partir de unas coordenadas de referencia
    muestra por la interfaz toda la información del objeto
    avión recibido y lo posiciona en el mapa junto con sus rectas calculadas
    """
    screen.fill((255,255,255))
    text("Aviones en rodadura", 55,5, myfont, screen,(0,0,255))
    font=pygame.font.SysFont('Arial',10)
    Interface.draw_interface(screen)

    if Aviones[-1].n==0:
        for i in range(0,len(Aviones)-1):
            if Aviones[i].stop==False:
                Aviones[i].showinfo(myfont,screen)
                printrectas(Aviones[i],ref_coordenates,myfont,screen)
                dibujaravion(Aviones[i].name,Aviones[i].id,ref_coordenates,
                (Aviones[i].lat,Aviones[i].lon),font, screen)
```

```
    else:
        for i in range(0,len(Aviones)):
            if Aviones[i].stop==False:
                Aviones[i].showinfo(myfont,screen)
                printrectas(Aviones[i],ref_coordenates,font,screen)
                dibujaravion(Aviones[i].name,Aviones[i].id,ref_coordenates,
                (Aviones[i].lat,Aviones[i].lon),font, screen)

    pygame.display.flip()

def gettowerpos():
    """
    Función con la que obtenemos la posición de la torre promediando las primeras

    Sirva de precedente para el resto del programa, el primer
    elemento de las coordenadas sera siempre la latitud y el
    segundo elemento la longitud, independientemente de si son
    coordenadas geográficas o UTM.
    """

    gpsd = gps(mode=WATCH_ENABLE|WATCH_NEWSTYLE)
    latitude=[]
    longitude=[]
    i=0

    print('Calibrando mapa ...')
    while i<10:
        report = gpsd.next()
        if report['class'] == 'TPV':
            latitude.append(getattr(report,'lat',0.0))
            longitude.append(getattr(report,'lon',0.0))
            i=i+1
```

```
            time.sleep(1)

        final_lat=mean(latitude)
        final_lon=mean(longitude)

        tower_position=convert2UTM([final_lat,final_lon])
        print('Coordenadas UTM obtenidas por GPS:'+"\n"+'Latitud: '
        +str(tower_position[0])+"\n"+'Longitud:'+str(tower_position[1]))

        return tower_position

    def getError(knwn_coord, gps_coord):
        """
            Función encargada de calcular el error percibido por la torre
            entre la posición obtenida mediante el promedio de 10 muestras
            y la posición calculada en cada nueva iteración del bucle
            """
            return[gps_coord[0]-knwn_coord[0],gps_coord[1]-knwn_coord[1]]

    def printrectas(Avion, ref_coordenates, myfont, screen):
        """
        Función encargada de representar en la interfaz las rectas
        enviadas por cada uno de los aviones en el sistema. En verde
        se representan la recta principal y a cada lado, en rojo, las
        recta de tolerancia
        """

        j=Avion.route[1]
        #Para ver si se ha calculado bien la recta, se coge el valor de x
        #que debería de tener y se calcula el valor de y a partir de los
        #parámetros de la recta calculada
```

```
x_recta=Avion.route[j+1][1]

if Avion.rectas[0]==0:
    #Caso de recta horizontal
    y_recta=Avion.rectas[0]*x_recta+Avion.rectas[1] #y=n
    x_izq=x_recta
    x_dcha=x_recta
    y_izq=Avion.rectas[2]
    y_dcha=Avion.rectas[3]
    pygame.draw.line(screen,(255,0,0),(((Avion.route[j][1]
    -ref_coordenates[1])/CTE_MAP_LON)+BASE_X,((ref_coordenates[0]
    -y_izq)/CTE_MAP_LAT)+BASE_Y),(((x_izq-ref_coordenates[1])/CTE_MAP_LON)
    +BASE_X,((ref_coordenates[0]-y_izq)/CTE_MAP_LAT)+BASE_Y),1)

    pygame.draw.line(screen,(255,0,0),(((Avion.route[j][1]
    -ref_coordenates[1])/CTE_MAP_LON)+BASE_X,((ref_coordenates[0]
    -y_dcha)/CTE_MAP_LAT)+BASE_Y),(((x_dcha-ref_coordenates[1])/CTE_MAP_LON)
    +BASE_X,((ref_coordenates[0]-y_dcha)/CTE_MAP_LAT)+BASE_Y),1)
elif Avion.rectas[0]=="inf":
    #Caso de recta vertical
    y_recta=Avion.route[j+1][0]
    x_izq=Avion.rectas[2]
    x_dcha=Avion.rectas[3]
    y_izq=y_recta
    y_dcha=y_recta
    pygame.draw.line(screen,(255,0,0),(((x_izq-ref_coordenates[1])
    /CTE_MAP_LON)+BASE_X,((ref_coordenates[0]-Avion.route[j][0])
    /CTE_MAP_LAT)+BASE_Y),(((x_izq-ref_coordenates[1])/CTE_MAP_LON)
    +BASE_X,((ref_coordenates[0]-y_izq)/CTE_MAP_LAT)+BASE_Y),1)
    pygame.draw.line(screen,(255,0,0),(((x_dcha-ref_coordenates[1])
    /CTE_MAP_LON)+BASE_X,((ref_coordenates[0]-Avion.route[j][0])
```

```
            /CTE_MAP_LAT)+BASE_Y),(((x_dcha-ref_coordenates[1])/CTE_MAP_LON)
            +BASE_X,((ref_coordenates[0]-y_dcha)/CTE_MAP_LAT)+BASE_Y),1)
        else:
            y_recta=Avion.rectas[0]*x_recta+Avion.rectas[1] #y=mx+n
            y_izq_1=Avion.rectas[0]*Avion.route[j][1]+Avion.rectas[2]
            y_izq_2=Avion.rectas[0]*x_recta+Avion.rectas[2] #y_izq=mx+n_izq
            y_dcha_1=Avion.rectas[0]*Avion.route[j][1]+Avion.rectas[3]
            y_dcha_2=Avion.rectas[0]*x_recta+Avion.rectas[3] #y_izq=mx+n_dcha
            pygame.draw.line(screen,(255,0,0),(((Avion.route[j][1]
            -ref_coordenates[1])/CTE_MAP_LON)+BASE_X,
            ((ref_coordenates[0]-y_izq_1)/CTE_MAP_LAT)+BASE_Y),
            (((x_recta-ref_coordenates[1])/CTE_MAP_LON)+BASE_X,
            ((ref_coordenates[0]-y_izq_2)/CTE_MAP_LAT)+BASE_Y),1)
            pygame.draw.line(screen,(255,0,0),(((Avion.route[j][1]
            -ref_coordenates[1])/CTE_MAP_LON)+BASE_X,
            ((ref_coordenates[0]-y_dcha_1)/CTE_MAP_LAT)+BASE_Y),
            (((x_recta-ref_coordenates[1])/CTE_MAP_LON)+BASE_X,
            ((ref_coordenates[0]-y_dcha_2)/CTE_MAP_LAT)+BASE_Y),1)

        pygame.draw.line(screen,(0,255,0),(((Avion.route[j][1]
        -ref_coordenates[1])/CTE_MAP_LON)+BASE_X,((ref_coordenates[0]
        -Avion.route[j][0])/CTE_MAP_LAT)+BASE_Y),(((x_recta
        -ref_coordenates[1])/CTE_MAP_LON)+BASE_X,((ref_coordenates[0]
        -y_recta)/CTE_MAP_LAT)+BASE_Y),1)

    def dibujaravion(name,identificador,ref_coordenates, coordenadas, font, screen):
        """
        Función encargada de realizar el símbolo de un avión en la
        posición en la que encuentre en la interfaz e indicar su nombre
        e identificador con una etiqueta arriba a la izquierda del mismo
        """
```

```
        text(str(name + ' '+str(identificador)), ((coordenadas[1]
        -ref_coordenates[1])/CTE_MAP_LON)+BASE_X+7,((ref_coordenates[0]
        -coordenadas[0])/CTE_MAP_LAT)+BASE_Y-12, font, screen,(0,0,255))
        pygame.draw.line(screen,(0,0,255),(((coordenadas[1]
        -ref_coordenates[1])/CTE_MAP_LON)+BASE_X-4,
        ((ref_coordenates[0]-coordenadas[0])/CTE_MAP_LAT)+BASE_Y),
        (((coordenadas[1]-ref_coordenates[1])/CTE_MAP_LON)+BASE_X+4,
        ((ref_coordenates[0]-coordenadas[0])/CTE_MAP_LAT)+BASE_Y),2)
        pygame.draw.line(screen,(0,0,255),(((coordenadas[1]
        -ref_coordenates[1])/CTE_MAP_LON)+BASE_X,((ref_coordenates[0]
        -coordenadas[0])/CTE_MAP_LAT)+BASE_Y-4),(((coordenadas[1]
        -ref_coordenates[1])/CTE_MAP_LON)+BASE_X,((ref_coordenates[0]
        -coordenadas[0])/CTE_MAP_LAT)+BASE_Y+6),2)
        pygame.draw.line(screen,(0,0,255),(((coordenadas[1]
        -ref_coordenates[1])/CTE_MAP_LON)+BASE_X-2,((ref_coordenates[0]
        -coordenadas[0])/CTE_MAP_LAT)+BASE_Y+3),(((coordenadas[1]
        -ref_coordenates[1])/CTE_MAP_LON)+BASE_X+3,((ref_coordenates[0]
        -coordenadas[0])/CTE_MAP_LAT)+BASE_Y+3),2)

def crearruta(ref_coordenates):
    """
    Función encargada de recoger las indicaciones del controlador
    aéreo respecto a la ruta que debe tomar un avión por el
    aeropuerto y materializarla en una lista de listas con los
    pares de coordenadas UTM de cada uno de los puntos por los que
    debe de pasar el avión.
    """

    ruta=[]
    check=True
```

```
latitud={
    'A':0,
    'B':152,
    'C':187,
    'D':240,
    'E':272,
    'F':282,
    'G':307,
    'H':325,
    'I':343,
    'J':360,
    'K':378,
    'L':494,
    'M':502,
    'N':508,
    'O':583,
}

longitud={
    'A':18,
    'B':91,
    'C':128,
    'D':165,
    'E':183,
    'F':215,
    'G':265,
    'H':307,
    'I':325,
    'J':531,
    'K':545,
}
```

```
runways={
    'RW14L':[297,343],
    'RW14R':[392,183],
    'RW18L':[0,325],
    'RW18R':[46,183],
    'RW32L':[583,375],
    'RW32R':[477,531],
    'RW36L':[310,183],
    'RW36R':[230,325],
}

print("Introduzca el punto de partida (lat,lon): ")
posicion=str(input())
posicion.upper()
ruta.append([posicion[0],posicion[2]])

while True:
    if check==True:
        print("Introduzca un punto de referencia (lat,lon): ")
        posicion=str(input())
        posicion.upper()

        if posicion =="FIN":
            return [0,0]
        elif len(posicion)>3:
            ruta.append(posicion)
        else:
            ruta.append([posicion[0],posicion[2]])
        check=False
    else:
        print("Introduzca un punto de referencia (lat,lon): ")
        print("Si desea que el anterior punto sea el final de
```

```
            la ruta introduzca FIN")
            posicion=str(input())
            posicion.upper()

            if posicion =="FIN":
                break
            elif len(posicion)>3:
                ruta.append(posicion)
            else:
                ruta.append([posicion[0],posicion[2]])

    for i in range(0, len(ruta)):
        try:
            if len(ruta[i])>3:
                ruta[i]=runways.get(ruta[i])
            else:
                ruta[i][0]=latitud.get(ruta[i][0])
                ruta[i][1]=longitud.get(ruta[i][1])
        except:
            print(ruta[i])

        #Se transforman las coordenadas desde el sistema de
        #coordenadas del mapa a UTM
        ruta[i][0]=-((ruta[i][0])*CTE_MAP_LAT-ref_coordenates[0])
        ruta[i][1]=(ruta[i][1])*CTE_MAP_LON+ref_coordenates[1]

    print(ruta)

    return ruta
```

Bibliografía

[1] L. Bonenberg, *Closely-coupled Integration of Locata and GPS for Engineering Applications*. PhD thesis, Nottingham University, 01 2014.

[2] G. Xu and Y. Xu, *GPS: theory, algorithms and applications*. Springer, 2016.

[3] P. Teunissen and O. Montenbruck, *Springer handbook of global navigation satellite systems*. Springer, 2017.

[4] GMV, "Gnss augmentation." `https://gssc.esa.int/navipedia/index.php/GNSS_Augmentation`, 2011. Accedido por última vez el 2020-05-18.

[5] NHTSA, "The evolution of automated safety technologies." `https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety#topic-road-self-driving`. Accedido por última vez el 2020-06-29.

[6] Tesla, "Autopilot." `https://www.tesla.com/es_ES/autopilot`. Accedido por última vez el 2020-06-29.

[7] realpython.com, "Official online python courses." `https://realpython.com`, 2014. Accedido por última vez el 2020-05-23.

[8] Y. Kouba, "Cours de système d'information géographique." `https://www.researchgate.net/publication/324149696_Cours_de_systeme_d'information_geographique`. Published in April 2018.

[9] Stemedu, "Especificaciones técnicas módulo gnss y antena." `https://www.amazon.es/GLONASS-Compatible-Arduino-Raspberry-Pixhawk/dp/B07LC84JLM/ref=sr_1_12?__mk_es_ES=%C3%85M%C3%85%C5%BD%C3%95%C3%91&keywords=neo+8m+gps&qid=1570647982&s=electronics&sr=1-12`. Accedido por última vez el 2020-06-26.

[10] Lenovo, "Especificaciones técnicas lenovo ideapad 300." `https://www.lenovo.com/py/es/laptops/ideapad/serie-300/Ideapad-300-15/p/88IP3000675`. Accedido por última vez el 2020-06-26.

[11] STMicroelectronics, "Especificaciones técnicas l298n." https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf. Accedido por última vez el 2020-06-26.

[12] RAVPOWER, "Especificaciones técnicas batería portátil." https://www.amazon.es/Li-Pol%C3%ADmero-RAVPower-Inteligente-resistente-Smartphone/dp/B01FSFA7GS/ref=sr_1_fkmr1_1?__mk_es_ES=%C3%85M%C3%85%C5%BD%C3%95%C3%91&crid=26YRIO7NUOSOQ&dchild=1&keywords=banco+de+bateria+coolreall+22000&qid=1593172873&s=electronics&sprefix=banco+de+%2Celectronics%2C176&sr=1-1-fkmr1. Accedido por última vez el 2020-06-26.

[13] R. P. Foundation, "Especificaciones técnicas raspberry pi 3b+." https://magpi.raspberrypi.org/articles/raspberry-pi-3bplus-specs-benchmarks. Accedido por última vez el 2020-06-26.

[14] I. G. Petrovski, *GPS, GLONASS, Galileo, and BeiDou for mobile devices: from instant to precise positioning*. Cambridge University Press, 2014.

[15] GMV, "Precise point positioning systems." https://gssc.esa.int/navipedia/index.php/PPP_Systems, 2011. Accedido por última vez el 2020-06-15.

[16] I. Moir, A. Seabridge, and M. Jukes, *Civil avionics systems*. John Wiley & Sons, 2013.

[17] D. Stacey, *Aeronautical radio communication systems and networks*. Wiley Online Library, 2008.

[18] ICAO, *Guide for Ground Based Augmentation System Implementation*. ICAO, 2013.

[19] Nvidia, "Self-driving cars." https://www.nvidia.com/en-us/self-driving-cars/. Accedido por última vez el 2020-06-29.

[20] P. Foundation, "Python official documentation web." https://docs.python.org/3.6/, 2016. Accedido por última vez el 2020-05-23.

[21] T. Bieniek, "Bidirectional utm-wgs84 converter for python." https://pypi.org/project/utm, 2019. MIT Licensed. Accedido por última vez el 2020-05-31.

[22] P. github organization, "Pygame library official documentation web." https://www.pygame.org, 2010. Accedido por última vez el 2020-05-23.