Universitat de les Illes Balears

escola politècnica superior

Treball Final de Grau

GRADO DE INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Control de posición y navegación de un robot aéreo de bajo coste mediante un sistema de captura de movimiento

MARC POZO PÉREZ

Tutores

Francisco Bonnín Pascual

Alberto Ortiz Rodríguez

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, 7 de septiembre de 2017

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ACRÓNIMOS

UAV Unmaned Aerial Vehicle

GPS Global Positioning System

ROS Robot Operating System

PID Proportional Integral Derivative

PC Personal Computer

P2P Peer To Peer

VRPN Virtual-Reality Peripherical Network

IMU Inertial Measurement Unit

Resumen

En el presente documento se describe el diseño e implementación de un sistema de control de posición, navegación y evitación de obstáculos para el cuadricóptero Parrot AR.Drone 2.0 basado en un sistema de captura de movimiento *OptiTrack*.

Para alcanzar este propósito a lo largo del proyecto, se abordan diferentes objetivos: (1) conocer el entorno de desarrollo en el que se elaborará el proyecto, en este caso ROS, (2) familiarizarse con el software del sistema de captura de movimiento, (3) implementar un estimador de posición utilizando las medidas obtenidas por el sistema de captura, (4) desarrollar una interfaz para el seguimiento de trayectorias, (5) implementar un sistema sencillo de navegación y, por último, (6) elaborar un algoritmo para la evitación de obstáculos basado en campos de potencial.

CAPÍTULO 1

INTRODUCCIÓN

1.1 Motivación

A lo largo de los últimos años, se han producido numerosos avances en el desarrollo de los sistemas aéreos autónomos. Inicialmente, estos vehículos se desarrollaron mayoritariamente en campos relacionados con la tecnología militar. No obstante estos vehículos aéreos no tripulados denominados UAV (Unmanned Aerial Vehicle), gracias a la mejora de la tecnología implicada, principalmente a nivel de sensorización y desarrollo de procesadores potentes de bajo consumo y tamaño reducido, han encontrado su lugar en una gran variedad de aplicaciones. En la figura 1.1 podemos ver algunas aplicaciones innovadoras que involucran UAVs.

(a) Drones para agricultura.

(b) Drones de transporte.

Figura 1.1: Aplicaciones de drones.

Los UAVs no suelen emplearse en entornos interiores, debido a la imposibilidad de introducir sistemas de localización externos, dado que estos sistemas están limitados a entornos con línea visual a las balizas (satélites en el caso del GPS). Por lo tanto, el vehículo no puede volar de forma autónoma dentro de un entorno interior mediante estos dispositivos.

Este proyecto se centra en la obtención de un sistema capaz de conseguir que un vehículo aéreo de bajo coste navegue de manera autónoma por un entorno de interior, mediante el uso de un sistema de captura de movimiento basado en luz infrarroja.

Para la realización del presente proyecto se dispone de un Parrot AR.Drone 2.0 (ver figura 1.2). Es un dron cuadricóptero que se puede controlar con un teléfono inteligente o una tablet mediante la aplicación AR.Flight.

Figura 1.2: Parrot AR.Drone 2.0.

Además, el sistema de captura de movimiento comprende 8 cámaras de seguimiento de alta velocidad y un software de captura de movimiento denominado *Motive*. Este sistema permite detectar con una alta precisión cualquier dispositivo que contenga unos determinados marcadores. Estas cámaras de seguimiento no son todas iguales [1]: cuatro de ellas son *PRIME 13* (ver figura 1.3a) y las otras cuatro son *PRIME 13W* (ver figura 1.3b). Las *PRIME 13W* tiene un mayor campo visual pero tienen un menor rango de distancia. En cambio, las *PRIME 13* tienen un menor campo visual pero un mayor rango.

(a) *PRIME 13*. (b) *PRIME 13W*.

Figura 1.3: Cámaras del sistema *OptiTrack* disponible en el laboratorio de robótica aérea.

1.2 Objetivos y alcance

Los objetivos generales de este proyecto son implementar un controlador de posición basado en un sistema de captura de movimiento y combinarlo con un algoritmo de evitación de obstáculos para obtener un sistema de navegación autónomo para un vehículo aéreo.

Estos son los objetivos específicos que se han planteado para alcanzar los objetivos generales previos:

- Aprendizaje del sistema operativo para aplicaciones de robótica ROS (Robot Operating System).
- Estudio del sistema de captura de movimiento *OptiTrack* en cuanto a calibración, estimación de posición y comunicación con el entorno ROS.
- Implementación de un filtro de *Kalman* adecuado para estimar el estado del dron, el cual ha de poder contemplar la incertidumbre de los datos de posicionamiento, así como tolerar posibles ocultaciones del dron.
- Realización de un control de posicionamiento basado en control PID.
- Implementación de un método simple de navegación por puntos de paso.
- Desarrollo de un sistemas sencillo de evitación de obstáculos basado en campos de potencial. Por estar fuera del alcance de este proyecto, la posición de los obstáculos será obtenida mediante el sistema de captura.
- Realización de un conjunto de pruebas en el simulador y después con el robot real, una vez todo el software sea operativo.

En resumen, el alcance del presente proyecto será la navegación por los puntos de paso indicados por el usuario dentro de un entorno, en el cual podrá haber diversos obstáculos intermedios y/o se podrán producir diversas ocultaciones del robot aéreo.

La navegación se realizará en un espacio cerrado, el laboratorio de robótica aérea de la Universidad de las Islas Baleares. La zona destinada a este fin contiene un área de vuelo, en la cual hay ocho cámaras *OptiTrack* que capturan cualquier objeto al que se le haya adosado unos determinados marcadores y se encuentren en esta área. En esta zona, se añadirán diversos obstáculos que podrán impedir el movimiento del dron o para producir ocultaciones en la capturación de la posición del robot. Así se comprobará que la solución adoptada es adecuada.

1.3 Estructura del documento

Este documento esta dividido en diversos capítulos. La presente introducción es el primer capítulo, en el que se presenta la motivación, los objetivos y el alcance del proyecto

y esta misma sección, que describe la estructura del documento.

En el segundo capítulo se incluye los fundamentos teóricos de algunos de los módulos implementados en el proyecto.

El tercer capítulo explicará diversas herramientas software que han sido necesarias para poder realizar el presente proyecto. En las secciones de este capítulo se explicará porqué se han escogido estas herramientas y qué función realiza cada una de ellas.

En el siguiente capítulo, inicialmente se mostrará un esquema de los diferentes módulos necesarios para realizar el proyecto. Luego, se indicará el sistema de ejes que se ha decidido seguir y qué transformaciones serán necesarias. Por último, se explicará cómo se ha implementado cada uno de los diferentes módulos.

En el quinto capítulo se mostrará cómo se han configurado diversos parámetros de cada uno de los módulos y se presentará un conjunto de experimentos, tanto en el simulador como sobre el modelo real, con una breve explicación de cada una de estas pruebas. También se comparará los experimentos del simulador con los experimentos sobre el robot real.

En el último capítulo se recogen un conjunto de conclusiones y algunos trabajos futuros que se pueden realizar a partir del presente proyecto.

Además, se puede encontrar en los anexos el código fuente del software necesario para realizar el proyecto. El primer anexo se corresponde con el código del estimador del estado, el segundo con el controlador de posición, el tercero con el módulo de evitación de obstáculos y el último se corresponde con el módulo de navegación.

Finalmente, se muestra la bibliografía. En este apartado se recogen un conjunto de referencias bibliográficas que han sido necesarias para poder elaborar el presente documento.

CAPÍTULO 2

FUNDAMENTOS TEÓRICOS

Este capítulo revisa de forma escrita los elementos teóricos que constituyen el núcleo del presente proyecto en cuanto a la implementación de la solución: el filtro de *Kalman*, el controlador PID y la estrategia de evitación de obstáculos basado en campos de potencial.

2.1 Filtro de Kalman

El filtro de *Kalman* es un algoritmo recursivo óptimo de procesamiento de datos. Éste emplea una serie de mediciones observables en el tiempo, que contienen un ruido blanco aditivo (este ruido no muestra correlación con la variable del tiempo) y otras inexactitudes, y produce estimaciones de variables desconocidas que tienden a ser más precisas que aquellas basadas en una sola medida. Este algoritmo fue desarrollado por Rudolf E. Kalman en 1960 [2].

Este algoritmo se puede expresar con una ecuación. Sin embargo, a menudo se conceptualiza como dos procesos distintos: *"Predicción"* y "*Corrección*". En el proceso de predicción, se producen estimaciones de las variables de los estados actuales, junto con sus incertidumbres. En el proceso de corrección, una vez se observa el resultado de la siguiente medición con su correspondiente error, estas estimaciones se actualizan usando un promedio ponderado, con más peso a las estimaciones con mayor certeza. Este algoritmo se puede desarrollar en tiempo real, mediante las mediciones de entrada actuales, el estado estimado previamente calculado y la matriz de incertidumbre.

Las posibles aplicaciones del filtro de *Kalman* incluyen la navegación y el control de vehículos, las cuales constituyen casos particulares del procesamiento de señales, aunque también encuentra utilidad en campos tan dispares como la econometría.

El filtro de *Kalman* es un estimador recursivo. Esto significa en este caso que sólo el estado estimado en el tiempo anterior y la actual medición son necesarias para poder estimar el estado actual.

Normalmente las fases de predicción y corrección se alternan, primero se realiza la fase de predicción que estima el estado actual del sistema, y luego la fase de corrección que actualiza el estado estimado mediante la medición. Sin embargo, esto no se produce siempre en este orden: si una medición no esta disponible, una vez se ha realizado la fase de predicción, puede omitirse la fase de corrección y realizarse varios procesos de predicción. Del mismo modo, si se dispone de múltiples observaciones independientes al mismo tiempo, se pueden realizar múltiples procesos de corrección.

En el caso de realizarse varias fases de predicción y/o corrección consecutivas es posible llegar a un grado de inexactitud del estado actual no deseado. Por ejemplo, en el caso de realizar varias fases de predicción y que las estimaciones con mayor certeza vengan dadas durante la fase de corrección.

A continuación se detallan las mencionadas fases de predicción y corrección [3]:

- Predicción:
  - Predicción del estado

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \mathbf{x}_{k-1|k-1} + \mathbf{B}_k u_k \tag{2.1}$$

  - Predicción de la covarianza

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^T + \mathbf{Q}_{k-1} \tag{2.2}$$

- Corrección:
  - Residuo de medida

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1} \tag{2.3}$$

  - Residuo de covarianza

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^\top + \mathbf{R}_k \tag{2.4}$$

  - Ganancia de *Kalman*

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^\top \mathbf{S}_k^{-1} \tag{2.5}$$

  - Estimación actualizada del estado

$$\mathbf{x}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k \tag{2.6}$$

  - Estimación actualizada de la covarianza.

$$\mathbf{P}_{k|k} = (I - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1} \tag{2.7}$$

A continuación explicaremos cada una de las variables anteriores:

* $\hat{\mathbf{x}}_{\mathbf{k|k\text{-}1}}$: Vector de estado estimado *a priori*.
* $\mathbf{F_k}$: Matriz de transición de estado. Matriz que relaciona el estado en el instante anterior con el estado en el instante actual, en ausencia de señales de control.
* $\mathbf{x_{k\text{-}1|k\text{-}1}}$: Vector de estado del estado anterior.
* $\mathbf{B_k}$: Matriz que relaciona las señales de control (opcionales) con el estado actual.
* $\mathbf{u_k}$: Vector de señales de control.
* $\mathbf{P_{k|k\text{-}1}}$: Covarianza del error asociada a la estimación *a priori*.
* $\mathbf{P_{k\text{-}1|k\text{-}1}}$: Covarianza del error asociada al estado anterior.
* $\mathbf{Q_{k\text{-}1}}$: Covarianza del ruido del proceso.
* $\mathbf{z_k}$: Vector de mediciones en el instante actual.
* $\mathbf{R_k}$: Covarianza del ruido en las mediciones.
* $\mathbf{H_k}$: Matriz que relaciona el estado actual con las observaciones del entorno.
* $\mathbf{K_k}$: Ganancia de *Kalman*. La ganancia de *Kalman* indica la confianza en las mediciones obtenidas.
* $\mathbf{x_{k|k}}$: Vector de estado actual estimado.
* $\mathbf{P_{k|k}}$: Covarianza del error asociada al estado actual.

2.2 Controlador PID

El control PID (Proporcional-Integral-Derivativo) [4] ha sido universalmente aceptado en el control industrial y de hecho es el algoritmo de control más utilizado en la industria actual. La popularidad de los controladores PID se puede atribuir en parte a un rendimiento suficiente en una amplia gama de condiciones de funcionamiento y a su simplicidad funcional, que permite a los ingenieros operar de una manera sencilla y directa.

Como su nombre indica, el algoritmo PID consta de tres términos: proporcional, integral y derivativo. La idea básica es leer los valores de los sensores y luego generar la salida para el actuador deseada calculando las respuestas proporcionales, integrales y derivativas. Antes de explicar cada uno de los parámetros correspondientes al PID, se explicará qué es un sistema en lazo cerrado y algunos conceptos relacionados con este término.

Un sistema en lazo cerrado es un proceso por el cual la salida de un sistema se redirige a la entrada con el objetivo de controlar su comportamiento. En la figura 2.1 se ilustra un proceso de este tipo.

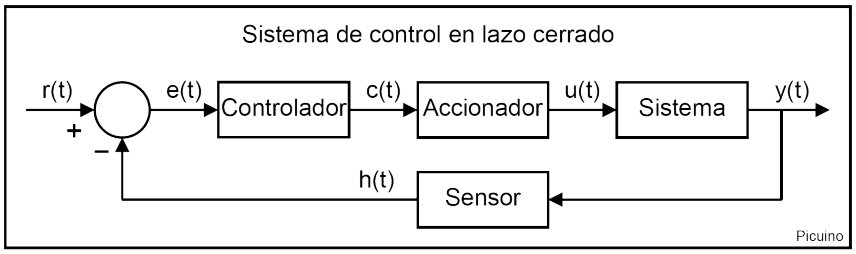


Figura 2.1: Sistema de control en lazo cerrado.

En un sistema de control típico, la variable de proceso es el parámetro del sistema que quiere controlarse, como la velocidad, la temperatura o la humedad. Se emplea un sensor para medir la variable de proceso y proporcionar retroalimentación al sistema de control. El punto de consigna es el valor deseado al que se quiere mantener el sistema, por ejemplo, 85 % de humedad relativa en un sistema de control de humedad. En cualquier momento, la diferencia entre el valor del proceso y el punto de consigna es procesada por el algoritmo de control, para determinar la salida del actuador deseada para compensar el sistema.

Por ejemplo, si la humedad relativa medida por el proceso es del 70 % y el punto de consigna es del 85 %, entonces el actuador especificado por el algoritmo de control realizará una determinada acción, expulsar agua al ambiente mediante un humidificador. Esta expulsión de agua dará lugar a una mayor humedad relativa en el ambiente, es decir, un aumento en la variable de proceso. Este ejemplo es un sistema de control en lazo cerrado.

A continuación, se explicará cada uno de los términos del controlador PID:

- ***Término P***: El componente proporcional depende únicamente de la diferencia entre el punto de consigna y la variable de proceso. Esta diferencia se conoce como el término error. La ganancia proporcional "K_p" determina la relación entre la respuesta de salida y la señal de error. Por ejemplo, si el término de error tiene una magnitud de 10, una ganancia proporcional de 3 produciría una respuesta proporcional de 30.

  Generalmente el aumento de la ganancia proporcional aumentará la velocidad de la respuesta del sistema de control. Sin embargo, si la ganancia proporcional es elevada, la respuesta del sistema será inestable. El aumento de "K_p"provoca una mayor amplitud de las oscilaciones y, por ello, el sistema se puede volver inestable e incluso oscilará fuera de control.

- ***Término D***: El componente derivativo provoca que la salida aumente si la variable de proceso aumenta rápidamente y viceversa. La respuesta derivativa es proporcional al cambio de la variable de proceso. Aumentar el parámetro de tiempo derivativo "K_d" hará que el sistema de control reaccione rápidamente a los cambios con el término de error y aumentará la velocidad de la respuesta del

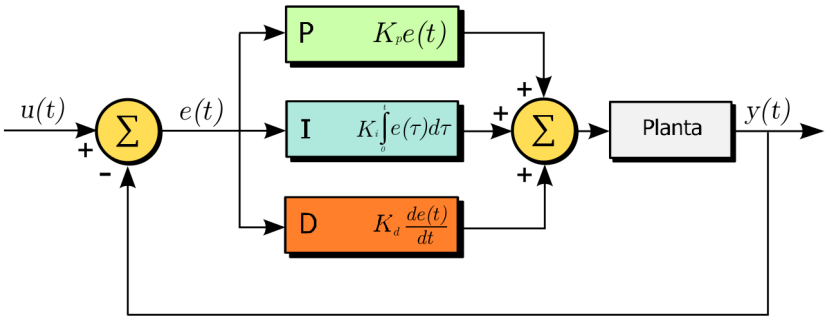


Figura 2.2: Esquema PID.

sistema de control global.

La mayoría de los sistemas de control prácticos usan una ganancia derivativa baja "K_d", debido a que la respuesta derivativa es muy sensible al ruido en la señal correspondiente a la variable del proceso. Si la señal de realimentación del sensor es ruidosa o si la velocidad del circuito de control es lenta, la respuesta derivativa puede hacer que el sistema de control sea inestable.

- ***Término I***: El componente integral suma el término de error a lo largo del tiempo. El resultado es que incluso un pequeño término de error hará que el componente integral varíe. La respuesta integral aumentará continuamente con el tiempo a menos que el error sea cero, por lo que el efecto es conducir el error de estado estable a cero. Un fenómeno que puede producirse resulta cuando la acción integral satura la respuesta del controlador sin que se conduzca la señal de error hacia cero.

En la figura 2.2 se muestra cómo calcular cada uno de los diferentes componentes para un sistema controlado por un regulador PID. La salida proporcionada se puede observar en la ecuación 2.8.

$$y(t) = K_p e(t) + K_i \int_0^t e(t)\,dt + K_d \frac{d}{dt} e(t) \tag{2.8}$$

Donde e(t) es la señal de error calculada como u(t) - y(t).

2.3 Campos de potencial

Existen diferentes tipos de algoritmos de evitación de obstáculos en el área de la robótica. Uno de los más básicos es el método basado en campos de potencial (ver bibliografía [5]).

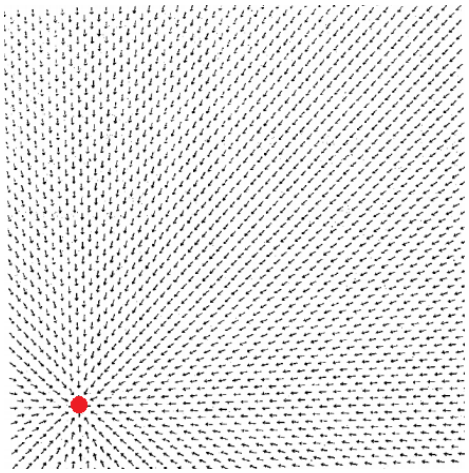
En los siguientes puntos se explica en qué consiste este método:

- Los diferentes obstáculos y el punto objetivo generan campos de fuerza a su alrededor.

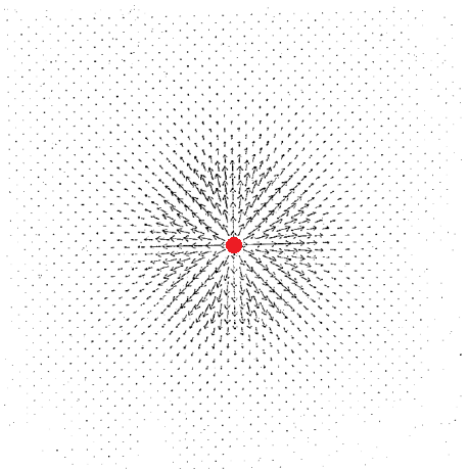
- El robot se modela como una partícula sometida a una serie de campos de fuerza creados por los componentes mencionados en el apartado anterior.
- Los obstáculos dan lugar a fuerzas de repulsión.
- El punto objetivo da lugar a fuerza de atracción.

Si suponemos un escenario lleno de obstáculos en el cual un robot debe llegar a un objetivo, el robot irá hacia este objetivo, ya que el campo potencial creado por el punto objetivo realizará una fuerza atractiva hacia él, pero de forma que cada vez que el robot se encuentre cerca de un obstáculo, el campo potencial creado por éste provocará una fuerza repulsiva evitando la colisión con éste. Así durante todo el recorrido se obtendrá un vector de fuerza resultante de las repulsiones de los obstáculos y atracción al objetivo. Finalmente, el robot llegará al punto objetivo evitando todos los obstáculos del escenario.

En la figura 2.3a se observa el campo de fuerzas creado por el punto objetivo y en la figura 2.3b el campo de fuerzas creado por un obstáculo.



(a) Campo creado por el punto objetivo localizado abajo izquierda.



(b) Campo creado por un obstáculo localizado en el centro.

Figura 2.3: Ilustración del concepto de campos de potencial.

CAPÍTULO 3

HERRAMIENTAS SOFTWARE UTILIZADAS

En este capítulo se incluye una revisión breve de las principales herramientas software utilizadas para llegar a acabo este proyecto: el entorno de programación ROS (Robot Operating System), el simulador *Gazebo*, el software de captura de movimiento *Motive* y otros paquetes adicionales no disponibles de forma nativa en ROS.

3.1 ROS

ROS [6] es una plataforma software que proporciona al usuario una colección de herramientas, bibliotecas y convenciones que tienen como objetivo simplificar la tarea de crear aplicaciones robóticas complejas y robustas, para una amplia variedad de plataformas robóticas.

Esta plataforma software nace en el año 2007 bajo el nombre de *"switchyard"* por el laboratorio de inteligencia artificial de *Stanford*. En 2008 el desarrollo continuó en *Willow Garage*, un instituto de investigación de robótica con varias instituciones colaborando en un modelo de desarrollo federado.

ROS proporciona no solo servicios estándar del sistema operativo en cuanto a abstracción de hardware, gestión de memoria y gestión de procesos, sino también funcionalidades de alto nivel tales como comunicaciones asíncronas y síncronas, base de datos centralizada y sistema de configuración de robots.

La estructura de ROS es modular. Se crean proyectos en paquetes que contienen los códigos necesarios para desarrollar las aplicaciones robóticas según las necesidades de cada usuario. Con la estructura de software abierto, ROS permite emplear diversos paquetes de distintos proyectos de otras plataformas software como *Gazebo* u *OpenCV*, para poder operar con estos y realizar proyectos de cualquier grado de dificultad.

ROS comprende una arquitectura P2P (Peer to Peer). Una estructura P2P consiste en una comunicación en la que todos o algunos aspectos funcionan sin clientes ni servidores fijos, sino con una serie de *"nodos"* que se comportan como iguales entre sí (ver figura 3.1). Implementa dos tipos de comunicación: la asíncrona se realiza mediante *"topics"* y la síncrona mediante *"servicios"*. La configuración se almacena en un servidor con el nodo principal "*master*" que se encarga de todos los nodos, servicios y *topics*.

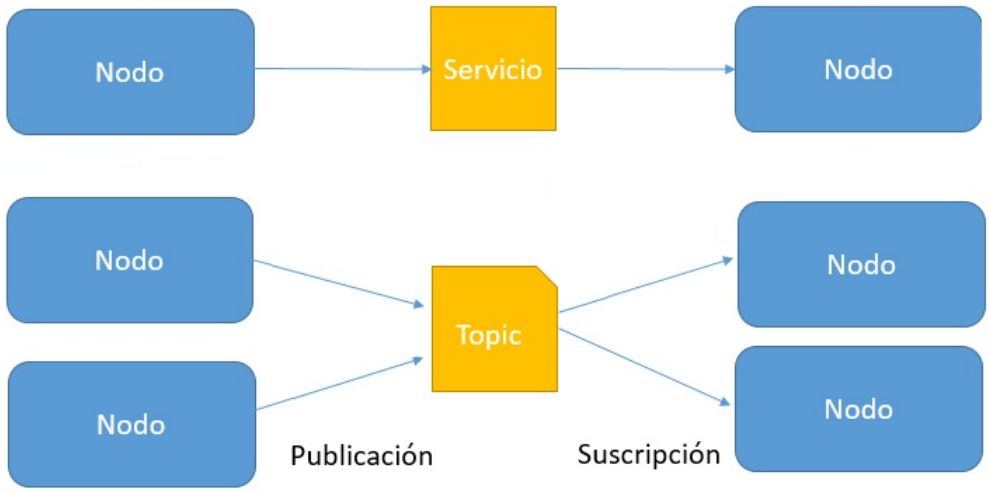


Figura 3.1: Arquitectura P2P ROS.

Este sistema se estructura en tres niveles [7]: el nivel del sistema de archivos, el nivel de computación y el nivel de la comunidad. Estos niveles se resumen a continuación:

- ***Nivel de sistema de archivos***: Se refiere a la organización de los archivos dentro del sistema. Se organizan mayoritariamente en paquetes. Algunos conceptos importantes de este nivel son los siguientes:
  - ***Paquetes***: Los paquetes son la unidad principal para organizar el software en ROS. Un paquete puede contener procesos de tiempo de ejecución de ROS (nodos), una librería dependiente de ROS, conjuntos de datos, archivos de configuración o cualquier otra herramienta que esté organizada de manera útil.
  - ***Archivos paquetes.xml***: Los archivos paquetes.xml proporcionan datos sobre un paquete, incluyendo su nombre, versión, descripción, información de licencia, dependencias y otra información, como paquetes exportados.
- ***Nivel de computación***: Se refiere a la red P2P de los procesos ROS que constituyen una aplicación. Los conceptos básicos de computación de ROS son los nodos, el nodo *master*, el servidor de parámetros, los mensajes, los servicios y los *topics*:

- ***Nodos***: Los nodos son procesos que realizan el cálculo. Por ejemplo, un nodo controla un localizador basado en un sensor láser, otro nodo controla los motores de la rueda, otro la localización, etc.

- ***Nodo master***: El nodo *master* proporciona registro de nombres y búsqueda al resto de la aplicación. Sin el nodo *master*, los nodos no podrían encontrarse, intercambiar mensajes o invocar servicios.

- ***Servidor de parámetros***: El servidor de parámetros permite almacenar los datos globales por clave en una ubicación central.

- ***Mensajes***: Los nodos se comunican entre sí al transmitir mensajes. Un mensaje es simplemente una estructura de datos que comprende campos con valor. Están permitidos tipos primitivos estándar (entero, punto flotante, booleano, etc.), así como matrices de tipos primitivos.

- ***Topics***: Los mensajes se enrutan a través de un sistema de transporte con semántica de publicación / suscripción. Un nodo envía un mensaje publicándolo en un tema determinado. El *topic* es un nombre que se utiliza para identificar el contenido del mensaje. Un nodo que está interesado en un cierto tipo de datos se suscribirá al *topic* apropiado. Puede haber varios publicadores y suscriptores simultáneos para un único *topic*.

- ***Servicios***: El modelo de publicación / suscripción es un paradigma de comunicación muy flexible, pero su transporte multisectorial de un solo sentido no es apropiado para las interacciones de petición / respuesta, que a menudo son necesarias. La petición / respuesta se realiza a través de servicios, que están definidos por un par de estructuras de mensajes: una para la solicitud y otra para la respuesta. Un nodo proveedor ofrece un servicio bajo un nombre y un cliente utiliza el servicio enviando el mensaje de solicitud y esperando la respuesta.

- ***Nivel de comunidad***: Se refiere a la creación de una comunidad robótica que comparte software y códigos para la creación de diferentes paquetes.

  Los principales conceptos asociados a este nivel son:

  - ***Distribuciones***: Las distribuciones ROS son colecciones de librerías versionadas que se pueden instalar. Éstas facilitan la instalación de una colección de software y también mantienen versiones consistentes a través de un conjunto de software.

  - ***Repositorios***: ROS se basa en una red federada de repositorios de código, donde diferentes instituciones pueden desarrollar y liberar sus propios componentes de software de robot.

  - ***ROS WIKI***: La comunidad *ROS Wiki* es el principal foro donde se documenta información sobre ROS.

– ***ROS Answer***: Se trata de una red de comunicación entre usuarios de ROS en la que se pueden realizar consultas acerca de proyectos basados en éste.

3.2 GAZEBO

Gazebo es un simulador 3D multi-robot para espacios interiores y exteriores, con un motor de física y cinemática potente. Éste permite una rápida ejecución de algoritmos, diseño de modelos robóticos y pruebas de regresión usando escenarios realistas (ver figura 3.2).

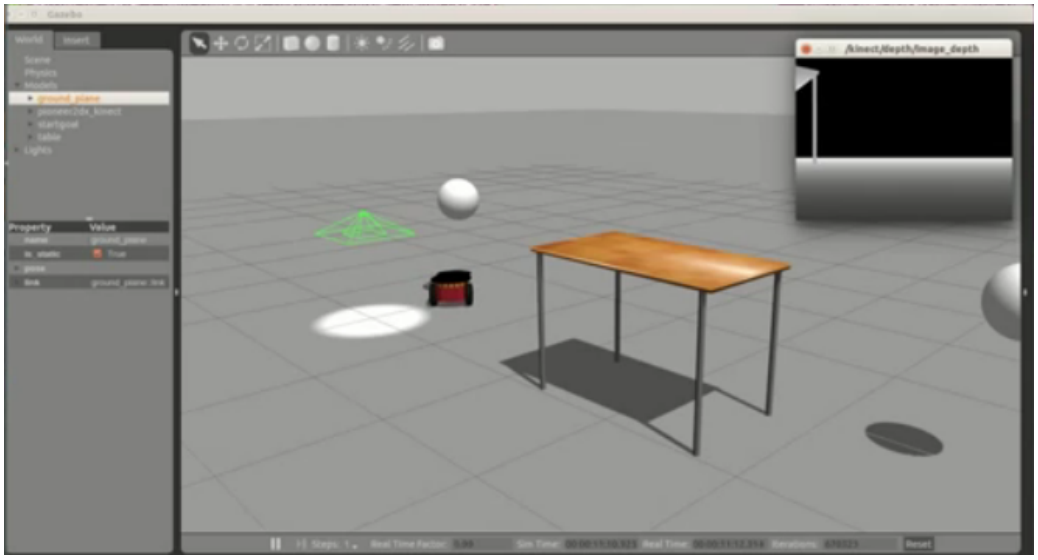


Figura 3.2: Ejemplo de entorno de simulación empleando *Gazebo* con un robot terrestre.

Este simulador permite aplicar modelos físicos realistas tal como fuerzas gravitatorias, velocidades, aceleraciones... Además, la integración entre ROS y *Gazebo* es proporcionada por un conjunto de *plugins* que soportan una gran variedad de robots y sensores. Estos *plugins* presentan la misma interfaz de mensajes que el resto del sistema de ROS, se pueden implementar algoritmos perfectamente compatibles con la simulación y el hardware. Por ello, podemos desarrollar una aplicación en este simulador, y luego implementarla sobre el modelo real con mínimos cambios en el código.

Se ha empleado esta herramienta software para evitar colisiones y problemas en el laboratorio. El sistema que se debía implementar se debía realizar en un entorno cerrado con poco espacio de vuelo. Debido a esto, se ha realizado un conjunto de pruebas en el simulador, para observar la fiabilidad de los algoritmos y evitar daños innecesarios en el robot real.

3.3 Motive

Motive es una plataforma de software diseñada para controlar sistemas de captura de movimiento para diversas aplicaciones de seguimiento. *Motive* no solo permite al usuario calibrar y configurar el sistema, sino que también proporciona interfaces para capturar y procesar datos 3D. Éste obtiene información 3D a través de reconstrucción, en término lógico *Motive,* es el proceso de compilar múltiples imágenes 2D de marcadores, para obtener coordenadas 3D. Mediante el uso de coordenadas 3D de los marcadores de seguimiento, *Motive* puede gestionar 6 grados de libertad (posición 3D y orientación) para múltiples cuerpos rígidos, y permitir el seguimiento de movimientos complejos en el espacio 3D [8].

En el menú *layout* del programa tenemos las opciones mas utilizadas en este proyecto (ver figura 3.3).

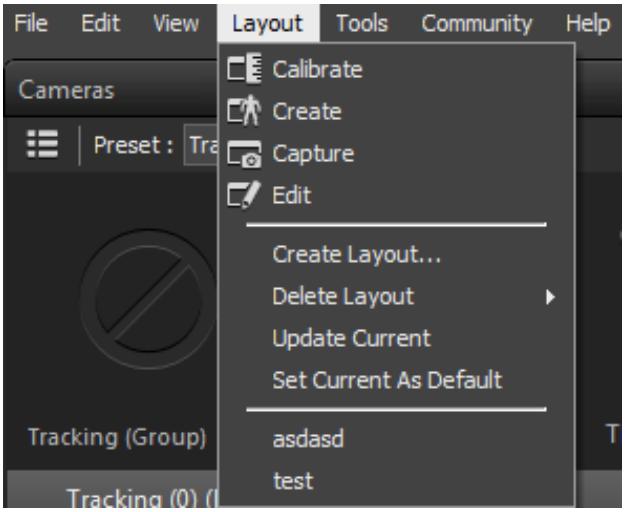


Figura 3.3: Secciones de *layout*.

- ***Calibrate***: La opción de calibración sirve para configurar las cámaras. Por ejemplo, eliminar reflejos de luz no deseados que pueden proporcionar una posición errónea.
- ***Create***: Esta opción corresponde en la creación de los elementos que luego detectarán las cámaras mediante luz infrarroja, tales como *skelet bodies, rigid bodies* o marcadores individuales.
- ***Capture***: Con esta opción podremos grabar, procesar y emitir las tomas de las cámaras. Mediante el apartado *edit* podremos editar estas grabaciones y etiquetar los distintos elementos que aparezcan en ellas.

Todos estas opciones de *layout* se pueden configurar a conveniencia del usuario, para optimizar el proceso y poder utilizar el sistema de seguimiento como el usuario desee.

Hay un conjunto de vistas que ayudarán al usuario a orientarse y saber si el sistema esta funcionando correctamente.

En *Camera Preview* (ver figura 3.4) se muestra lo que están detectando las cámaras. Todos los reflejos de luz que se detecten, se mostrarán en esta vista. Supuestamente, todos los reflejos detectados tendrían que venir dados por los marcadores pero en ocasiones se producen reflejos no deseados. En esta vista se detectan estas anomalías y se pueden eliminar mediante máscaras. Además, podemos cambiar el modo de operación de las cámaras entre los diferentes modos disponibles.

Figura 3.4: Ejemplo de la vista *Camera preview*.

En *Perspective View* (ver figura 3.5) se observa en tiempo real el movimiento de los elementos rastreados. En la parte superior izquierda, se encuentran los elementos definidos, tales como *skelet bodies o rigid bodies*, que pueden seleccionarse y gestionarse desde esta vista. Si seleccionamos uno de ellos se mostrará las diferentes características, como su posición y orientación en tiempo real.

A continuación, se explicará brevemente qué son los marcadores y qué es necesario para crear un *rigid body* o un *skelet body*.

Los marcadores son unas pequeñas bolas reflectantes que devuelven la luz infrarroja emitida por las cámaras *OptiTrack*. Gracias a la reflexión de los marcadores, las cámaras son capaces de detectarlos e introducir la posición de la realidad en el entorno virtual.

Un *rigid body* está formado por un conjunto de marcadores que mantienen su posición constante. Así el software que realiza el *tracking* es capaz de etiquetarlos y diferenciarlos por la colocación de los diferentes marcadores. Para la creación de un *rigid body* son necesarios 3 marcadores o más. Esto es debido a que para conseguir la

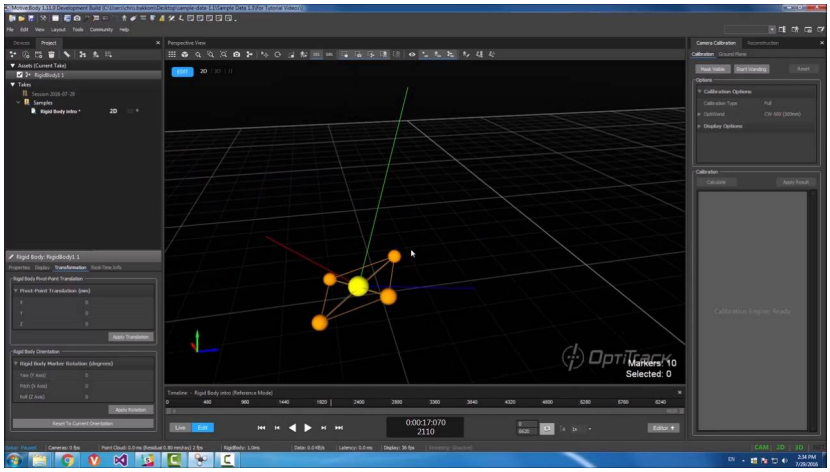


Figura 3.5: Ejemplo de la vista *Perspective view*.

posición 3D de un cuerpo es necesario 3 marcadores, pero se recomienda definir 4 por si se producen ocultaciones y para facilitar diferenciar este recurso de otros. Además, dentro de un cuerpo rígido, un conjunto de marcadores debe colocarse asimétricamente, porque proporciona una clara distinción de la orientación.

Figura 3.6: Parrot Ar.Drone 2.0 con sus respectivos marcadores.

3.4 Módulos externos

Adicionalmente, se han incorporado los siguientes módulos. El módulo VRPN *client* es el que ha permitido extraer la posición de nuestro UAV e introducirlo en ROS. El módulo de teleoperación permite interactuar con el dron mediante un *joystick* por si se produce algún tipo de problema durante una operación de vuelo. El *ardrone driver* permite transmitir las velocidades que deseamos que el *dron* adquiera.

3.4.1 VRPN *client*

La aplicación *Motive* proporciona dos interfaces de comunicación con software externo:

- ***NatNet***: Es una interfaz con conexión cliente / servidor para el envío y recepción de datos a través de la red. [9]
- ***VRPN***: Es una interfaz que facilita la interacción entre aplicaciones y dispositivos interactivos para realidad virtual [10].

Finalmente escogimos VRPN porque era el más simple de implementar e integrar con ROS, compatible con *OptiTrack* y por sus diferentes características.

VRPN ofrece la posibilidad de coordinar y controlar distintos periféricos desde una máquina sin preocuparse por la topología de la red. Además, proporciona una capa de abstracción que da lugar a que todos los dispositivos parezcan del mismo tipo y sean tratados de la misma manera. También puede coordinarse a través de la red lo que permite tener varios dispositivos VRPN conectados y en equipos distintos.

El protocolo VRPN requiere de dos elementos: un servidor y un cliente. El servidor recoge toda la información obtenida por los sensores o recursos empleados y se la envía a los diferentes clientes que tenga registrados. El cliente, por su parte, se registra a los servidores de los dispositivos a los cuales quiera conectarse, y mediante la red o de otra manera obtiene la información proporcionada por el servidor o servidores.

En la figura 3.7 se muestra un esquema que representa la configuración adoptada para el presente proyecto. El sensor que capta la información sería *OptiTrack*, el servidor se ejecuta en el PC que contiene *Motive*, que se encarga de recolectar la información suministrada por *OptiTrack* y la envía a los diferentes clientes mediante la red. El PC portátil, que es el que debe tener toda la información, se tiene que registrar en el servidor anterior. Una vez realizado este proceso, ya se tiene acceso a las posiciones del dron.

El módulo *"vrpn_client_ros"* [11] implementa la interfaz VRPN en ROS y hace posible el proceso de la figura 3.7.

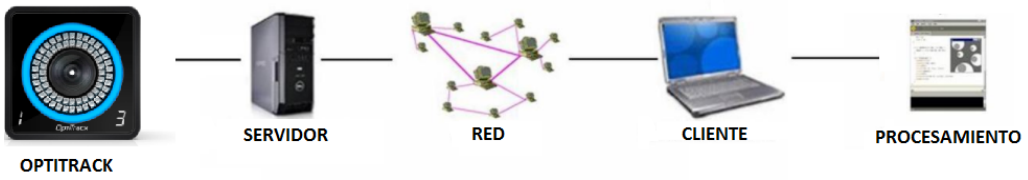


Figura 3.7: Comunicación VRPN con el sistema *OptiTrack*.

3.4.2 Módulo de teleoperación

Este módulo permite alternativamente controlar el vuelo en modo manual mediante un *joystick* y en modo autónomo. Se ha añadido este módulo para permitir al usuario tomar el control del dron cuando se pilote en modo autónomo. Si en este último modo

se va a producir una colisión, el usuario podrá adquirir el control del robot aéreo y evitar el accidente.

Inicialmente, se empleó este módulo para poder controlar el dron mediante un *joystick*. Así se pudo observar cómo navegaba el dron por el entorno y adquirir destreza a la hora de pilotarlo.

En la figura 3.8 se observa el *joystick* empleado denominado "*Extreme 3D Pro Joystick*". A continuación, explicaremos qué controles implementa este módulo.

Figura 3.8: Controles *Extreme 3D Pro Joystick*.

- ***Botón 1***: Cambia el control a modo manual.
- ***Botón 3***: Cambia el control a modo autónomo.
- ***Botón 6***: Este botón solamente se debe utilizar en combinación con otros botones. Esto se ha realizado de esta forma para dar una mayor seguridad a los comandos producidos con estas combinaciones.
- ***Joystick 8****: Este pequeño *joystick* junto con la palanca del *Joystick* permite ordenar la elevación del dron (palanca inclinada hacia adelante) así como el descenso de altitud (palanca inclinada hacia atrás).

- ***Botón 9***: Manteniéndolo pulsado junto con el botón 6 se ordena la acción de despegar.

- ***Botón 12***: Manteniéndolo pulsado junto con el botón 6 se ordena la acción de aterrizaje.

- ***Joystick***: Por último, dependiendo de hacia dónde inclinemos la palanca del *joystick* se producirá un cambio u otro en los dos ejes de rotación de la aeronave, *roll* y *pitch*. Estos cambios combinados con los movimientos producidos por el *joystick* 8* permitirán desplazar el dron a cualquier punto del espacio. Además, podemos girar el *joystick* en sentido horario o antihorario para dar lugar a un cambio en el último eje de rotación de la aeronave, ψ. Esto permitirá orientar al dron. En la referencia [12] se puede encontrar información sobre los ejes de rotación de una aeronave.

Los botones no mencionados no producen ningún efecto en el movimiento del dron.

3.4.3 *Ardrone driver*

Algunos dispositivos necesitan de *drivers* específicos para la comunicación entre el dispositivo y el PC. *Ardrone driver* implementa los *drivers* para los cuadricópteros AR Drone 1.0 y AR Drone 2.0. Se encarga de la comunicación con el AR Drone y a su vez publica una serie de *topics* y servicios para el control y la lectura de sus sensores. Los *topics* más relevantes para este proyecto son los siguientes:

- ***ardrone/navdata***: Este *topic* permite observar las distintas mediciones de los sensores del dron como el porcentaje de batería (necesario para saber si se puede realizar un vuelo), velocidades, altura, ...

- ***ardrone/takeoff y ardrone/land***: Mediante estos *topics* podemos ordenar que el dron despegue o aterrice. Para poder realizar alguna de estas acciones deberemos publicar un mensaje vacío en el *topic* correspondiente.

En la referencia [13] podemos observar todas las actualizaciones y los contenidos que ofrece este módulo.

CAPÍTULO 4

IMPLEMENTACIÓN

Este capítulo describe la implementación de la solución adoptada en este proyecto. Después de proporcionar una visión general, se describe de forma detallada cada uno de los elementos importantes de la solución.

4.1 Vista general de la solución adoptada

En la figura 4.1 se muestra un esquema que contiene todos los módulos necesarios y *topics* empleados para realizar el presente proyecto.

En el esquema cada rectángulo representa un módulo. Algunos de ellos se han explicado anteriormente en la sección 3.4. El resto de los módulos se explicarán en la sección 4.3, a excepción del módulo *Joystick*, que corresponde al *driver* del *joystick*. Los diferentes *topics* son representados mediante flechas que representan la información transmitida de un módulo a otro. Por último, la "Lista de navegación" que representa el fichero que contiene la lista de puntos a los que debe acudir el vehículo.

Este esquema es válido sobre el modelo real. En el entorno virtual, el módulo VRPN *client* es reemplazado por el módulo *Gazebo*, el cuál publica un *topic* con la posición de la plataforma, emulando el sistema *OptiTrack*.

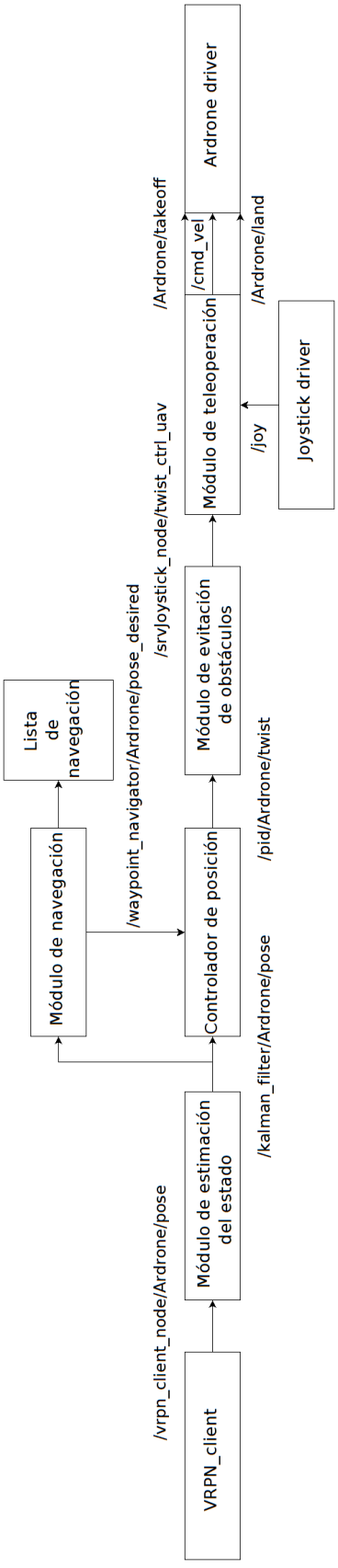


Figura 4.1: Visión general de la solución adoptada.

4.2 Convención de ejes

En esta sección se indican los sistemas de coordenadas usados para el presente proyecto.

Para su definición se ha seguido la referencia proporcionada por el documento REP 103 de ROS [14].

Todos los sistemas de coordenadas usados cumplen con la regla de la mano derecha. La regla de la mano derecha es una técnica común para entender convenciones de orientación para vectores en tres dimensiones. Según esta regla, el dedo pulgar derecho apunta a lo largo del eje Z en una dirección Z positiva y el rizo de los dedos representa un movimiento desde el eje X al eje Y. Cuando se ve desde el eje superior o Z, el sistema está en sentido contrario a las agujas del reloj [15].

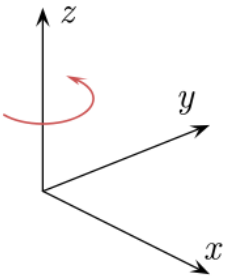


Figura 4.2: Sistema de coordenadas definido mediante la regla de la mano derecha.

El sistema de coordenadas del robot es definido de tal manera que el eje X apunta hacia delante, el Y hacia la izquierda y el Z hacia arriba. Estos ejes definen las direcciones positivas para el desplazamiento del dron.

En cuanto a las rotaciones entorno a un eje, se consideran positivas si estas son en sentido contrario de la agujas del reloj, siguiendo la regla de la mano derecha.

Finalmente, los ejes de coordenadas definidos por el sistema *OptiTrack* presentan una orientación diferente. Este sistema sigue la misma convención de ejes que el mencionado anteriormente pero con una rotación en el eje X de 90º. El sistema de coordenadas es el que se ilustra en la siguiente figura 4.3.

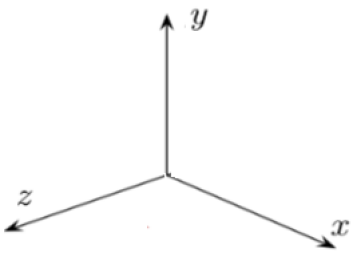


Figura 4.3: Ejes del sistema *OptiTrack*.

4.3 Módulos implementados

En esta sección se explica como se han implementado los diferentes módulos necesarios para el presente proyecto, incluyendo expresiones matemáticas, diagramas de flujo y explicaciones de los métodos más relevantes con el objetivo de detallar la función que realizan los diferentes módulos.

4.3.1 Módulo de estimación del estado

El módulo de estimación del estado se ha implementado basándose en los fundamentos teóricos recogidos en la sección 2.1. El objetivo de este algoritmo es filtrar las medidas de posición y orientación proporcionados por el sistema *OptiTrack*, obteniendo así una estimación del estado del dron. Además, el módulo permite estimar el estado cuando se producen ocultaciones en el entorno, inhabilitando el sistema *OptiTrack*.

Para el desarrollo de este módulo, se ha empleado diferentes clases de la librería de software abierto *OpenCV*. Entre ellas, destacamos la clase *KalmanFilter* [16]. Esta clase proporciona una serie de herramientas que facilitan la implementación de un filtro de *Kalman*, tales como creación de las matrices necesarias, o las funciones de predicción y corrección.

Se ha definido el movimiento del vehículo como un movimiento rectilíneo uniforme. Las expresiones matemáticas que representan este movimiento son las siguientes.

$$x_t = x_{t\text{-}1} + \dot{x}_{t\text{-}1} \triangle t \qquad \dot{x}_t = \dot{x}_{t\text{-}1} \tag{4.1}$$

$$y_t = y_{t\text{-}1} + \dot{y}_{t\text{-}1} \triangle t \qquad \dot{y}_t = \dot{y}_{t\text{-}1} \tag{4.2}$$

$$z_t = z_{t\text{-}1} + \dot{z}_{t\text{-}1} \triangle t \qquad \dot{z}_t = \dot{z}_{t\text{-}1} \tag{4.3}$$

$$\psi_t = \psi_{t\text{-}1} + \dot{\psi}_{t\text{-}1} \triangle t \qquad \dot{\psi}_t = \dot{\psi}_{t\text{-}1} \tag{4.4}$$

A continuación, se muestra todas las matrices necesarias para implementar el filtro de *Kalman*:

- Matriz de transición de estados.

$$F_t = \begin{bmatrix} 1 & 0 & 0 & 0 & \triangle t & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \triangle t & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & \triangle t & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \triangle t \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

- Matriz de covarianza de ruido.

$$Q_t = \begin{bmatrix} \sigma^2_{xx} & 0 & 0 & 0 & \sigma^2_{x\dot{x}} & 0 & 0 & 0 \\ 0 & \sigma^2_{yy} & 0 & 0 & 0 & \sigma^2_{y\dot{y}} & 0 & 0 \\ 0 & 0 & \sigma^2_{zz} & 0 & 0 & 0 & \sigma^2_{z\dot{z}} & 0 \\ 0 & 0 & 0 & \sigma^2_{\psi\psi} & 0 & 0 & 0 & \sigma^2_{\psi\dot{\psi}} \\ \sigma^2_{\dot{x}x} & 0 & 0 & 0 & \sigma^2_{\dot{x}\dot{x}} & 0 & 0 & 0 \\ 0 & \sigma^2_{\dot{y}y} & 0 & 0 & 0 & \sigma^2_{\dot{y}\dot{y}} & 0 & 0 \\ 0 & 0 & \sigma^2_{\dot{z}z} & 0 & 0 & 0 & \sigma^2_{\dot{z}\dot{z}} & 0 \\ 0 & 0 & 0 & \sigma^2_{\dot{\psi}\psi} & 0 & 0 & 0 & \sigma^2_{\dot{\psi}\dot{\psi}} \end{bmatrix} \tag{4.6}$$

La matriz Q_t 4.6 es la covarianza que existen entre el ruido de todas las variables. Debido a que, esta matriz contiene muchos parámetros y configurarlos conllevaría una gran cantidad de tiempo, se ha decidido considerar una aceleración constante media como el ruido para cada una de las medidas observables. Entonces, tendremos una aceleración constante para cada una de estas medidas que se configurarán previamente. Sabiendo que las ecuaciones correspondientes a la cinemática de un movimiento uniformemente acelerado son:

$$x_t = x_{t-1} + \dot{x}_{t-1} \triangle t + \frac{1}{2}\ddot{x}_{t-1} \triangle t^2, \tag{4.7}$$

$$\dot{x}_t = \dot{x}_{t-1} + \ddot{x}_{t-1} \triangle t, \tag{4.8}$$

$$\ddot{x}_t = \ddot{x}_{t-1}, \tag{4.9}$$

entonces las covarianzas para la medida x se pueden representar como:

$$\sigma^2_{xx} = \sigma_x \cdot \sigma_x = \frac{1}{2}\ddot{x}_{t-1}{}^2 \triangle t^2 \cdot \frac{1}{2}\ddot{x}_{t-1}{}^2 \triangle t^2 = \frac{1}{4}a_x{}^2 \triangle t^4, \tag{4.10}$$

$$\sigma^2_{\dot{x}x} = \sigma_x \cdot \sigma_{\dot{x}} = \frac{1}{2}\ddot{x}_{t-1}{}^2 \triangle t^2 \cdot \ddot{x}_{t-1} \triangle t = \frac{1}{2}a_x{}^2 \triangle t^3 \text{ y} \tag{4.11}$$

$$\sigma^2_{\dot{x}\dot{x}} = \sigma_{\dot{x}} \cdot \sigma_{\dot{x}} = \ddot{x}_{t-1} \triangle t \cdot \ddot{x}_{t-1} \triangle t = a_x{}^2 \triangle t^2. \tag{4.12}$$

Realizando la misma operación con el caso de medidas, podemos reescribir la matriz Q_t como:

$$Q_t = \begin{bmatrix} \frac{1}{4}a_x{}^2 \triangle t^4 & 0 & 0 & 0 & \frac{1}{2}a_x{}^2 \triangle t^3 & 0 & 0 & 0 \\ 0 & \frac{1}{4}a_y{}^2 \triangle t^4 & 0 & 0 & 0 & \frac{1}{2}a_y{}^2 \triangle t^3 & 0 & 0 \\ 0 & 0 & \frac{1}{4}a_z{}^2 \triangle t^4 & 0 & 0 & 0 & \frac{1}{2}a_z{}^2 \triangle t^3 & 0 \\ 0 & 0 & 0 & \frac{1}{4}a_\psi{}^2 \triangle t^4 & 0 & 0 & 0 & \frac{1}{2}a_\psi{}^2 \triangle t^3 \\ \frac{1}{2}a_x{}^2 \triangle t^3 & 0 & 0 & 0 & a_x{}^2 \triangle t^2 & 0 & 0 & 0 \\ 0 & \frac{1}{2}a_y{}^2 \triangle t^3 & 0 & 0 & 0 & a_y{}^2 \triangle t^2 & 0 & 0 \\ 0 & 0 & \frac{1}{2}a_z{}^2 \triangle t^3 & 0 & 0 & 0 & a_z{}^2 \triangle t^2 & 0 \\ 0 & 0 & 0 & \frac{1}{2}a_\psi{}^2 \triangle t^3 & 0 & 0 & 0 & a_\psi{}^2 \triangle t^2 \end{bmatrix} \tag{4.13}$$

- Matriz que muestra la relación entre el estado actual y las observaciones del entorno.

$$H_t = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.14}$$

- Matriz de covarianza de ruido de las mediciones.

$$R_t = \begin{bmatrix} {\sigma_{mx}}^2 & 0 & 0 & 0 \\ 0 & {\sigma_{my}}^2 & 0 & 0 \\ 0 & 0 & {\sigma_{mz}}^2 & 0 \\ 0 & 0 & 0 & {\sigma_{m\psi}}^2 \end{bmatrix} \tag{4.15}$$

  Los valores de dicha matriz se configuran manualmente.

- Medida obtenida por los sensores.

$$\mathbf{z_t} = \begin{bmatrix} x_m \\ y_m \\ z_m \\ \psi_m \end{bmatrix} \tag{4.16}$$

- Inicialización del estado. Se inicializa con el primer valor medido por los sensores y se considera una velocidad inicial igual a 0. La matriz obtenida con estos cambios es la siguiente:

$$\mathbf{x_0} = \begin{bmatrix} x_m \\ y_m \\ z_m \\ \psi_m \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.17}$$

Con todas las matrices de esta sección se pueden realizar todas las operaciones que se muestran en las etapas de predicción y corrección.

En la figura 4.4 se muestra un diagrama de flujo que esquematiza este filtro de *Kalman*. En éste se observa las funciones principales y en qué momento se realizan cada una de las funciones.

- ***Inicialización***: Esta función inicializa el valor del estado, al primer valor medido por el sistema *OptiTrack*.

- ***Actualizar valores***: Mediante esta función se actualiza el estado usando los valores correspondientes al último estado publicado. Además, se inicializa un temporizador para controlar el tiempo transcurrido entre cada ciclo.

- ***Predicción***: Se realizan los diferentes cálculos de la etapa de predicción del filtro de *Kalman*.

- ***Corrección***: Si el tiempo desde el cual se ha obtenido un valor por las cámaras es menor a un cierto período, se realizan los cálculos correspondientes a la etapa de corrección. Este período se ha configurado con un valor mayor al tiempo que se obtienen muestras por las cámaras. Así, si se produce algún tipo de ocultación, este tiempo será mayor al período y se estimará una posición mediante la etapa de predicción.

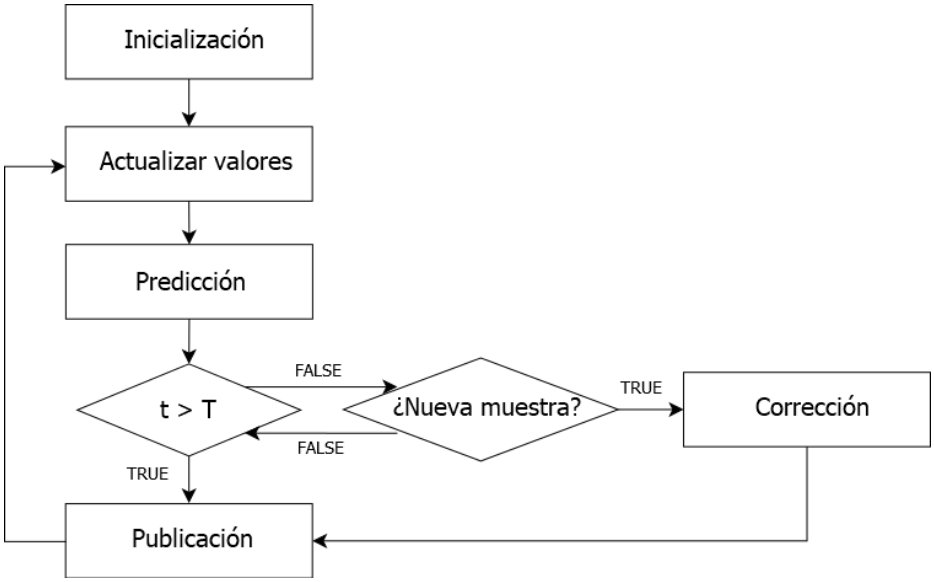


Figura 4.4: Diagrama de flujo del módulo de estimación del estado.

- ***Publicación***: En esta función se publican todos los valores obtenidos mediante los cálculos: la posición filtrada, la covarianza de esta posición, la velocidad calculada y su covarianza. Los dos últimos son usados por el módulo "*Controlador de posición*".

Notese que mediante este nodo se puede realizar la estimación de estado tanto en el modelo real como en el simulador.

4.3.2 Controlador de posición

Este módulo incorpora un controlador PID para cada uno de los ejes de posición (X, Y ,Z) así como para la rotación entorno al eje Z. Mediante este módulo se pretende adquirir la velocidad necesaria para alcanzar cualquier lugar del espacio con la orientación deseada.

Estas son las expresiones matemáticas que se deben implementar en este módulo:

$$\dot{x}_{\text{deseada}} = \varepsilon_{\text{x}} K_{\text{p}} + \frac{\delta \varepsilon_{\text{x}}}{\delta t} k_{\text{d}} + \int \varepsilon_{\text{x}} dt k_{\text{i}}, \tag{4.18}$$

$$\dot{y}_{\text{deseada}} = \varepsilon_{\text{y}} K_{\text{p}} + \frac{\delta \varepsilon_{\text{y}}}{\delta t} k_{\text{d}} + \int \varepsilon_{\text{y}} dt k_{\text{i}}, \tag{4.19}$$

$$\dot{z}_{\text{deseada}} = \varepsilon_{\text{z}} K_{\text{p}} + \frac{\delta \varepsilon_{\text{z}}}{\delta t} k_{\text{d}} + \int \varepsilon_{\text{z}} dt k_{\text{i}} \text{ y} \tag{4.20}$$

$$\dot{\psi}_{\text{deseada}} = \varepsilon_{\psi} K_{\text{p}} + \frac{\delta \varepsilon_{\psi}}{\delta t} k_{\text{d}} + \int \varepsilon_{\psi} dt k_{\text{i}}. \tag{4.21}$$

donde el error ε corresponde al valor deseado menos el valor estimado.

El término derivativo se puede desglosar en la siguiente expresión.

$$\frac{\delta \varepsilon_{\mathrm{x}}}{\delta t} = \frac{\varepsilon_{\mathrm{x_t}} - \varepsilon_{\mathrm{x_{t\text{-}1}}}}{\delta t} = \frac{(x_{\mathrm{deseada_t}} - x_{\mathrm{medida_t}}) - (x_{\mathrm{deseada_{t\text{-}1}}} - x_{\mathrm{medida_{t\text{-}1}}})}{\delta t} \tag{4.22}$$

Si consideramos que la posición deseada en el instante t es igual a la posición deseada en el instante anterior t-1, condición que se cumple en la mayoría de los casos, podemos simplificar la ecuación 4.22 como:

$$\frac{-x_{\mathrm{medida_t}} + x_{\mathrm{medida_{t\text{-}1}}}}{\delta t} = -\frac{\delta x_{\mathrm{medida}}}{\delta t} = -\dot{x}_{\mathrm{medida}} \tag{4.23}$$

Introduciendo este cambio, las expresiones finales correspondientes a los cuatro PIDs son:

$$\dot{x}_{\mathrm{deseada}} = \varepsilon_{\mathrm{x}} K_{\mathrm{p}} - \dot{x}_{\mathrm{medida}} k_{\mathrm{d}} + \int \varepsilon_{\mathrm{x}} dt k_{\mathrm{i}}, \tag{4.24}$$

$$\dot{y}_{\mathrm{deseada}} = \varepsilon_{\mathrm{y}} K_{\mathrm{p}} - \dot{y}_{\mathrm{medida}} k_{\mathrm{d}} + \int \varepsilon_{\mathrm{y}} dt k_{\mathrm{i}}, \tag{4.25}$$

$$\dot{z}_{\mathrm{deseada}} = \varepsilon_{\mathrm{z}} K_{\mathrm{p}} - \dot{z}_{\mathrm{medida}} k_{\mathrm{d}} + \int \varepsilon_{\mathrm{z}} dt k_{\mathrm{i}} \text{ y} \tag{4.26}$$

$$\dot{\psi}_{\mathrm{deseada}} = \varepsilon_{\psi} K_{\mathrm{p}} - \dot{\psi}_{\mathrm{medida}} k_{\mathrm{d}} + \int \varepsilon_{\psi} dt k_{\mathrm{i}}. \tag{4.27}$$

donde se usan las estimaciones de velocidad proporcionadas, por el módulo de estimación del estado. Los valores $\mathrm{K_p}$, $\mathrm{K_d}$ y $\mathrm{K_i}$ son configurados manualmente (ver sección 5.1.2).

En la figura 4.5 se muestran las funciones principales de este módulo. Por defecto, tendremos una posición deseada inicial de coordenadas xyz (0, 0, 1) y orientación en ψ de 0 radianes.

- ***Obtener posición actual***: En esta función se obtiene la posición y la velocidad proporcionados por el estimador de estado.

- ***Cálculo velocidad deseada***: Se realizan los diferentes cálculos indicados en las ecuaciones 4.24, 4.25, 4.26 y 4.27 para obtener esta velocidad deseada. Además, se realiza una saturación del término integral con el objetivo de evitar que esto crezca desmesuradamente, y evitar así el efecto llamado *"windup"* [17].

- ***Rotación velocidad a coordenadas robot***: Las velocidades obtenidas a partir de los cálculos del controlador se obtienen en coordenadas mundo y se debe realizar una rotación para poder asignar las velocidades necesarias para alcanzar la posición deseada.

$$\begin{bmatrix} \dot{x}' \\ \dot{y}' \end{bmatrix} = \begin{bmatrix} cos(-\psi) & -sin(-\psi) \\ sin(-\psi) & cos(-\psi) \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{4.28}$$

Mediante esta matriz de rotación se conseguirá las velocidades deseadas en coordenadas robot.

- ***Publicación de la velocidad***: Se publicarán las diferentes velocidades deseadas obtenidas en coordenadas robot. Además, estas velocidades son previamente saturadas a un máximo valor de velocidad.

- ***Cambio posición deseada***: Si el robot llega a la posición deseada, el nodo de navegación envía una nueva posición deseada y esta es usada como la nueva consigna para los controladores PID.

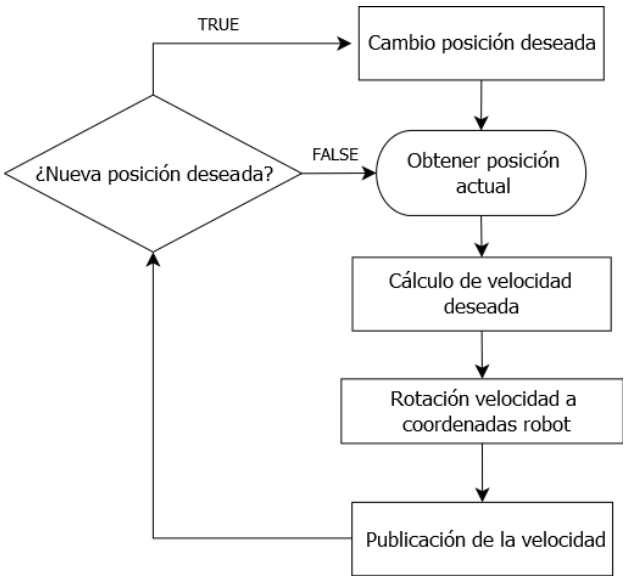


Figura 4.5: Diagrama de flujo del módulo del controlador de posición.

4.3.3 Módulo de navegación

Este módulo incorpora un sistema de navegación sencillo basado en una lista de navegación por puntos de paso. Una lista de navegación por puntos de paso consiste en un listado que contiene diferentes posiciones que debe alcanzar el robot donde no importa el camino que se recorra.

Este módulo básicamente contiene cuatro funciones:

- ***Obtener posición actual***: Observa la posición actual en la cuál se encuentra el vehículo.

- ***Comparación***: Compara la posición actual con la posición deseada en el preciso instante. Si el error de posición es menor a 20 cm en todos los ejes (X, Y, Z), y el error de orientación también es menor a 0.15 radianes, se considera que se ha alcanzado la posición deseada.

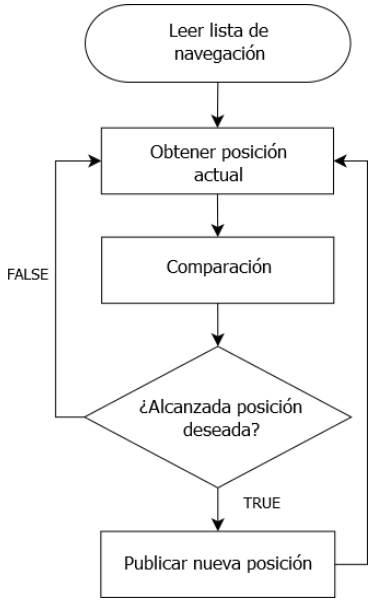


Figura 4.6: Diagrama de flujo del módulo del navegación.

- ***Leer lista de navegación***: Mediante un fichero, en el cual contiene las posiciones deseadas, el módulo obtiene todas estas posiciones y se guardan. Además, publica la primera posición deseada.

- ***Publicación nueva posición***: Publica la siguiente posición deseada de la lista de navegación.

En la figura 4.6 se muestra el orden en el que se ejecutan las anteriores funciones.

4.3.4 Módulo de evitación de obstáculos

Este es el último módulo implementado. La función de este módulo es que el robot alcance todos los objetivos que se encuentran en la lista de navegación esquivando los obstáculos que se encuentre en su recorrido. Para ello se ha implementado un algoritmo de evitación de obstáculos basado en campos de potencial.

Mediante esta técnica, el módulo de evitación de obstáculos se encarga de generar una velocidad de repulsión de cada uno de los obstáculos, cuya magnitud depende la proximidad de estos al vehículo. Cabe decir que en nuestra implementación, los puntos objetivo no generan ninguna fuerza de atracción. Para calcular esta velocidad de repulsión se utiliza la función sigmoidal inversa 4.29.

$$f(x) = 1 - \frac{1}{1 + e^{-\alpha_1 (x - \alpha_2)}}, \tag{4.29}$$

dónde, x es la distancia entre el dron y el obstáculo, y α_1 y α_2 son parámetros que se deben de configurar previamente.

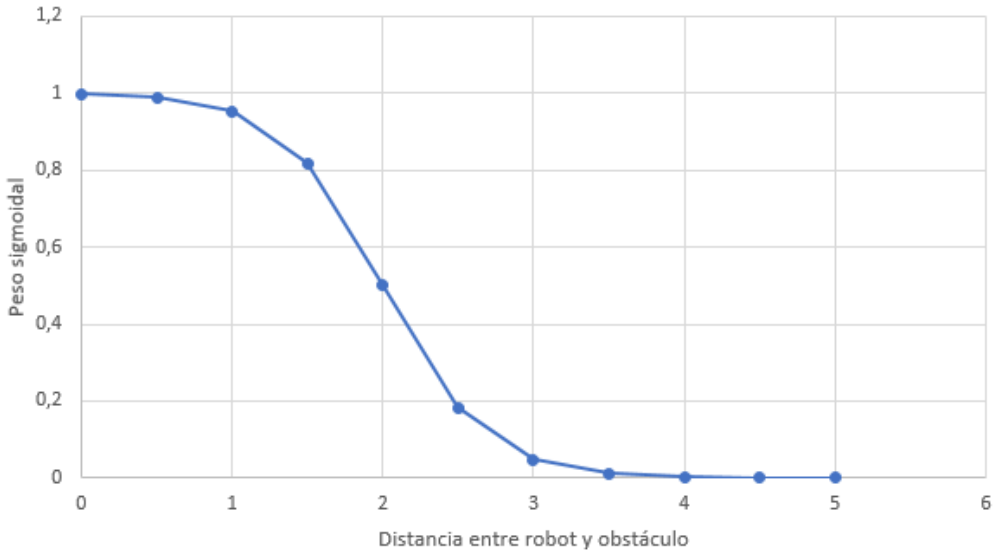


Figura 4.7: Ejemplo de función sigmoidal.

En la figura 4.7 se observa que dependiendo de la distancia entre el dron y el obstáculo, el resultado de la función sigmoidal se encuentra en un punto u otro del intervalo (0, 1). A partir de una distancia mayor a un cierto valor, el resultado de la función es 0, y a partir de una distancia menor a un cierto valor, el resultado es de 1. El resultado de la función se multiplica por la velocidad de repulsión deseada. Así, se consigue que a distancias lejanas los obstáculos no afecten a la trayectoria del robot, y que cuando el robot se encuentra cerca de un obstáculo se produzca una fuerza de repulsión que nunca excede el valor de la velocidad de repulsión estipulada.

En el caso de que haya diferentes obstáculos, se calcula la función sigmoidal para cada uno de los obstáculos, se suman estos pesos entre si y se multiplica por la velocidad de repulsión deseada.

Para detectar donde se encuentran los obstáculos se utilizara el mismo sistema de captura de movimiento. Para ello, se han colocado varios marcadores en los diferentes obstáculos.

La velocidad de repulsión obtenida se transformará en coordenadas robot con la misma matriz de rotación utilizada en el controlador de posición (ver ecuación 4.28).

Finalmente, se suman las velocidades proporcionadas por el controlador de posición a las velocidades de repulsión, usando una cierta ponderación a cada una de estas velocidades. La velocidad resultante es saturada a un cierto valor máximo para limitar la reacción del vehículo.

De esta manera, se obtiene una velocidad resultante con un cierto módulo y con su dirección y sentido dependientes de los obstáculos y del punto objetivo de ese instante.

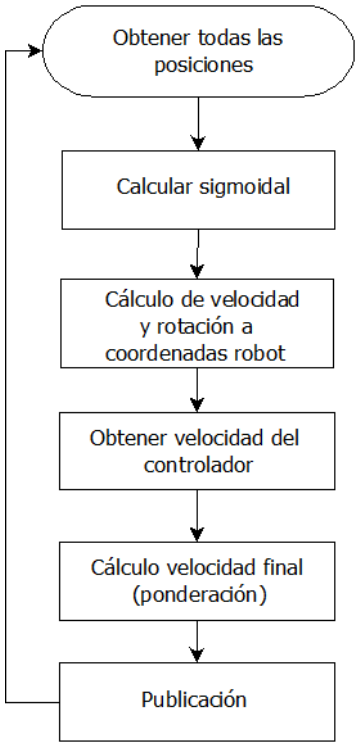


Figura 4.8: Diagrama de flujo del módulo de evitación de obstáculos.

En la figura 4.8 se muestra un esquema del módulo implementado, en la que podemos ver sus funciones principales:

- ***Obtener todas las posiciones***: En esta función se obtienen tanto las posiciones de los obstáculos como la del UAV.
- ***Calcular sigmoidal***: Cálculo de los pesos de la velocidades de repulsión a partir de la ecuación 4.29.
- ***Cálculo de velocidad y rotación a coordenadas robot***: Cálculo de la velocidad total producida por las velocidades de repulsión de todos los obstáculos y rotación de esta velocidad a coordenadas robot.
- ***Obtener velocidad del controlador***: Lectura y guardado de la velocidad publicada por el módulo del controlador.
- ***Cálculo velocidad final***: Obtención de la velocidad final a partir de la velocidad del controlador y la velocidad de repulsión con diferentes ponderaciones en cada una de estas.
- ***Publicación***: Publicación de la velocidad final.

CAPÍTULO 5

RESULTADOS EXPERIMENTALES

En este capítulo se discuten los diferentes experimentos llevados a cabo para verificar la corrección de los diferentes componentes desarrollados, precedidos por los procesos de configuración necesarios.

5.1 Procesos de configuración

Los módulos de estimación del estado, controlador de posición y evitación de obstáculos contienen diferentes parámetros que deben ser configurados previamente para conseguir un correcto posicionamiento del dron, realizando los desplazamientos a una velocidad y distancia a los obstáculos adecuadas.

Para la mayoría de las configuraciones ha sido necesario el uso de la herramienta *dynamic_reconfigure* [18]. Ésta proporciona una interfaz gráfica que permite cambiar los parámetros del nodo dinámicamente, sin necesidad de reiniciar el nodo.

En las siguientes secciones se detalla como se han configurado los parámetros de los diferentes módulos y que pruebas se han realizado hasta obtener la solución final.

5.1.1 Configuración del filtro de Kalman

Para el módulo de estimación de estado se deben configurar nueve parámetros:

- ***Período***: Este parámetro es el período de predicción. Este período indica cuándo el módulo de estimación de posición realizará solamente la etapa de predicción sin la de corrección. En otras palabras cuando se produce una ocultación, cada qué intervalo de tiempo se actualiza y se publica la posición.
- ***Aceleraciones***: Cuatro parámetros correspondientes a las aceleraciones en los ejes X, Y, Z y la rotación sobre el eje Z. Estos son necesarios para completar la matriz Q_t (ver ecuación 4.13).

- ***Covarianza del sensor***: En la matriz R_t (ecuación 4.15) se encuentran las covarianzas del sensor para todos los ejes de posición y para la rotación entorno el eje Z.

Sabiendo que el período de muestras de *OptiTrack* es de 1/120 segundos, nuestro período de predicción por ocultación debe ser mayor, y se ha configurado con un valor 3 veces superior (0.03 segundos). Por lo tanto, si no se detecta la posición del dron, se publica la posición del dron cada 0.03 segundos solamente utilizando la etapa de predicción.

Para configurar el resto de parámetros era necesario el uso de una interfaz gráfica, donde se pudiese observar la posición del UAV al largo del tiempo. Para ello, se ha pilotado el robot aéreo mediante el *joystick* sin la utilización de los otros módulos implementados.

Por una parte, las covarianzas del sensor son bajas dado que el sistema *OptiTrack* es un buen sistema de posicionamiento. En cuanto a las aceleraciones del dron, estos han sido fijados a 0.2 m/s^2.

Mediante una gráfica que muestra las posiciones y la rotación entorno al eje Z en función del tiempo, configuramos estos 8 parámetros. Debido a que el posicionamiento usando *OptiTrack* es muy preciso rápidamente se hallaron los parámetros correspondientes a las covarianzas del sensor. Para la configuración de los parámetros de las aceleraciones, se han provocado ocultaciones al sistema de captura de movimiento, para que solo se ejecute la etapa de predicción. Este proceso ha sido repetido hasta encontrar un valor adecuado para las aceleraciones.

5.1.2 Configuración del controlador de posición

El controlador de posición requiere de la configuración de doce parámetros.

Estos corresponden a los parámetros K_p, K_d y K_i para cada uno de los cuatro controladores PID (véanse ecuaciones 4.24 a 4.27)

La configuración de estos parámetros se ha hecho de dos formas: la primera consiste en mantener la posición actual del robot aéreo y la segunda en establecer un punto objetivo para observar como se alcanza ese punto.

El proceso de configuración se ha llevado a cabo por fases. Inicialmente se han configurado los parámetros K_p, K_d y K_i del eje de posición X manteniendo los otros inactivos. Luego se ha realizado el mismo proceso para el eje de posición Y. La configuración de ambos ejes son relativamente iguales, ya que se emplea la misma dinámica para desplazarse en estos dos ejes. En tercer lugar se ha configurado el PID correspondiente al eje Z y, finalmente, el PID encargado de la orientación.

Cada PID ha sido configurado empezando por el parámetro K_p. Este ha sido ajustado para obtener un comportamiento ligeramente oscilatorio entorno al punto deseado

(véase figura 5.1 [A]). Luego se ha establecido un cierto valor a K_d para lograr la posición deseada sin oscilaciones pronunciadas (véase figura 5.1 [B]). Por último, si es necesario se ha incorporado un cierto valor de K_i cuando se observa error en el estacionario (véase figura 5.1 [C]).

El experimento que se muestra en la figura 5.1 consiste en mantener el dron en el punto 0 del eje X.

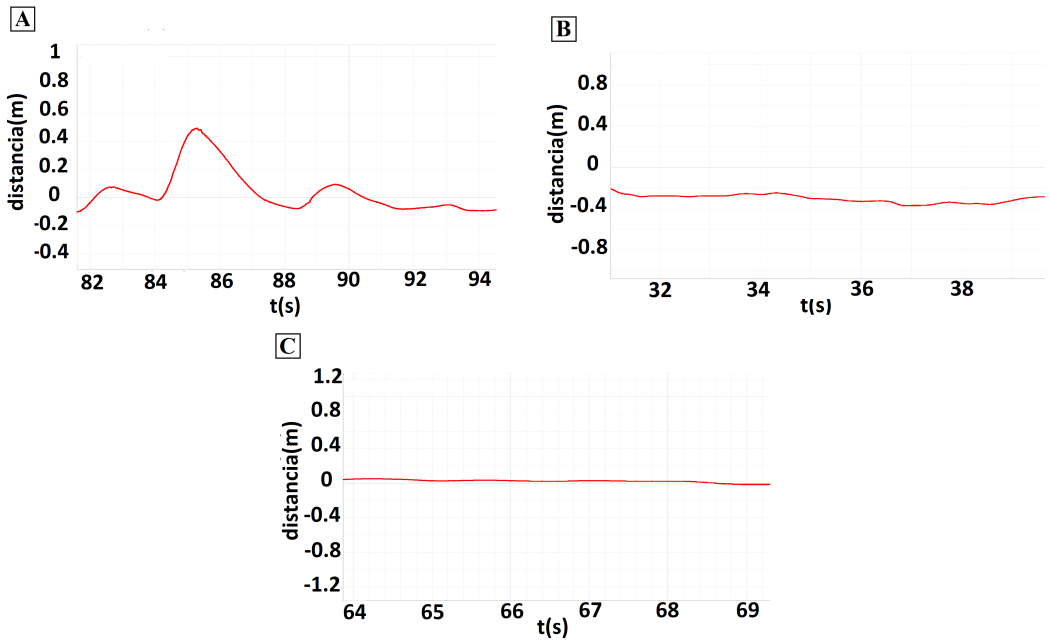


Figura 5.1: Configuración del controlador PID: posición obtenida con K_p [A], posición obtenida con K_p y K_d [B] y posición obtenida con K_p, K_d y K_i [C].

Además de los parámetros de los PIDs, se han fijado los valores de velocidad máxima y el valor máximo para los términos integrativos.

5.1.3 Configuración del módulo de evitación de obstáculos

Los principales parámetros del módulo de evitación de obstáculos son los correspondientes a los de la función sigmoidal (ecuación 4.29).

La configuración de estos valores se realizan mediante una hoja de cálculo, en la cual se representa una gráfica que muestra el peso de la sigmoidal en función de la distancia entre el dron y el obstáculo. Dependiendo del valor de los parámetros se varia el peso de la velocidad de repulsión. También, se debe configurar el valor de la ponderación. Este valor, en el intervalo (0, 1), fija el peso de la velocidad de repulsión, y el restante corresponde al peso de la velocidad del controlador.

A modo de ejemplo, siendo, 0.65 el peso de la velocidad de repulsión, el peso de la velocidad del controlador resulta 1-0.65 = 0.35. Esta ponderación se aplica una vez

obtenido el valor total de repulsión resultante de la suma de todas las sigmoidales.

En la figura 5.2 se observa un ejemplo de una gráfica realizada para los valores de α_1 = 1.15 y para el de α_2= 10.

Una vez escogido el valor de los parámetros se han realizado distintas pruebas de evitación de obstáculos sobre el simulador para comprobar la asignación de los parámetros. Durante la configuración de estos parámetros se han tenido en cuenta las dimensiones del *AR.Drone* y la de los obstáculos, para que no se produzca ninguna colisión.

Por último, se debe de configurar la velocidad máxima final que puede alcanzar el cuadricóptero.

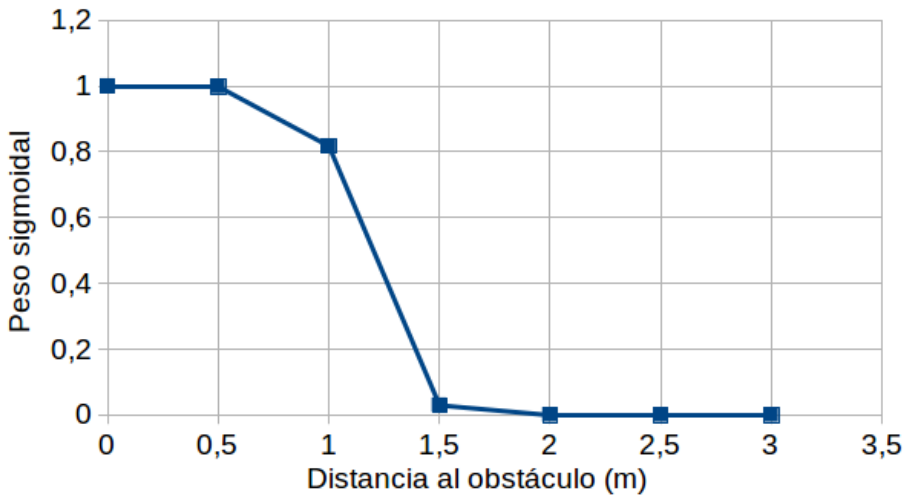


Figura 5.2: Ejemplo de configuración de parámetros del módulo de evitación de obstáculos.

5.2 Metodología

La evaluación experimental del comportamiento de los diferentes módulos implementados se ha realizado primeramente usando el simulador (sección 5.3) y posteriormente, usando el vehículo real (sección 5.4). Para ambos casos se presentan gráficos con los resultados obtenidos.

Algunas de ellas contienen unos símbolos en forma de rombo que representan la posición inicial y otros en forma de triángulo que representan la posición final del recorrido. En el caso de haber solamente un rombo, la posición inicial y final del recorrido es la indicada por el rombo. Además, mediante líneas de distinto color se representa: la trayectoria deseada por el dron (verde), las distintas posiciones recorridas por el dron (azul) y la posición del dron cuando se producen ocultaciones o trayectorias realizadas por obstáculos (rojo).

Diferenciamos tres tipos de experimentos:

- ***Experimentos básicos*** : Son aquellos en los cuales no se producen ningún tipo de ocultación y no se encuentran ningún obstáculo en la trayectoria que describe el UAV.

- ***Experimentos con ocultaciones*** : Son aquellos en los cuáles se producen algún tipo de ocultación puntual.

- ***Experimentos de evitación de obstáculos***: En estos experimentos se encuentran diversos obstáculos donde el dron deberá evitarlos para alcanzar los puntos objetivos. Los obstáculos fijos se representan con cuadrados rojos.

5.3 Experimentos con el simulador

En esta sección se muestra todos los experimentos realizados en el presente proyecto sobre el simulador

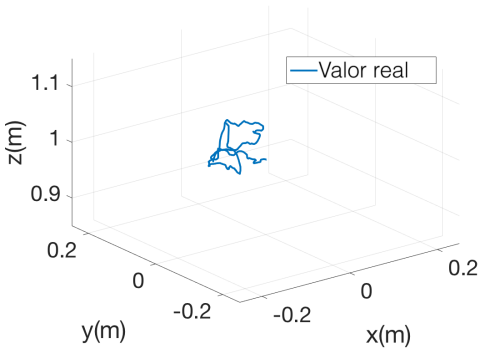
5.3.1 Experimentos básicos

En esta sección se han realizado los siguientes experimentos:

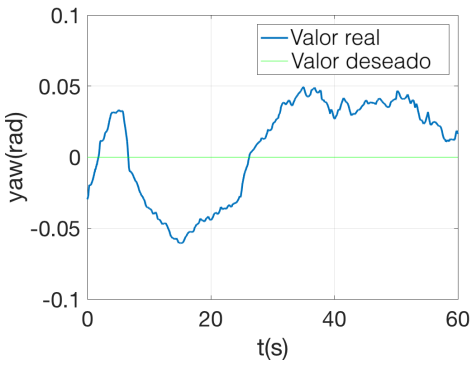
- ***Hovering***: Este experimento consiste en mantener el dron en una determinada posición durante un tiempo determinado. La posición escogida es a un metro de altura y los otros valores en el origen.

  En las gráficas 5.3a y 5.3a se presenta la posición y orientación estimadas durante un tiempo total de 60 segundos. En la primera de ellas se representa la posición y en la segunda la orientación a lo largo del tiempo.

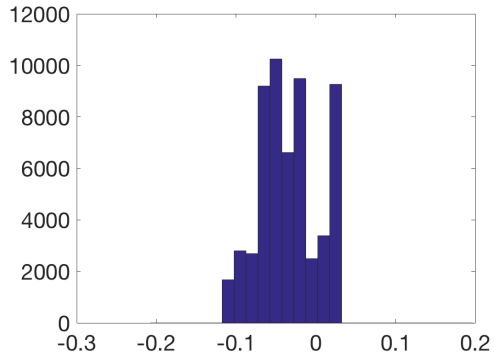
  Las figuras 5.3c, 5.3d, 5.3e y 5.3f corresponden a los histogramas de error. Se puede observar en todos los histogramas los valores se encuentran entre el intervalo (-0.1, 0.1).



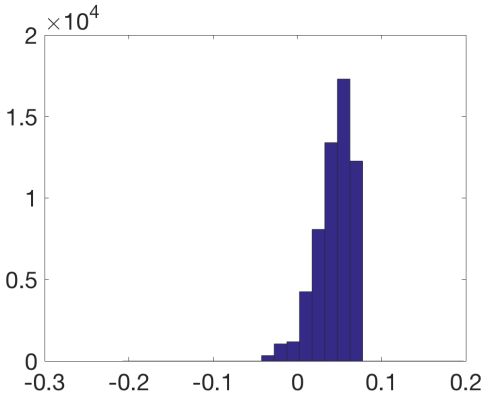
(a) Posición.



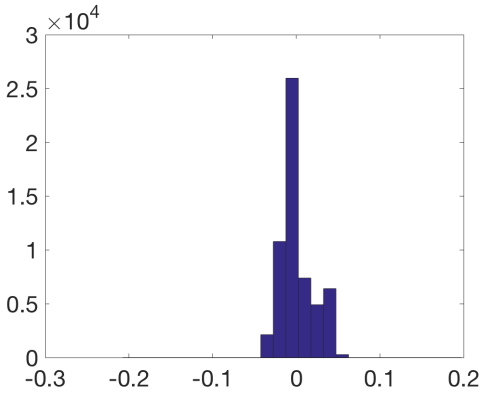
(b) Orientación en función del tiempo.



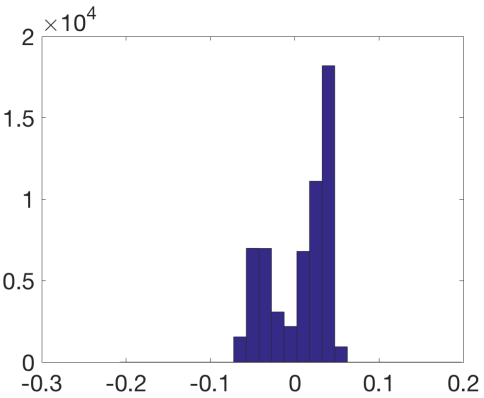
(c) Histograma del error en x.



(d) Histograma del error en y.



(e) Histograma del error en z.



(f) Histograma del error en ψ.

Figura 5.3: Experimento de *hovering* mediante el simulador.

- ***Desplazamiento a lo largo de un solo eje***: En este experimento se realiza un recorrido solamente en un eje hasta un punto objetivo. El robot aéreo comienza en la posición inicial x = 0, y = -1.5, z = 0, ψ = 0 hasta la posición final x = 0, y = 1.5, z = 1, ψ = 0.

  La figura 5.4 muestra el experimento realizado. Se observa que las posiciones iniciales y finales no son iguales las obtenidas a las deseadas, esto es debido al

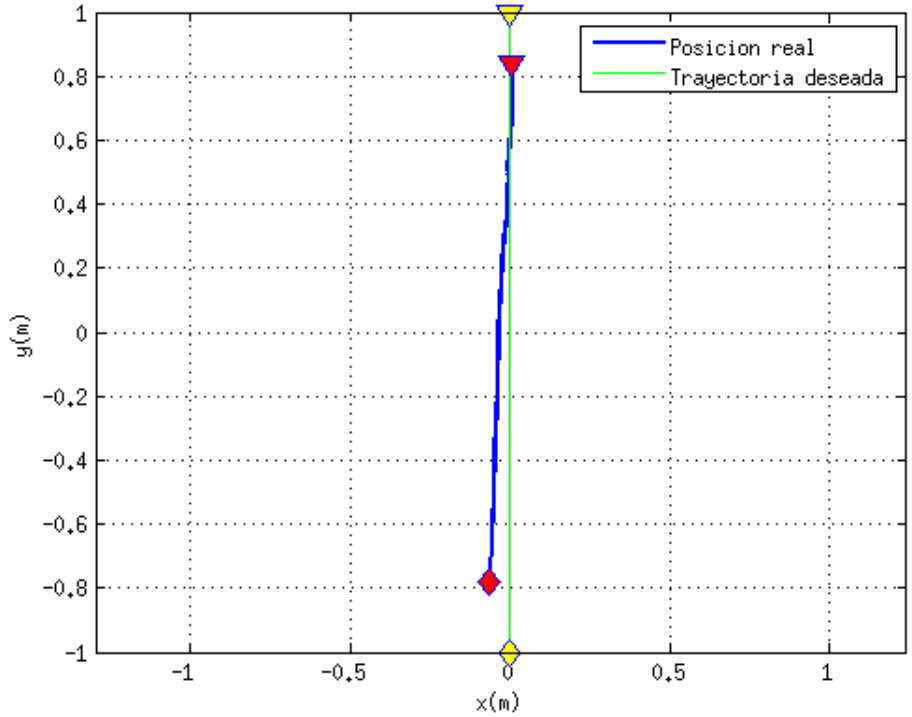


Figura 5.4: Desplazamiento a lo largo de un solo eje en el simulador.

margen permitido por el algoritmo de navegación (véase sección 4.3.3).

- ***Giro de 360º sobre el eje Z manteniendo la posición inicial***: Este experimento consiste en realizar un giro del dron sobre si mismo alrededor del eje Z de 360º manteniendo siempre la misma posición.

  En la figura 5.5 se observa este giro realizado por el dron. En la figura 5.6 se muestran los posibles valores de orientación.

- ***Diferentes recorridos en forma de cuadrado***: Estos experimentos se han realizado para observar como se desplazaba el dron en el entorno.

  En un primer experimento se ha realizado una trayectoria en forma de cuadrado manteniendo una altura fija. Este experimento se muestra en la figura 5.7 [A], donde podemos observar que solo hay un rombo amarillo debido a que el mismo punto indica el inicio y el final de la trayectoria. Un segundo experimento, correspondiente a la figura 5.7 [B], se realiza una trayectoria parecida a la anterior, pero este caso realizando un cuadrado en el plano YZ. Finalmente, el experimento ilustrado en la figura 5.7 [C] muestra un vuelo siguiendo una trayectoria correspondiente a un doble cuadrado en l plano horizontal (XY), en la que el primer cuadrado se realiza a 1m de altura y el segundo a 1.5m.

Las estimaciones de posición obtenidas durante los experimentos se asemejan a las posiciones deseadas, por lo que, podemos concluir que las configuraciones del controlador de posición, y del módulo de navegación son adecuadas.

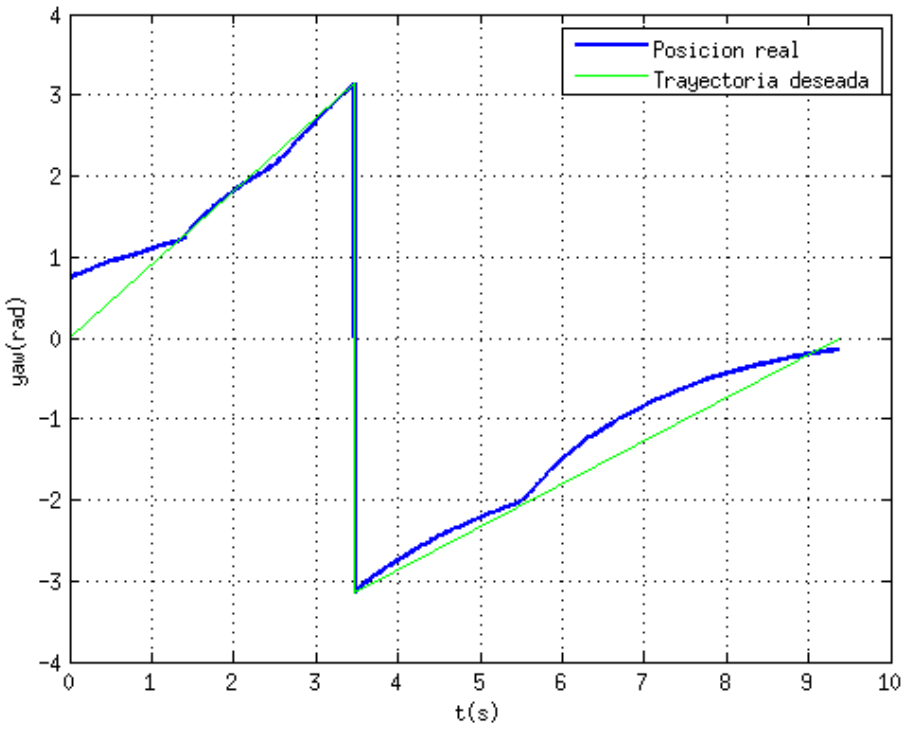


Figura 5.5: Vuelta completa alrededor del eje Z en el simulador.

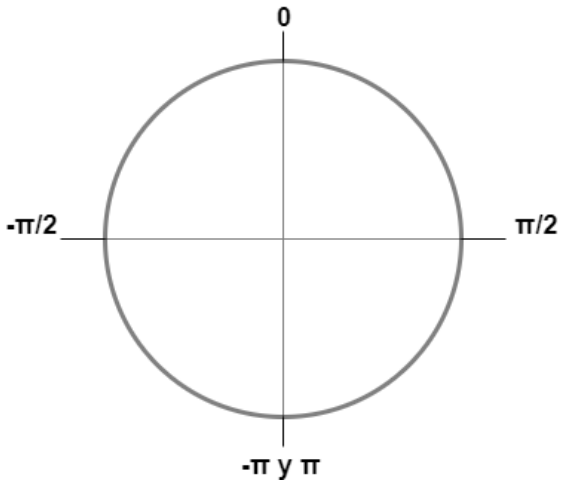


Figura 5.6: Representación de los posibles valores de orientación.

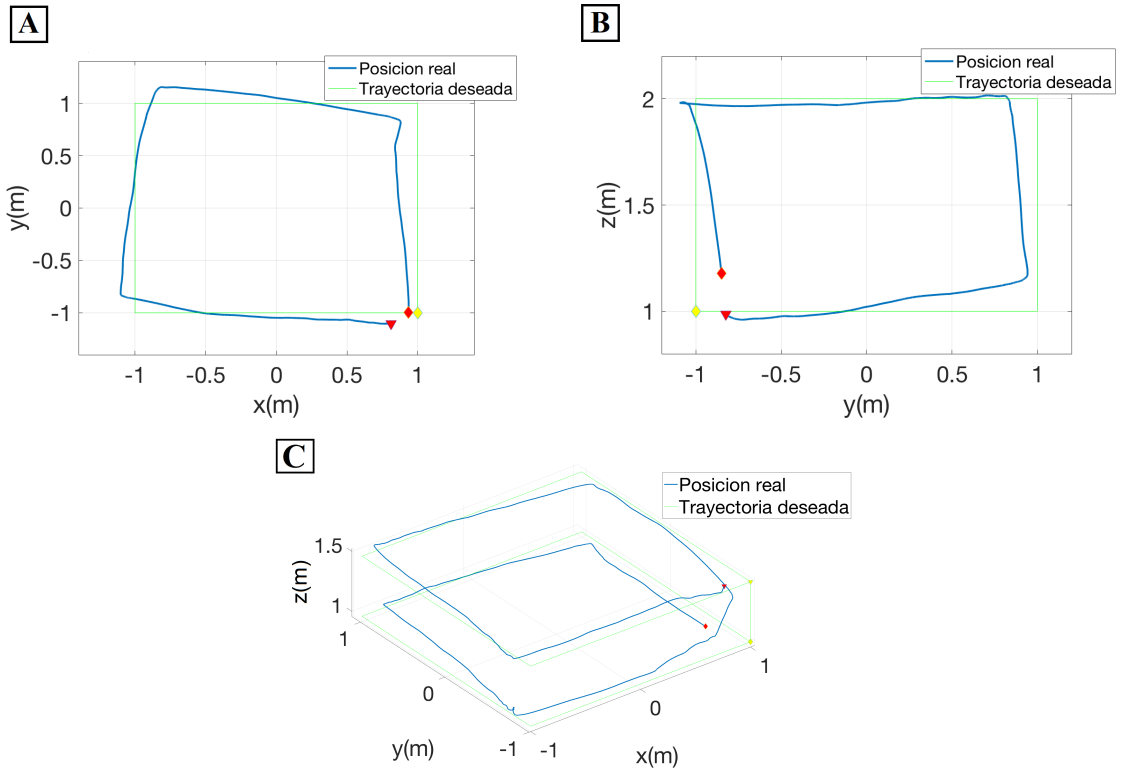


Figura 5.7: Diferentes recorridos por el entorno en el simulador: cuadrado en el plano XY [A], cuadrado en el plano YZ [B] y doble cuadrado en el plano XY y cambio de altura entre cuadrados [C].

5.3.2 Experimentos con ocultaciones

Con el objetivo de simular ocultaciones con el simulador hemos forzado que este no publique la posición del robot aéreo durante un determinado intervalo de tiempo. Cuando se produce una ocultación el módulo de estimación del estado solamente realiza la etapa de predicción y publicación. De esta manera podemos comprobar si se asemeja la posición predecida con la posición con corrección cuando se retoma la publicación por parte del simulador.

Este experimento 5.8 simula una ocultación, durante 2 segundos. La línea azul muestra la posición cuando se publica la posición del robot mientras que la línea roja es la posición predicha por el filtro de *Kalman* durante la ocultación. Se puede observar como la ultima posición obtenida mediante la etapa de predicción es similar a la posición cuando se vuelve a publicar la posición del dron.

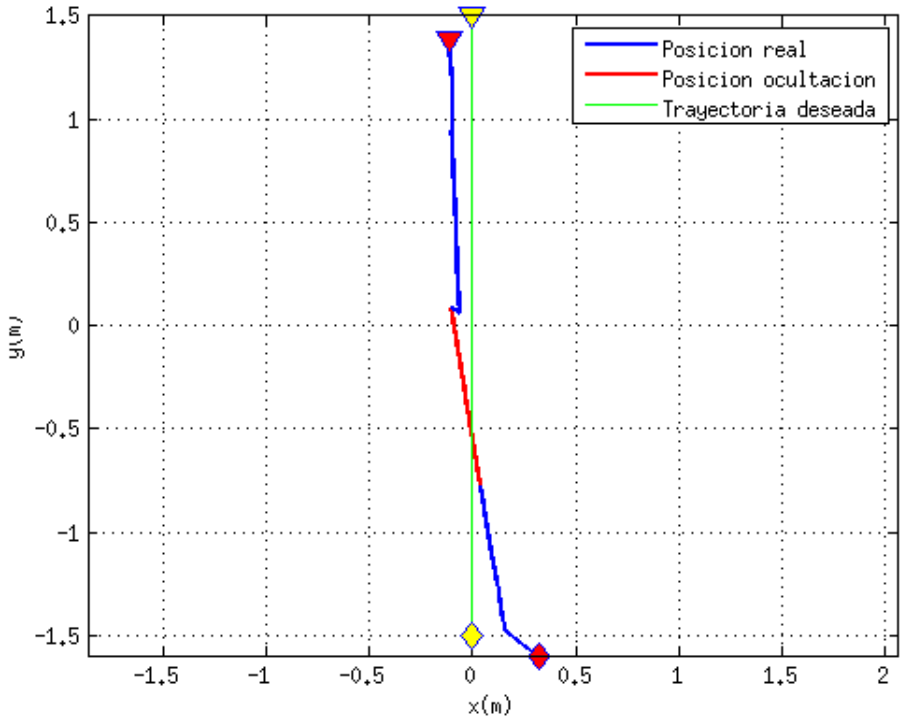


Figura 5.8: Ocultación durante 2 segundos producida en el simulador en un desplazamiento a lo largo de un solo eje.

5.3.3 Experimentos de evitación de obstáculos

En esta sección se incluyen un conjunto de experimentos evaluar el rendimiento del módulo de evitación de obstáculos. Para realizar los obstáculos en el simulador se ha implementado un rectángulo de dimensiones 0.4 x 0.4 x 2 metros.

- ***Evitación de un obstáculo fijo en la trayectoria del dron***: Este experimento muestra como el dron es capaz de esquivar un obstáculo que se encuentra dentro de su trayectoria.

  En la figura 5.9 se muestra el experimento mencionado previamente, donde el cuadrado representa el obstáculo. Se observa que la posición deseada seria una linea recta, pero debido al obstáculo, el robot realiza un cambio en su trayectoria para no colisionar con éste y llegar a la posición y = 1.5 para luego volver a su posición inicial.

- ***Evitación de un obstáculo fijo cerca de la trayectoria del dron***: En este experimento se realiza una trayectoria cuadricular en la cual habrá algunos obstáculos cerca de esta trayectoria que provocarán que el dron se desvié puntualmente de su recorrido.

  En la figura 5.10 se aprecia que la trayectoria realizada por el dron se ve afectada por los dos obstáculos que se encuentran cerca de su trayectoria, impidiendo que el dron realice una trayectoria cuadricular.

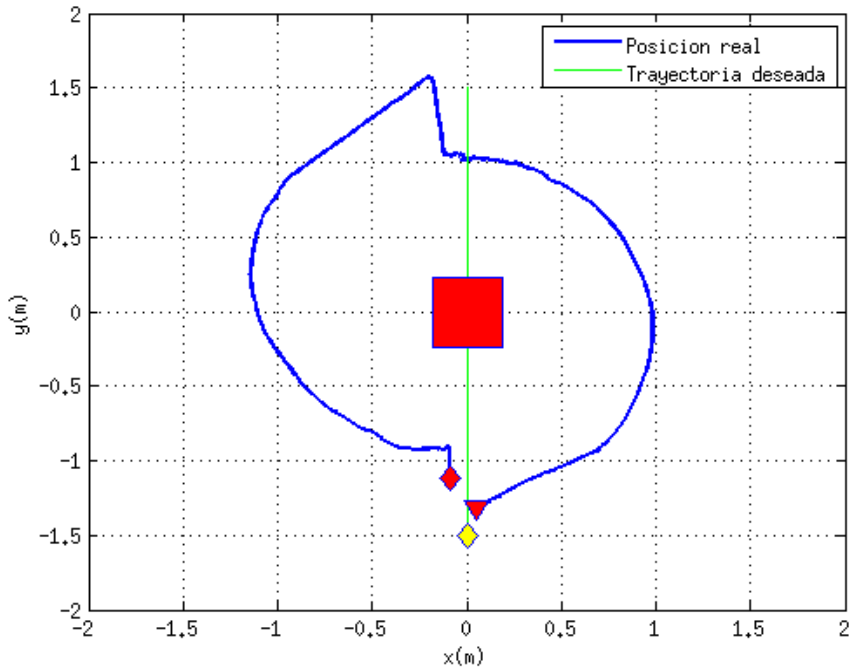


Figura 5.9: Evitación de un obstáculo fijo dentro de la trayectoria del robot en el simulador.

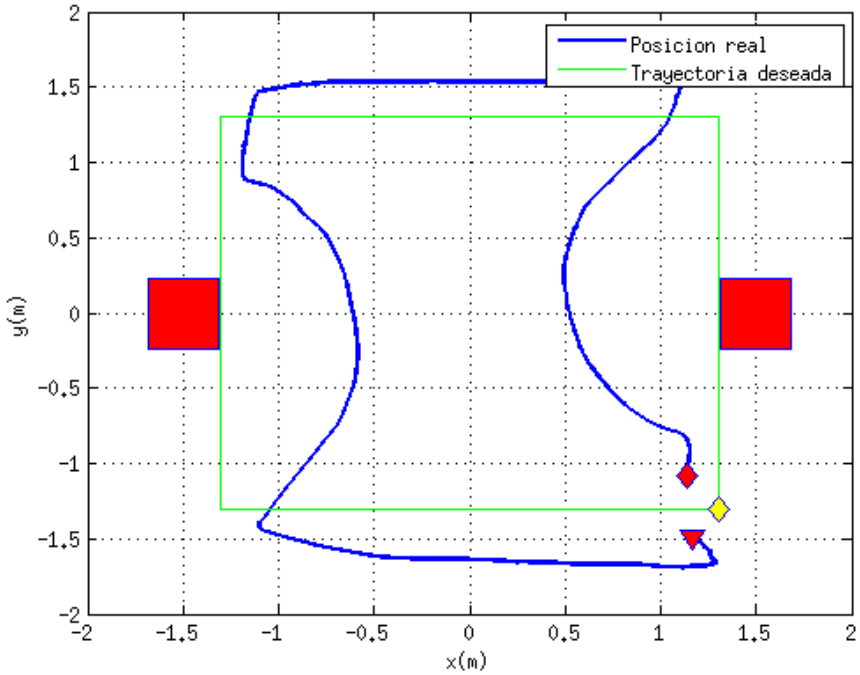


Figura 5.10: Evitación de diferentes obstáculos fijos cerca de la trayectoria del robot en el simulador.

- ***Evitación de un obstáculo móvil en la trayectoria del dron***: Este experimento muestra como el dron es capaz de esquivar un obstáculo móvil que, durante un determinado intervalo de tiempo, se encuentra dentro de su trayectoria.

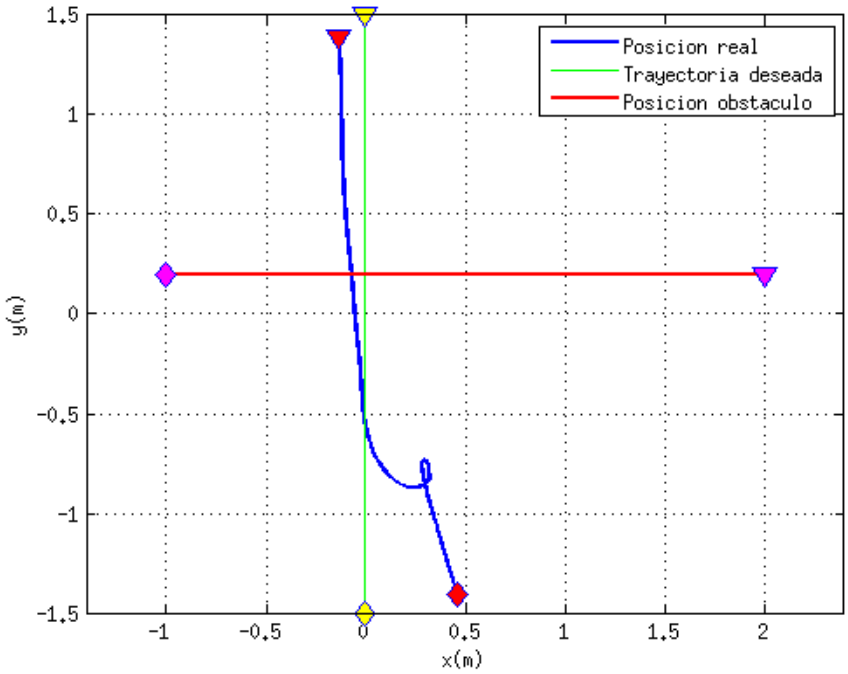


Figura 5.11: Evitación de un obstáculo móvil dentro de la trayectoria del robot en el simulador.

Si observamos la figura 5.11 se puede ver las trayectorias del dron y del obstáculo durante este experimento. La línea roja muestra el recorrido efectuado por el obstáculo mientras que la línea azul muestra el recorrido efectuado por el dron.

La figura 5.12 muestra trayectorias parciales realizadas durante el experimento. Podemos observar que el dron no colisiona con el obstáculo, modificando su trayectoria de la manera requerida.

Con todos estos experimentos se concluye que el módulo de evitación de obstáculos presenta un buen rendimiento en el entorno de simulación.

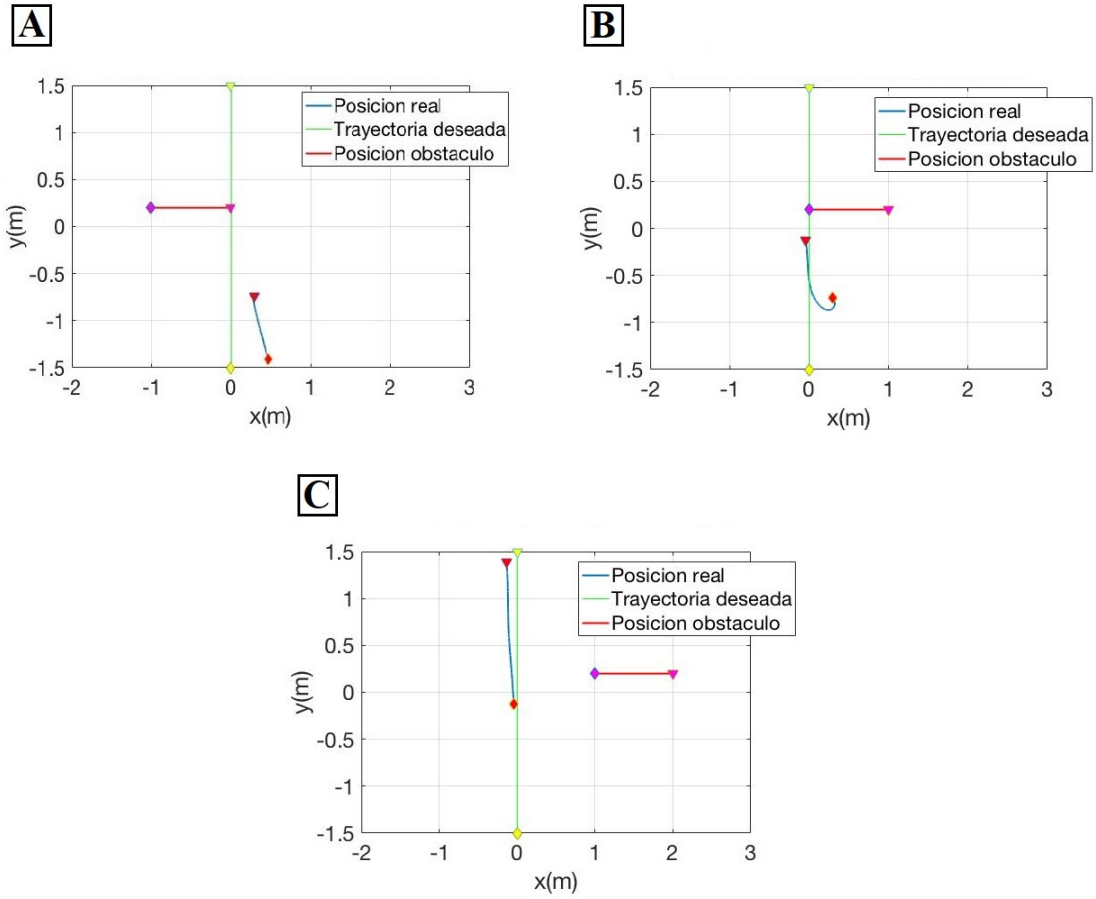


Figura 5.12: Evitación de un obstáculo móvil dividido en diferentes intervalos de tiempo con el simulador: primer intervalo [A], segundo intervalo [B] y último intervalo [C].

5.4 Experimentos con el modelo real

Una vez realizados todos los experimentos sobre el simulador se han realizado los experimentos sobre el modelo real.

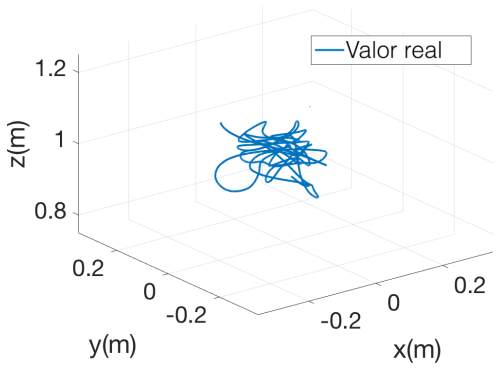
5.4.1 Experimentos básicos

A continuación se muestra los experimentos básicos realizados sobre el modelo real:

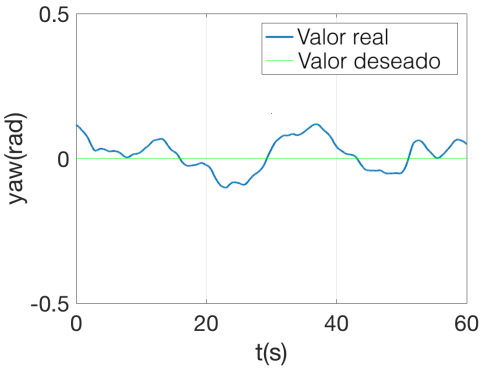
- ***Hovering***: Este experimento consiste en mantener el dron en una determinada posición durante un tiempo determinado. Se ha escogido la misma posición y el mismo tiempo que el empleado en el simulador.

  Las gráficas representadas son las mismas que se han representado en el experimento *hovering* mediante el simulador.

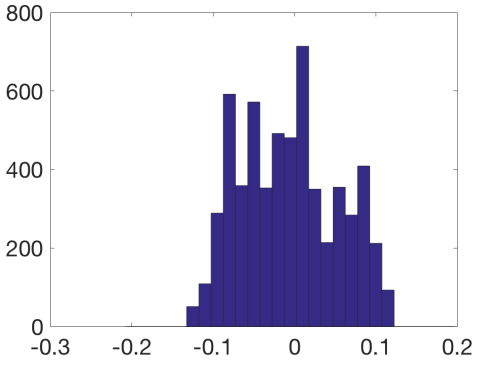
  En las figuras 5.13a y 5.13b se muestra como se ha conseguido realizar un *hovering* satisfactorio con éxito. En las figuras 5.13c, 5.13e y 5.13f sus valores se encuentran entre el intervalo (-0.1, 0.1), en cambio en el histograma 5.13d el



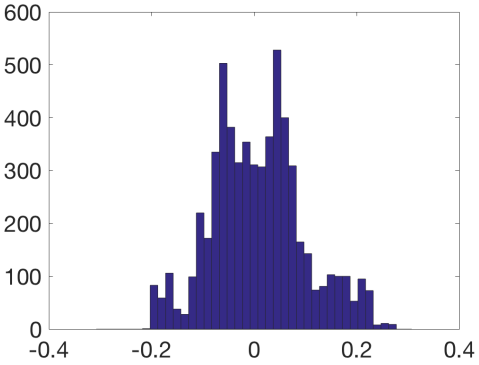
(a) Posición.



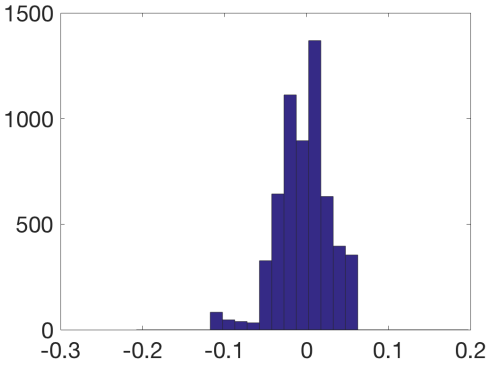
(b) Orientación en función del tiempo.



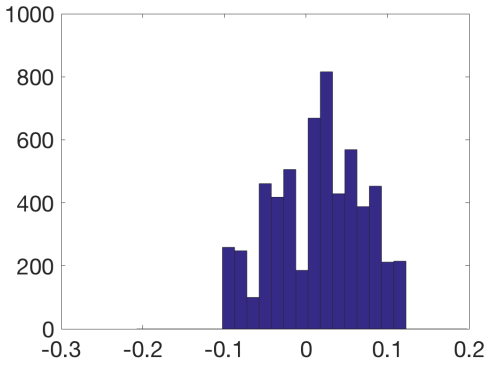
(c) Histograma del error en x.



(d) Histograma del error en y.



(e) Histograma del error en z.



(f) Histograma del error en ψ.

Figura 5.13: Experimento de *hovering* con el modelo real.

intervalo corresponde a (-0.2, 0.2). Los intervalos correspondientes al robot real son superiores a los de la simulación.

- ***Desplazamiento a lo largo de un solo eje***: En este experimento se realiza un recorrido desplazándose en un solo eje hasta un punto objetivo. El punto inicial y final corresponden a la misma posición usada en el experimento realizado sobre el simulador.

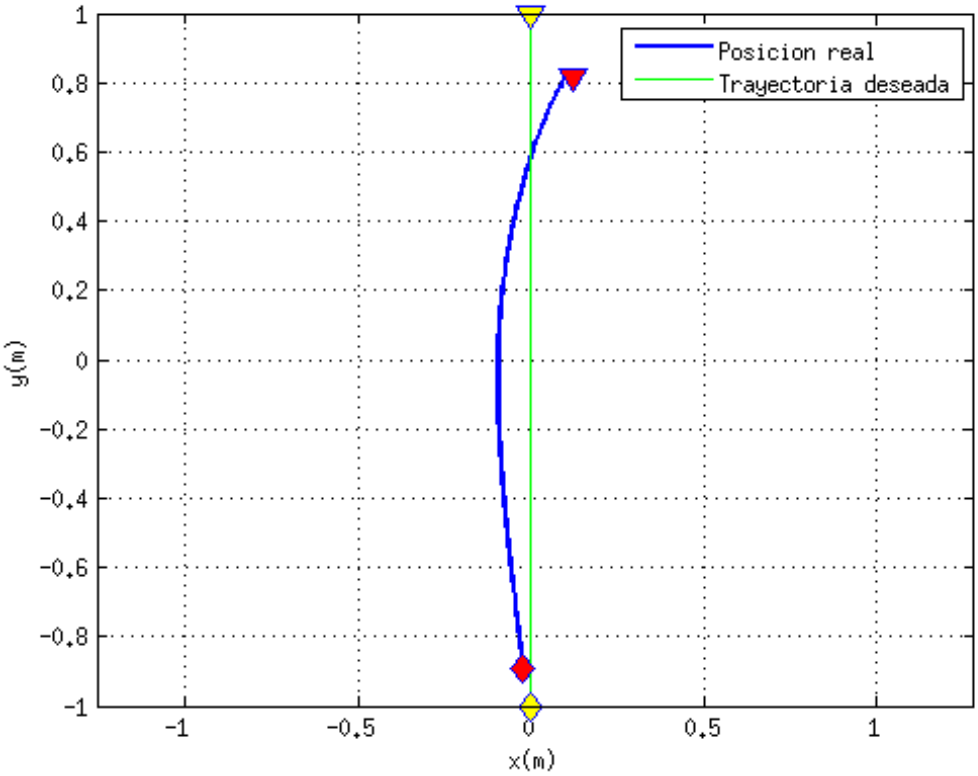


Figura 5.14: Desplazamiento a lo largo de un solo eje con el modelo real.

La figura 5.14 muestra el experimento realizado. En el cual el dron se desplaza por el eje Y hasta el punto objetivo. En la figura se aprecia un pequeño desplazamiento en el eje X debido al incremento en el intervalo de tolerancia.

- ***Giro de 360º sobre el eje Z manteniendo la posición inicial***: Como se ha realizado sobre el simulador, se realiza una vuelta completa del robot real sobre sí mismo.

  En la figura 5.15 se observa el giro realizado por el robot en el modelo real. En ella se puede ver el cambio gradual de orientación de 3.14 a -3.14, tal y como ocurre usando el simulador.

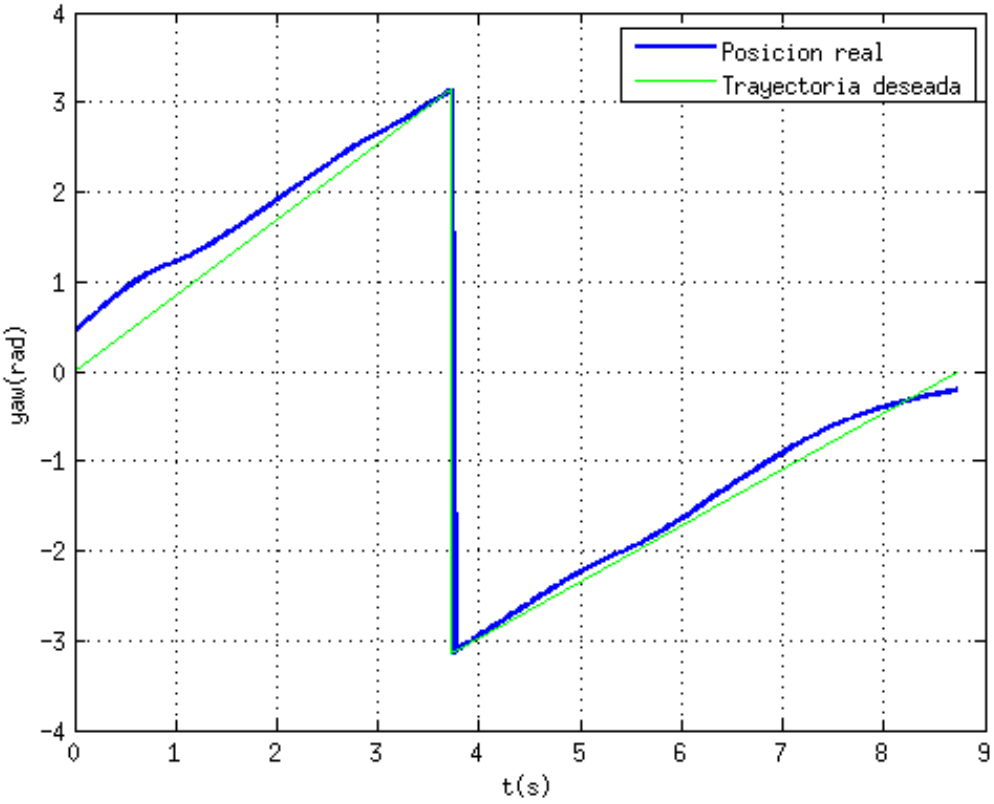


Figura 5.15: Vuelta completa alrededor del eje Z con el modelo real.

- ***Diferentes recorridos en forma de cuadrado***: Estos experimentos muestran como se desplaza el dron por el entorno.

  Un primer experimento (figura 5.16 [A]) corresponde a una trayectoria cuadricular manteniendo la altura fija. Un segundo experimento (figura 5.16 [B]) se realiza una trayectoria cuadricular en el plano YZ. En él se puede observar como en algunos puntos objetivos el cuadricóptero ha tenido que retroceder para poder alcanzarlos. El último experimento (figura 5.16 [C]) corresponde a una trayectoria realizando un doble cuadrado, el primero a 1 m y el segundo a 2 m. En él también podemos apreciar un retroceso cuando el vehículo pretende alcanzar algunos de los puntos objetivos.

Podemos concluir que los experimentos básicos realizados con el vehículo real se asemejan a los realizados mediante el simulador.

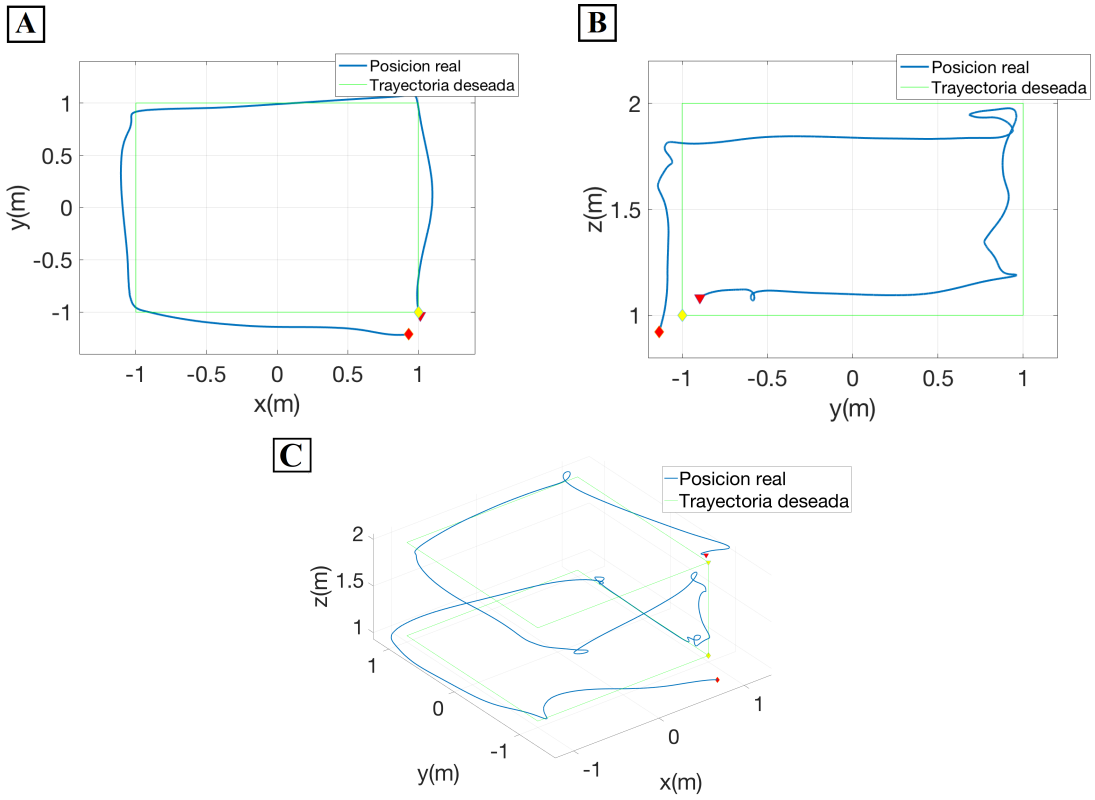


Figura 5.16: Diferentes recorridos por el entorno con el modelo real: cuadrado en el plano XY [A], cuadrado en el plano YZ [B] y doble cuadrado en el plano XY y cambio de altura entre cuadrados [C].

5.4.2 Experimentos con ocultaciones

Se ha realizado el mismo experimento de ocultación que en el simulador, impidiendo la publicación del estado del dron durante 2 segundos.

La figura 5.17 muestra el experimento. Si se compara con el experimento realizado en el simulador (figura 5.8), el resultado obtenido es semejante: la ultima posición obtenida realizando solamente la etapa de predicción es similar a la posición obtenida una vez se vuelve a publicar la posición del sensor.

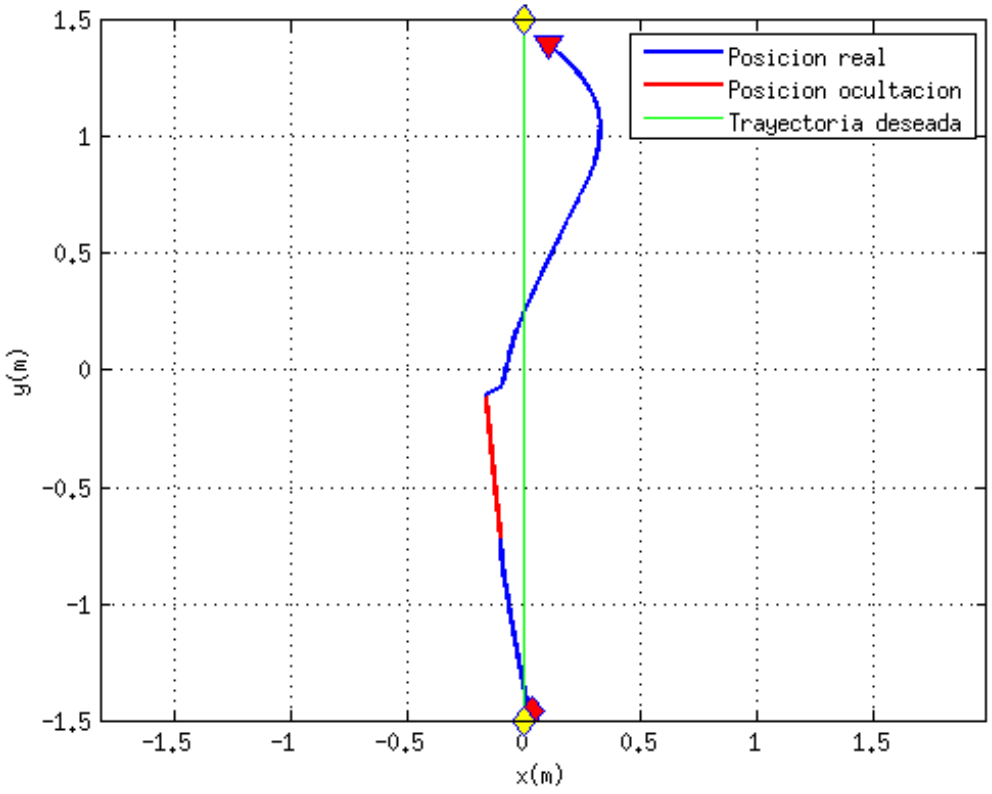


Figura 5.17: Ocultación durante 2 segundos producida en el modelo real en un desplazamiento a lo largo de un solo eje.

5.4.3 Experimentos de evitación de obstáculos

En esta sección se ha repetido los experimentos realizados con obstáculos en el simulador. Los obstáculos que se deben esquivar son unas cajas de cartón de una dimensiones 0.4 x 0.4 metros, donde la altura se considera infinita.

- ***Evitación de un obstáculo fijo en la trayectoria del dron***: En este experimento muestra como el dron es capaz de esquivar un obstáculo que se encuentra dentro de su trayectoria.

  En la figura 5.18 se muestra la trayectoria del dron y la posición de los obstáculos. Se puede observar que el robot aéreo esquiva el obstáculo sin ninguna dificultad dejando una distancia de seguridad entre el obstáculo y éste.

- ***Evitación de un obstáculo fijo cerca de la trayectoria del dron***: El experimento muestra la influencia de los obstáculos si se encuentran cerca de la trayectoria del UAV.

  La figura 5.19 muestra como la trayectoria realizada por el dron se ve afectada por los dos obstáculos cercanos.

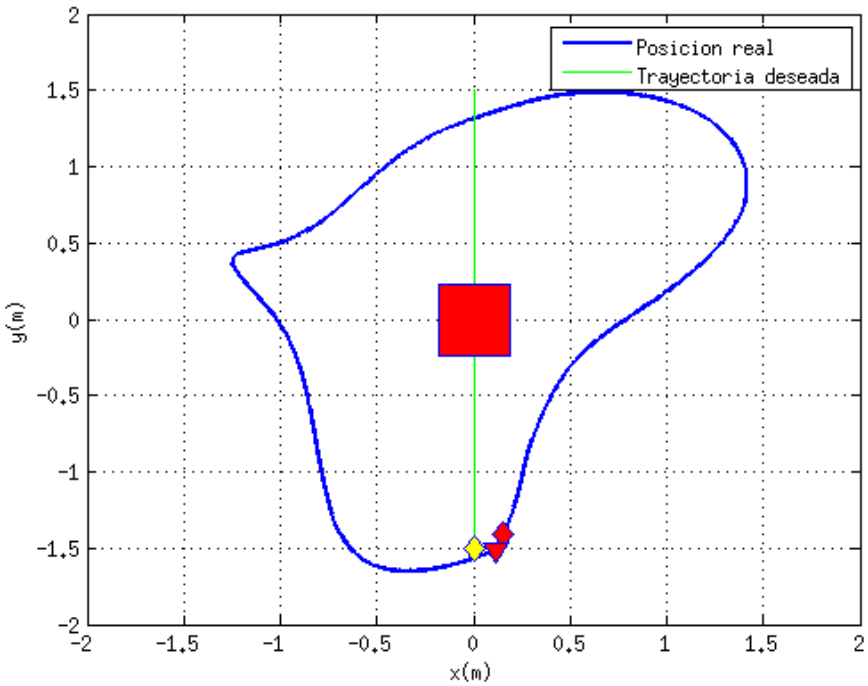


Figura 5.18: Evitación de un obstáculo fijo dentro de la trayectoria del robot con el modelo real.

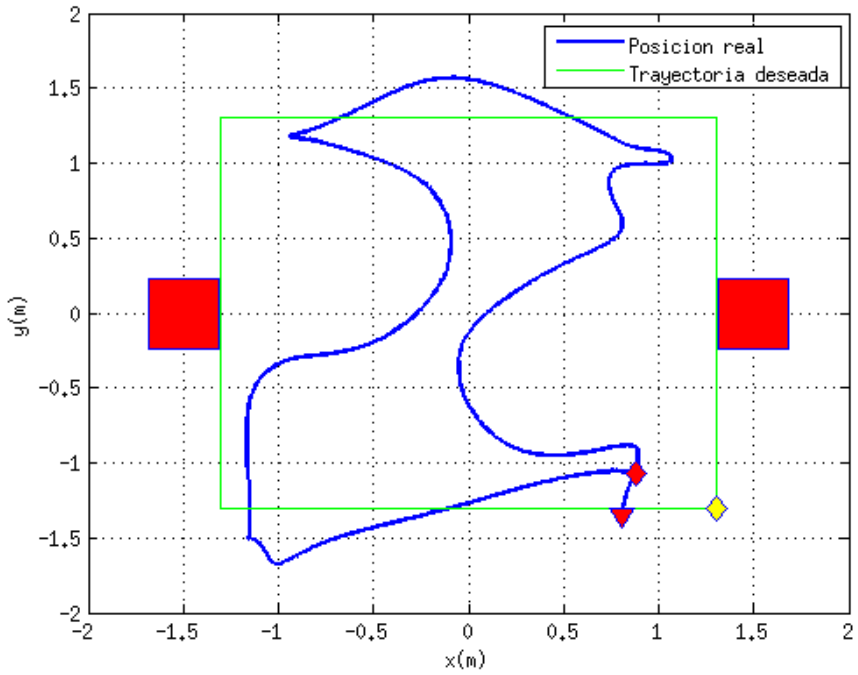


Figura 5.19: Evitación de diferentes obstáculos fijos cerca de la trayectoria del robot con el modelo real.

- ***Evitación de un obstáculo móvil en la trayectoria del dron***: Este experimento muestra como el dron es capaz de esquivar un obstáculo móvil. Para ello se ha movido un obstáculo hasta la trayectoria del cuadricóptero para comprobar su comportamiento.

  La figura 5.20 muestra el experimento, donde la línea roja muestra el recorrido efectuado por el obstáculo y la línea azul el recorrido efectuado por el dron.

  La figura 5.21 muestra el mismo experimento dividido en diferentes intervalos de tiempo. En ellos se puede observar como el vehículo se aleja del obstáculo cuando este se interpone en su trayectoria.

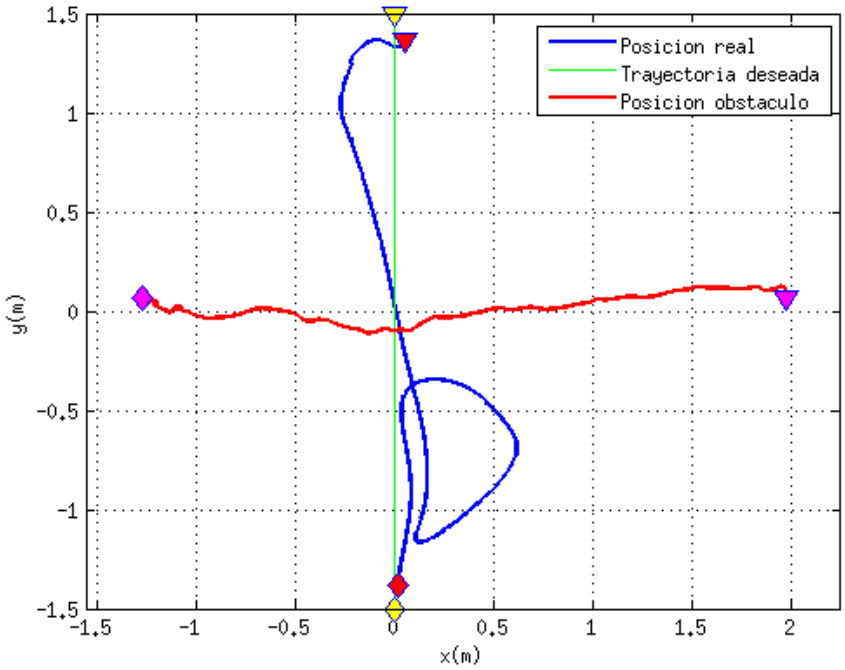


Figura 5.20: Evitación de un obstáculo móvil dentro de la trayectoria del robot con el modelo real.

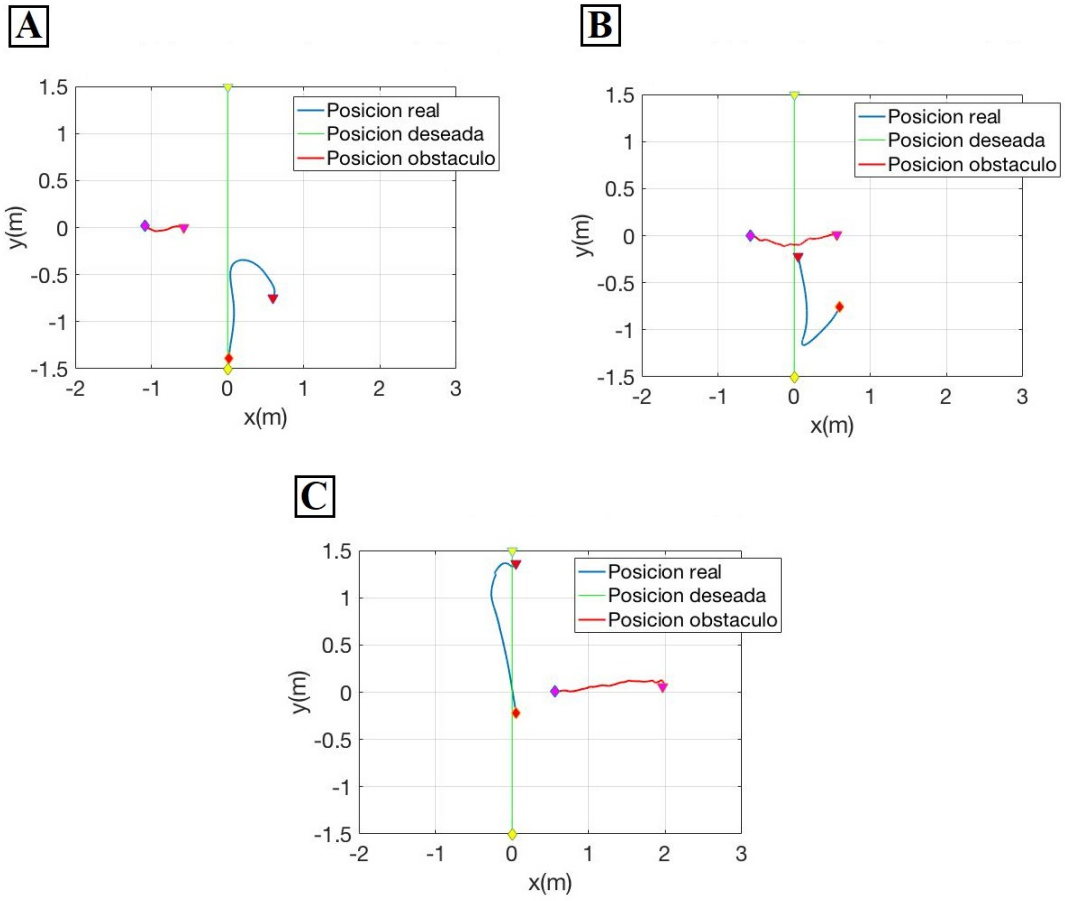


Figura 5.21: Evitación de un obstáculo móvil dividido en diferentes intervalos de tiempo con el modelo real: primer intervalo [A], segundo intervalo [B] y último intervalo [C].

Con todos estos experimentos podemos concluir que el módulo de evitación de obstáculos también presenta un buen rendimiento usando el modelo real.

CAPÍTULO 6

CONCLUSIONES Y TRABAJO FUTURO

En este apartado se agrupan las principales conclusiones a las que se han alcanzado, comprobando si se han conseguido lograr los objetivos marcados al inicio del documento. Además, se citan algunos de los trabajos futuros posibles que se podrían desarrollar para continuar el presente proyecto.

6.1 Conclusiones

Para comenzar con esta sección se analizará si se han conseguidos los objetivos marcados en la sección 1.2. Los objetivos generales eran desarrollar un controlador de posición basado en un sistema de captura de movimiento y combinarlo con un algoritmo de evitación de obstáculos.

Para alcanzar estos objetivos generales se han seguido un conjunto de objetivos específicos. Como se ha visto en los capítulos anteriores, se ha hecho un estudio del sistema operativo para aplicaciones de robótica ROS para conseguir el dominio necesario para la realización del presente proyecto. Una vez obtenido este conocimiento, el siguiente paso ha sido la familiarización con el sistema de captura de movimiento: cómo funciona y qué permite hacer. A continuación, se ha estudiado en profundidad como funciona el filtro de *Kalman*, el controlador de posición PID, y la evitación de obstáculos basado en campos de potencial. Una vez adquiridos estos conocimientos se ha procedido a la implementación de los módulos correspondientes, así como al desarrollo del módulo de navegación basado en puntos de paso.

Tras realizar la implementación de todos estos módulos se ha procedido a la configuración de los parámetros necesarios y a la realización de un conjunto de experimentos sobre el simulador.

Una vez comprobado que los resultados obtenidos de los experimentos eran los adecuados y no se habían producido ningún tipo de problema se ha proseguido con los experimentos sobre el modelo real. Los resultados obtenidos en el simulador y en el modelo real se asemejan y cumplen con los objetivos marcados al inicio del proyecto.

Como conclusión final, la realización del proyecto ha sido la esperada. Se han conseguido los objetivos planteados al inicio, tanto sobre el simulador como sobre el modelo real. Como resultado del proyecto se ha obtenido un robot aéreo capaz de alcanzar diferentes puntos del espacio y evitar los obstáculos que se encuentran en su trayectoria de manera autónoma mediante el sistema de captura de movimiento *OptiTrack*.

6.2 Trabajo Futuro

Para el presente proyecto se propone el siguiente trabajo futuro:

- ***Filtro de Kalman Extendido***: Sustitución del módulo de estimación de estado actual (filtro de *Kalman*) por el *Filtro de Kalman Extendido*. Este filtro nos permitiría introducir las aceleraciones lineales proporcionados por la IMU de tal manera que se podrían obtener estimaciones de posición más precisas en caso de una ocultación.

- ***Limitación de posición en el entorno de vuelo***: Las pruebas se han realizado en el laboratorio robótica aérea de la Universidad de las Islas Baleares. Este espacio es un entorno cerrado y limitado para el dron, y si se produce algún tipo de problema éste podría colisionar. No obstante, mediante el sistema de captura se pueden establecer unas paredes virtuales que impidan al vehículo salir de la zona de operación, y así evitar colisionar con las paredes. Ello supondría añadir una capa de control de alto nivel.

APÉNDICE A

CÓDIGO DEL MÓDULO DE ESTIMACIÓN DEL ESTADO

A.1 Código del módulo de estimación del estado (.h)

```
1  #ifndef KALMAN_FILTER_H_
2  #define KALMAN_FILTER_H_
3
4  #include <ros/ros.h>
5  #include "std_msgs/String.h"
6  #include <opencv2/opencv.hpp>
7  #include <opencv2/video.hpp>
8  #include "geometry_msgs/PoseStamped.h"
9  #include <geometry_msgs/TwistWithCovariance.h>
10 #include "geometry_msgs/PoseWithCovarianceStamped.h"
11 #include <math.h>
12 #include <boost/thread/mutex.hpp>
13 #include "tf/tf.h"
14 #include "tf/transform_broadcaster.h"
15 #include <dynamic_reconfigure/server.h>
16 #include <kalman_filter/ReconfigureConfig.h>
17
18
19 using namespace cv;
20
21 namespace tfg_kalman {
22
23   class Kalman_Filter{
24
25     public:
26
27       Kalman_Filter(const ros::NodeHandle nh);
28       virtual ~Kalman_Filter();
29       void configure();
30
```

```
31      private:
32
33        ros::NodeHandle nodeHandle;
34
35        //Dynamic reconfigure server
36
37        dynamic_reconfigure::Server<kalman_filter::ReconfigureConfig> dr_srv_;
38
39        //Params;
40
41        double ax, ay, az, ao, omx, omy, omz, omo, timeclb;
42        bool activate_gazebo;
43
44        //Variables
45
46        ros::Time tiempo, tiempoclb;
47        double tiempoSec, tiempo_anterior, period, tiempoSecclb;
48        double dt, yaw, before_yaw;
49        bool start, startimerclb, timerwarning, predictimer, stop_publish_pose;
50
51        geometry_msgs::PoseStamped last_pose_msg;
52
53        //Subscribers and publishers
54
55        ros::Subscriber pose_sub_;
56        ros::Publisher pose_publ_, twistcov_publ_, posecov_publ_;
57
58        //Service clients
59
60        //Timers
61
62        ros::Timer timer;
63
64        //Mutex
65
66        boost::mutex mutex_pos;
67
68        void poseClb(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
69        void timerClb(const ros::TimerEvent& event);
70        void matrix_refresh();
71        void inicial_state();
72        void refresh_time();
73        void kalmanPredict();
74        void kalmanCorrect();
75        //true = publicar valores de prediccion / false = publicar valores de
      correción
76        void publisher_pose_twist(bool predictimer);
77        void callback(kalman_filter::ReconfigureConfig &config, uint32_t level);
78    };
79
80  }
81
82  #endif /* KALMAN_FILTER_ */
```

A.2 Código del módulo de estimación del estado (.cpp)

```
#include "kalman_filter.h"

using namespace cv;

namespace tfg_kalman {

  KalmanFilter KF (8, 4, 0, CV_32F);
  Mat measurement (4, 1, CV_32F);

   Kalman_Filter::Kalman_Filter(const ros::NodeHandle nh):

  nodeHandle(nh)
  {
  configure();
  }

  Kalman_Filter::~Kalman_Filter() {
  }

  void Kalman_Filter::configure(){

    //Set up a dynamic reconfigure server

    dynamic_reconfigure::Server<kalman_filter::ReconfigureConfig>::CallbackType f
    ;
    f = boost::bind(&Kalman_Filter::callback, this, _1, _2);
    dr_srv_.setCallback(f);

    //Init

    start = true;
    startimerclb = false;
    timerwarning = true;
    predictimer = false;
    period = 1/120;
    stop_publish_pose = false;

    //Parameters

    nodeHandle.param("ax", ax, 0.2);
    nodeHandle.param("ay", ay, 0.2);
    nodeHandle.param("az", az, 0.2);
    nodeHandle.param("ao", ao, 0.1);
    nodeHandle.param("omx", omx, 0.01);
    nodeHandle.param("omy", omy, 0.01);
    nodeHandle.param("omz", omz, 0.01);
    nodeHandle.param("omo", omo, 0.01);
    nodeHandle.param("timeclb", timeclb, 0.03);
    nodeHandle.param("activate_gazebo", activate_gazebo, false);
```

```
55      //Advertising Topics
56
57      pose_publ_ = nodeHandle.advertise<geometry_msgs::PoseStamped>("/pose_filter",
          1);
58      posecov_publ_ = nodeHandle.advertise<geometry_msgs::PoseWithCovarianceStamped
        >("/posecov_filter", 1);
59      twistcov_publ_ = nodeHandle.advertise<geometry_msgs::TwistWithCovariance>("
        twistcov_filter", 1);
60
61      //Subscribing Topics
62
63      pose_sub_ = nodeHandle.subscribe("/pose", 1, &Kalman_Filter::poseClb, this);
64
65      //Timers
66
67      timer = nodeHandle.createTimer(ros::Duration(period),&Kalman_Filter::timerClb
        , this);
68    }
69
70    void Kalman_Filter::callback(kalman_filter::ReconfigureConfig &config, uint32_t
        level) {
71
72      //Set variables to new values
73
74      ax = config.ax;
75      ay = config.ay;
76      az = config.az;
77      ao = config.ao;
78      omx = config.omx;
79      omy = config.omy;
80      omz = config.omz;
81      omo = config.omo;
82      stop_publish_pose = config.stop_publish_pose;
83    }
84
85    void Kalman_Filter::timerClb(const ros::TimerEvent& event){
86
87      if(startimerclb == true){
88        tiempoclb = ros::Time::now();
89        tiempoSecclb = tiempoclb.toSec();
90        //segunda condición el sistema al arrancar tarda mas tiempo que timeclb
91        if(tiempoSecclb-tiempo_anterior > timeclb && tiempoSecclb-tiempo_anterior
        < 200){
92
93          if(timerwarning == true){
94            ROS_WARN("TIMERCLB is operating");
95            timerwarning = false;
96          }
97
98        Kalman_Filter::refresh_time();
99        Kalman_Filter::matrix_refresh();
100       Kalman_Filter::kalmanPredict();
101       predictimer = true;
102       Kalman_Filter::publisher_pose_twist(predictimer);
103       }
104     }
105   }
```

```
106  void Kalman_Filter::poseClb(const geometry_msgs::PoseStamped::ConstPtr&
       pose_msg){
107
108      mutex_pos.lock();
109      if(stop_publish_pose == false){
110        Kalman_Filter::refresh_time();
111        last_pose_msg = *pose_msg;
112
113       //Kalman filter optitrack
114
115       if(activate_gazebo == false){
116
117          double z = last_pose_msg.pose.position.y;
118          last_pose_msg.pose.position.y = -last_pose_msg.pose.position.z;
119          last_pose_msg.pose.position.z = z;
120          tf::Quaternion bt_q;
121          tf::quaternionMsgToTF(pose_msg->pose.orientation, bt_q);
122          double useless_pitch, useless_roll;
123          tf::Matrix3x3(bt_q).getRPY( useless_roll, yaw, useless_pitch);
124
125          //Algoritmo para cambio de angulo yaw, para que no haya 2 angulos
       iguales intervalo entre pi y -pi
126
127          if(abs(useless_roll) > M_PI/2){
128
129            if(yaw > 0){
130              yaw = M_PI - yaw;
131            }
132            if(yaw < 0){
133              yaw = -M_PI - yaw;
134            }
135          }
136
137
138       //Kalman filter gazebo
139
140       }else{
141          yaw = tf::getYaw(last_pose_msg.pose.orientation);
142       }
143
144      ROS_DEBUG("Position of sensor: %2.6f, %2.6f, %2.6f, %2.6f", pose_msg->pose
       .position.x,pose_msg->pose.position.y,pose_msg->pose.position.z, yaw);
145
146       if(start == true){
147          Kalman_Filter::inicial_state();
148          start = false;
149       }
150
151       Kalman_Filter::matrix_refresh();
152       Kalman_Filter::kalmanPredict();
153       Kalman_Filter::kalmanCorrect();
154       predictimer = false;
155       Kalman_Filter::publisher_pose_twist(predictimer);
156
157
158
159
```

```
160         if(timerwarning == false){
161
162             ROS_WARN("TIMERCLB stop");
163             timerwarning = true;
164
165         }
166
167        }else{
168
169           //reproducir ocultación
170
171           startimerclb = true;
172        }
173
174     mutex_pos.unlock();
175
176   }
177
178   void Kalman_Filter::matrix_refresh(){
179
180     //MATRICES DE PREDICCIÓN
181
182     //Ft
183
184     setIdentity(KF.transitionMatrix);
185     KF.transitionMatrix.at<float>(0,4) = dt;
186     KF.transitionMatrix.at<float>(1,5) = dt;
187     KF.transitionMatrix.at<float>(2,6) = dt;
188     KF.transitionMatrix.at<float>(3,7) = dt;
189
190     //Qt
191
192     setIdentity(KF.processNoiseCov, Scalar(0));
193     KF.processNoiseCov.at<float>(0,0)=(0.25)*(pow(ax,2))*(pow(dt,4));
194     KF.processNoiseCov.at<float>(4,0)=(0.5)*(pow(ax,2))*(pow(dt,3));
195     KF.processNoiseCov.at<float>(0,4)=(0.5)*(pow(ax,2))*(pow(dt,3));
196
197     KF.processNoiseCov.at<float>(1,1)=(0.25)*(pow(ay,2))*(pow(dt,4));
198     KF.processNoiseCov.at<float>(1,5)=(0.5)*(pow(ay,2))*(pow(dt,3));
199     KF.processNoiseCov.at<float>(5,1)=(0.5)*(pow(ay,2))*(pow(dt,3));
200
201     KF.processNoiseCov.at<float>(2,2)=(0.25)*(pow(az,2))*(pow(dt,4));
202     KF.processNoiseCov.at<float>(2,6)=(0.5)*(pow(az,2))*(pow(dt,3));
203     KF.processNoiseCov.at<float>(6,2)=(0.5)*(pow(az,2))*(pow(dt,3));
204
205     KF.processNoiseCov.at<float>(3,3)=(0.25)*(pow(ao,2))*(pow(dt,4));
206     KF.processNoiseCov.at<float>(3,7)=(0.5)*(pow(ao,2))*(pow(dt,3));
207     KF.processNoiseCov.at<float>(7,3)=(0.5)*(pow(ao,2))*(pow(dt,3));
208
209     KF.processNoiseCov.at<float>(4,4)=(pow(ax,2))*(pow(dt,2));
210     KF.processNoiseCov.at<float>(5,5)=(pow(ay,2))*(pow(dt,2));
211     KF.processNoiseCov.at<float>(6,6)=(pow(az,2))*(pow(dt,2));
212     KF.processNoiseCov.at<float>(7,7)=(pow(ao,2))*(pow(dt,2));
213
214
215
216
```

```
217     //MATRICES DE CORRECIÓN
218
219     //Rt
220
221     KF.measurementNoiseCov.at<float>(0,0)=(pow(omx,2));
222     KF.measurementNoiseCov.at<float>(1,1)=(pow(omy,2));
223     KF.measurementNoiseCov.at<float>(2,2)=(pow(omz,2));
224     KF.measurementNoiseCov.at<float>(3,3)=(pow(omo,2));
225
226     //Ht
227
228     KF.measurementMatrix.at<float>(0,0)=1;
229     KF.measurementMatrix.at<float>(1,1)=1;
230     KF.measurementMatrix.at<float>(2,2)=1;
231     KF.measurementMatrix.at<float>(3,3)=1;
232   }
233
234   void Kalman_Filter::inicial_state(){
235
236     //Xt-1
237
238     KF.statePost.at<float>(0,0)=last_pose_msg.pose.position.x;
239     KF.statePost.at<float>(1,0)=last_pose_msg.pose.position.y;
240     KF.statePost.at<float>(2,0)=last_pose_msg.pose.position.z;
241     KF.statePost.at<float>(3,0)= yaw;
242     KF.statePost.at<float>(4,0)= 0;
243     KF.statePost.at<float>(5,0)= 0;
244     KF.statePost.at<float>(6,0)= 0;
245     KF.statePost.at<float>(6,0)= 0;
246
247     dt = 0;
248
249   }
250
251   void Kalman_Filter::refresh_time(){
252
253     tiempo = ros::Time::now();
254     tiempoSec = tiempo.toSec();
255     dt = tiempoSec - tiempo_anterior;
256     startimerclb = true;
257     tiempo_anterior = tiempoSec;
258
259   }
260
261   void Kalman_Filter::kalmanPredict(){
262
263     KF.predict();
264
265   }
266
267   void Kalman_Filter::kalmanCorrect(){
268
269     //z(k)
270
271     measurement.at<float>(0,0)=last_pose_msg.pose.position.x;
272     measurement.at<float>(1,0)=last_pose_msg.pose.position.y;
273     measurement.at<float>(2,0)=last_pose_msg.pose.position.z;
```

```
274     measurement.at<float>(3,0)= yaw;
275
276     //problema entre -pi y pi velocidad muy alta cuando se realiza la corección
277
278     if(before_yaw > (M_PI/2) && yaw < -(M_PI/2)){
279       measurement.at<float>(3,0) = yaw + 2*M_PI;
280     }
281     if(yaw > (M_PI/2) && before_yaw < (-M_PI/2)){
282       measurement.at<float>(3,0) = yaw - 2*M_PI;
283     }
284
285     KF.correct(measurement);
286
287     //devolver el angulo a valores entre -pi y pi
288
289     if(KF.statePost.at<float>(3,0) > M_PI){
290       KF.statePost.at<float>(3,0)  = KF.statePost.at<float>(3,0) - 2*M_PI;
291     }
292     if(KF.statePost.at<float>(3,0) < -M_PI){
293       KF.statePost.at<float>(3,0)  = KF.statePost.at<float>(3,0) + 2*M_PI;
294     }
295
296   }
297
298   void Kalman_Filter::publisher_pose_twist(bool predictimer){
299
300     //publicar valores de predicción
301
302     if(predictimer == true){
303
304       //para valores de yaw diferentes entre pi y -pi
305
306       if(KF.statePre.at<float>(3,0) > M_PI){
307         KF.statePre.at<float>(3,0) = KF.statePre.at<float>(3,0) - 2*M_PI;
308       }
309       if(KF.statePre.at<float>(3,0) < -M_PI){
310         KF.statePre.at<float>(3,0) = KF.statePre.at<float>(3,0) + 2*M_PI;
311       }
312
313       //poseStamped
314
315       geometry_msgs::PoseStamped medition_msg;
316       medition_msg.header.frame_id = "/pose_filter";
317       medition_msg.header.stamp = tiempo;
318       medition_msg.pose.position.x = KF.statePre.at<float>(0,0);
319       medition_msg.pose.position.y = KF.statePre.at<float>(1,0);
320       medition_msg.pose.position.z = KF.statePre.at<float>(2,0);
321       medition_msg.pose.orientation = tf::createQuaternionMsgFromYaw(KF.statePre.
      at<float>(3,0));
322
323       pose_publ_.publish(medition_msg);
324       ROS_DEBUG("Position_filter Predict x y z o: %2.6f, %2.6f, %2.6f, %2.6f",
      medition_msg.pose.position.x, medition_msg.pose.position.y, medition_msg.pose
      .position.z, KF.statePre.at<float>(3,0));
325
326
327
```

```
328         //poseStampedWithCovariance
329
330         geometry_msgs::PoseWithCovarianceStamped meditioncov_msg;
331         meditioncov_msg.header.frame_id = "/posecov_filter";
332         meditioncov_msg.header.stamp = tiempo;
333         meditioncov_msg.pose.pose.position.x = KF.statePre.at<float>(0,0);
334         meditioncov_msg.pose.pose.position.y = KF.statePre.at<float>(1,0);
335         meditioncov_msg.pose.pose.position.z =KF.statePre.at<float>(2,0);
336         meditioncov_msg.pose.pose.orientation = tf::createQuaternionMsgFromYaw(KF.
        statePre.at<float>(3,0));
337         meditioncov_msg.pose.covariance[0] = KF.errorCovPre.at<float>(0,0);
338         meditioncov_msg.pose.covariance[1] = KF.errorCovPre.at<float>(1,1);
339         meditioncov_msg.pose.covariance[2] = KF.errorCovPre.at<float>(2,2);
340         meditioncov_msg.pose.covariance[3] = KF.errorCovPre.at<float>(3,3);
341
342         posecov_publ_.publish(meditioncov_msg);
343         ROS_DEBUG("Positioncov_filter x y z o : %2.9f, %2.9f ,%2.9f, %2.9f",
        meditioncov_msg.pose.covariance[0], meditioncov_msg.pose.covariance[1],
        meditioncov_msg.pose.covariance[2], meditioncov_msg.pose.covariance[3]);
344
345         //TwistWithCovariance
346
347         geometry_msgs::TwistWithCovariance speedcov_msg;
348         speedcov_msg.twist.linear.x = KF.statePre.at<float>(4,0);
349         speedcov_msg.twist.linear.y = KF.statePre.at<float>(5,0);
350         speedcov_msg.twist.linear.z = KF.statePre.at<float>(6,0);
351         speedcov_msg.twist.angular.z = KF.statePre.at<float>(7,0);
352         speedcov_msg.covariance[0] = KF.errorCovPre.at<float>(4,4);
353         speedcov_msg.covariance[1] = KF.errorCovPre.at<float>(5,5);
354         speedcov_msg.covariance[2] = KF.errorCovPre.at<float>(6,6);
355         speedcov_msg.covariance[3] = KF.errorCovPre.at<float>(7,7);
356
357         twistcov_publ_.publish(speedcov_msg);
358
359         ROS_DEBUG("Twist_filter vx vy vz wz : %2.6f, %2.6f, %2.6f, %2.6f",
        speedcov_msg.twist.linear.x, speedcov_msg.twist.linear.y, speedcov_msg.twist.
        linear.z, speedcov_msg.twist.angular.z);
360         ROS_DEBUG("Twistcov_filter vx vy vz wz: %2.9f, %2.9f ,%2.9f, %2.9f",
        speedcov_msg.covariance[0], speedcov_msg.covariance[1], speedcov_msg.
        covariance[2], speedcov_msg.covariance[3]);
361
362     //publicar valores de correción
363
364     }else{
365
366         //poseStamped
367
368         geometry_msgs::PoseStamped medition_msg;
369         medition_msg.header.frame_id = "/pose_filter";
370         medition_msg.header.stamp = tiempo;
371         medition_msg.pose.position.x = KF.statePost.at<float>(0,0);
372         medition_msg.pose.position.y = KF.statePost.at<float>(1,0);
373         medition_msg.pose.position.z =KF.statePost.at<float>(2,0);
374         medition_msg.pose.orientation = tf::createQuaternionMsgFromYaw(KF.statePost
        .at<float>(3,0));
375
376         pose_publ_.publish(medition_msg);
```

```
377       ROS_DEBUG("Position_filter  x y z o: %2.6f, %2.6f, %2.6f, %2.6f",
      medition_msg.pose.position.x, medition_msg.pose.position.y, medition_msg.pose
      .position.z, KF.statePost.at<float>(3,0));
378
379       //poseStampedWithCovariance
380
381       geometry_msgs::PoseWithCovarianceStamped meditioncov_msg;
382       meditioncov_msg.header.frame_id = "/posecov_filter";
383       meditioncov_msg.header.stamp = tiempo;
384       meditioncov_msg.pose.pose.position.x = KF.statePost.at<float>(0,0);
385       meditioncov_msg.pose.pose.position.y = KF.statePost.at<float>(1,0);
386       meditioncov_msg.pose.pose.position.z =KF.statePost.at<float>(2,0);
387       meditioncov_msg.pose.pose.orientation = tf::createQuaternionMsgFromYaw(KF.
      statePost.at<float>(3,0));
388       meditioncov_msg.pose.covariance[0] = KF.errorCovPost.at<float>(0,0);
389       meditioncov_msg.pose.covariance[1] = KF.errorCovPost.at<float>(1,1);
390       meditioncov_msg.pose.covariance[2] = KF.errorCovPost.at<float>(2,2);
391       meditioncov_msg.pose.covariance[3] = KF.errorCovPost.at<float>(3,3);
392
393       posecov_publ_.publish(meditioncov_msg);
394       ROS_DEBUG("Positioncov_filter x y z o : %2.9f, %2.9f ,%2.9f, %2.9f",
      meditioncov_msg.pose.covariance[0], meditioncov_msg.pose.covariance[1],
      meditioncov_msg.pose.covariance[2], meditioncov_msg.pose.covariance[3]);
395
396       //TwistWithCovariance
397
398       geometry_msgs::TwistWithCovariance speedcov_msg;
399       speedcov_msg.twist.linear.x = KF.statePost.at<float>(4,0);
400       speedcov_msg.twist.linear.y = KF.statePost.at<float>(5,0);
401       speedcov_msg.twist.linear.z = KF.statePost.at<float>(6,0);
402       speedcov_msg.twist.angular.z = KF.statePost.at<float>(7,0);
403       speedcov_msg.covariance[0] = KF.errorCovPost.at<float>(4,4);
404       speedcov_msg.covariance[1] = KF.errorCovPost.at<float>(5,5);
405       speedcov_msg.covariance[2] = KF.errorCovPost.at<float>(6,6);
406       speedcov_msg.covariance[3] = KF.errorCovPost.at<float>(7,7);
407
408       twistcov_publ_.publish(speedcov_msg);
409
410       ROS_DEBUG("Twist_filter vx vy vz wz : %2.6f, %2.6f, %2.6f, %2.6f",
      speedcov_msg.twist.linear.x, speedcov_msg.twist.linear.y, speedcov_msg.twist.
      linear.z, speedcov_msg.twist.angular.z);
411       ROS_DEBUG("Twistcov_filter vx vy vz wz: %2.9f, %2.9f ,%2.9f, %2.9f",
      speedcov_msg.covariance[0], speedcov_msg.covariance[1], speedcov_msg.
      covariance[2], speedcov_msg.covariance[3]);
412
413       before_yaw = KF.statePost.at<float>(3,0);
414     }
415
416   }
417
418 }
```

APÉNDICE B

CÓDIGO DEL CONTROLADOR DE POSICIÓN

B.1 Código del controlador de posición (.h)

```
1  #ifndef PID_H_
2  #define PID_H_
3
4  #include <ros/ros.h>
5  #include "std_msgs/String.h"
6  #include <opencv2/opencv.hpp>
7  #include <opencv2/video.hpp>
8  #include "geometry_msgs/PoseStamped.h"
9  #include <geometry_msgs/Twist.h>
10 #include <geometry_msgs/TwistWithCovariance.h>
11 #include "geometry_msgs/PoseWithCovarianceStamped.h"
12 #include <math.h>
13 #include <boost/thread/mutex.hpp>
14 #include "tf/tf.h"
15 #include "tf/transform_broadcaster.h"
16 #include <dynamic_reconfigure/server.h>
17 #include <pid/PidReconConfig.h>
18 using namespace std;
19
20 using namespace cv;
21
22 namespace tfg_PID {
23
24   class PID{
25
26     public:
27
28       PID(const ros::NodeHandle nh);
29       virtual ~PID();
30       void configure();
31
32
```

```
33      private:
34
35        ros::NodeHandle nodeHandle;
36
37        //Dynamic reconfigure server
38
39        dynamic_reconfigure::Server<pid::PidReconConfig> dr_srv;
40        double x, y, z, yaw;
41
42        //Params;
43
44        double yaw_kp, yaw_kd, yaw_ki, x_kp, x_kd,
45        x_ki, y_kp, y_kd, y_ki, z_kp, z_kd, z_ki;
46        double x_max_vel, y_max_vel, z_max_vel,
47        yaw_max_vel, x_max_Iout, y_max_Iout, z_max_Iout, yaw_max_Iout;
48
49        //Variables
50
51        ros::Time tiempo_anterior;
52
53        double last_yaw, yaw_desired, x_velout, y_velout,
54        z_velout, yaw_velout, dt;
55        double x_integral, y_integral, z_integral,
56        yaw_integral, tiempo_anteriorSec;
57        double x_error_before, y_error_before, z_error_before, yaw_error_before;
58        bool first_run, stop;
59
60        geometry_msgs::PoseStamped last_pose_msg, last_pose_desired_msg;
61        geometry_msgs::TwistWithCovariance last_twist_msg;
62
63        // Subscribers and publishers
64
65        ros::Subscriber pose_sub_, twistcov_sub_, posedesired_sub_;
66        ros::Publisher twist_publ_;
67
68        void poseclb(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
69        void twistcovclb(const geometry_msgs::TwistWithCovariance::ConstPtr&
      twistcov_msg);
70        void posedesired (const geometry_msgs::PoseStamped::ConstPtr&
      pose_desired_msg);
71        void calculate();
72        void publisher_twist_out();
73        void callback(pid::PidReconConfig &config, uint32_t level);
74    };
75
76  }
77
78  #endif /* PID_ */
```

B.2 Código del controlador de posición (.cpp)

```
#include "pid.h"

using namespace cv;

namespace tfg_PID{

    PID::PID(const ros::NodeHandle nh):
  nodeHandle(nh)
  {
  configure();
  }

  PID::~PID() {
  }

  void PID::configure(){

    //Set up a dynamic reconfigure server

    dynamic_reconfigure::Server<pid::PidReconConfig>::CallbackType f;
    f = boost::bind(&PID::callback, this, _1, _2);
    dr_srv.setCallback(f);

    //Init

    first_run = true;

    x_integral = 0;
    y_integral = 0;
    z_integral = 0;
    yaw_integral = 0;

    x_error_before = 0;
    y_error_before = 0;
    z_error_before = 0;
    yaw_error_before = 0;


    //Parameters

    nodeHandle.param("x_kp", x_kp, 5.0);
    nodeHandle.param("y_kp", y_kp, 5.0);
    nodeHandle.param("z_kp", z_kp, 5.0);
    nodeHandle.param("yaw_kp", yaw_kp, 5.0);
    nodeHandle.param("x_ki", x_ki, 5.0);
    nodeHandle.param("y_ki", y_ki, 5.0);
    nodeHandle.param("z_ki", z_ki, 5.0);
    nodeHandle.param("yaw_ki", yaw_ki, 5.0);
    nodeHandle.param("x_kd", x_kd, 5.0);
    nodeHandle.param("y_kd", y_kd, 5.0);
    nodeHandle.param("z_kd", z_kd, 5.0);
    nodeHandle.param("yaw_kd", yaw_kd, 5.0);
    nodeHandle.param("x_max_vel", x_max_vel, 5.0);
    nodeHandle.param("y_max_vel", y_max_vel, 5.0);
    nodeHandle.param("z_max_vel", z_max_vel, 5.0);
```

```
56      nodeHandle.param("yaw_max_vel", yaw_max_vel, 5.0);
57      nodeHandle.param("x_max_Iout", x_max_Iout, 5.0);
58      nodeHandle.param("y_max_Iout", y_max_Iout, 5.0);
59      nodeHandle.param("z_max_Iout", z_max_Iout, 5.0);
60      nodeHandle.param("yaw_max_Iout", yaw_max_Iout, 5.0);
61      nodeHandle.param("x_position", x, 0.0);
62      nodeHandle.param("y_position", y, 0.0);
63      nodeHandle.param("z_position", z, 0.0);
64      nodeHandle.param("yaw_position", yaw, 0.0);
65
66
67      //Advertising Topics
68
69      twist_publ_ = nodeHandle.advertise<geometry_msgs::Twist>("/twist_desired/out"
        , 1);
70
71      //Subscribing Topics
72
73      pose_sub_ = nodeHandle.subscribe("/kalman_filter/Ardrone/pose", 1, &PID::
        poseclb, this);
74      twistcov_sub_ = nodeHandle.subscribe("/kalman_filter/Ardrone/twistcov", 1, &
        PID::twistcovclb, this);
75      posedesired_sub_ = nodeHandle.subscribe("/waypointnavigator/Ardrone/
        pose_desired", 1, &PID::posedesired, this);
76
77      //Init position
78
79      last_pose_desired_msg.pose.position.x = x;
80      last_pose_desired_msg.pose.position.y = y;
81      last_pose_desired_msg.pose.position.z = z;
82      yaw_desired = yaw;
83    }
84
85  void PID::callback(pid::PidReconConfig &config, uint32_t level){
86
87       //Set variables to new values
88
89       x_ki = config.x_ki;
90       x_kp = config.x_kp;
91       x_kd = config.x_kd;
92       y_ki = config.y_ki;
93       y_kp = config.y_kp;
94       y_kd = config.y_kd;
95       z_ki = config.z_ki;
96       z_kp = config.z_kp;
97       z_kd = config.z_kd;
98       yaw_ki = config.yaw_ki;
99       yaw_kp = config.yaw_kp;
100      yaw_kd = config.yaw_kd;
101      last_pose_desired_msg.pose.position.x = config.x;
102      last_pose_desired_msg.pose.position.y = config.y;
103      last_pose_desired_msg.pose.position.z = config.z;
104      yaw_desired = config.yaw;
105      ROS_INFO("ERROR x : %2.6f,", x_kp);
106
107   }
108
```

```
109  void PID::poseclb (const geometry_msgs::PoseStamped::ConstPtr& pose_msg){
110
111     last_pose_msg = *pose_msg;
112     last_yaw = tf::getYaw(pose_msg->pose.orientation);
113
114     if(first_run == false){
115
116      PID::calculate();
117      PID::publisher_twist_out();
118
119     }else{
120
121      tiempo_anterior = ros::Time::now();
122      tiempo_anteriorSec = tiempo_anterior.toSec();
123      first_run = false;
124     }
125  }
126
127  void PID::twistcovclb (const geometry_msgs::TwistWithCovariance::ConstPtr&
       twistcov_msg){
128
129     last_twist_msg = *twistcov_msg;
130
131  }
132
133  void PID::posedesired (const geometry_msgs::PoseStamped::ConstPtr&
       pose_desired_msg){
134
135     last_pose_desired_msg = *pose_desired_msg;
136     yaw_desired = tf::getYaw(pose_desired_msg->pose.orientation);
137
138  }
139
140  void PID::calculate (){
141
142     //Actualizar dt
143
144     dt = ros::Time::now().toSec() - tiempo_anteriorSec;
145
146     //Calcular error
147
148     double x_error = last_pose_desired_msg.pose.position.x - last_pose_msg.pose.
       position.x;
149     double y_error = last_pose_desired_msg.pose.position.y - last_pose_msg.pose.
       position.y;
150     double z_error = last_pose_desired_msg.pose.position.z - last_pose_msg.pose.
       position.z;
151     double yaw_error = yaw_desired - last_yaw;
152
153     if(yaw_error > M_PI){
154      yaw_error = -(2*M_PI - yaw_error);
155     }
156     if(yaw_error < -M_PI){
157      yaw_error = -(-2*M_PI - yaw_error);
158     }
159
160
```

```
161      //Término proporcional
162
163      double x_Pout = x_kp * x_error;
164      double y_Pout = y_kp * y_error;
165      double z_Pout = z_kp * z_error;
166      double yaw_Pout = yaw_kp * yaw_error;
167
168      //Termino derivativo
169
170      double x_Dout = x_kd * last_twist_msg.twist.linear.x;
171      double y_Dout = y_kd * last_twist_msg.twist.linear.y;
172      double z_Dout = z_kd * last_twist_msg.twist.linear.z;
173      double yaw_Dout = yaw_kd * last_twist_msg.twist.angular.z;
174
175      //Término integral
176
177      if(x_error_before == 0 || y_error_before == 0 || z_error_before == 0 ||
       yaw_error_before == 0){
178
179       x_error_before = x_error;
180       y_error_before = y_error;
181       z_error_before = z_error;
182       yaw_error_before = yaw_error;
183      }
184
185      x_integral += ((x_error + x_error_before)/2) * dt;
186      y_integral += ((y_error + y_error_before)/2) * dt;
187      z_integral += ((z_error + z_error_before)/2) * dt;
188      yaw_integral += ((yaw_error + yaw_error_before)/2) * dt;
189
190      double x_Iout = x_ki * x_integral;
191      double y_Iout = y_ki * y_integral;
192      double z_Iout = z_ki * z_integral;
193      double yaw_Iout = yaw_ki * yaw_integral;
194
195      //Restringir a max/min termino integral
196
197      if(x_Iout > x_max_Iout){
198       x_Iout = x_max_Iout;
199      }
200      if(y_Iout > y_max_Iout){
201       y_Iout = y_max_Iout;
202      }
203      if(z_Iout > z_max_Iout){
204       z_Iout = z_max_Iout;
205      }
206      if(yaw_Iout > yaw_max_Iout){
207       yaw_Iout = yaw_max_Iout;
208      }
209      if(x_Iout < -x_max_Iout){
210       x_Iout = -x_max_Iout;
211      }
212      if(y_Iout < -y_max_vel){
213       y_Iout = -y_max_Iout;
214      }
215      if(z_Iout < -z_max_Iout){
216       z_Iout = -z_max_Iout;
```

```
217         }
218         if(yaw_Iout < -yaw_max_Iout){
219          yaw_Iout = -yaw_max_Iout;
220         }
221
222         //Velocidad de salida
223
224         x_velout = x_Pout - x_Dout + x_Iout;
225         y_velout = y_Pout - y_Dout + y_Iout;
226         z_velout = z_Pout - z_Dout + z_Iout;
227         yaw_velout = yaw_Pout - yaw_Dout + yaw_Iout;
228
229         //Guardar tiempo ros
230
231         tiempo_anteriorSec = ros::Time::now().toSec();
232
233         //Guardar anteriores errores
234
235         x_error_before = x_error;
236         y_error_before = y_error;
237         z_error_before = z_error;
238         yaw_error_before = yaw_error;
239     }
240
241     void PID::publisher_twist_out(){
242
243         //publicar velocidad del controlador
244
245         geometry_msgs::Twist speed_msg;
246
247         //Rotación a coordenadas robot
248
249         speed_msg.linear.x = x_velout * cos(-last_yaw) - y_velout * sin(-last_yaw);
250         speed_msg.linear.y = y_velout * cos(-last_yaw) + x_velout * sin(-last_yaw);
251         speed_msg.linear.z = z_velout;
252         speed_msg.angular.z = yaw_velout;
253
254         //Restringir a max/min velocidad del controlador
255
256         if(speed_msg.linear.x > x_max_vel){
257          speed_msg.linear.x = x_max_vel;
258         }
259         if(speed_msg.linear.y > y_max_vel){
260          speed_msg.linear.y = y_max_vel;
261         }
262         if(speed_msg.linear.z > z_max_vel){
263          speed_msg.linear.z = z_max_vel;
264         }
265         if(speed_msg.angular.z > yaw_max_vel){
266          speed_msg.angular.z = yaw_max_vel;
267         }
268         if(speed_msg.linear.x < -x_max_vel){
269          speed_msg.linear.x = -x_max_vel;
270         }
271         if(speed_msg.linear.y < -y_max_vel){
272          speed_msg.linear.y = -y_max_vel;
273         }
```

```
274         if(speed_msg.linear.z < -z_max_vel){
275          speed_msg.linear.z = -z_max_vel;
276         }
277         if(speed_msg.angular.z < -yaw_max_vel){
278          speed_msg.angular.z = -yaw_max_vel;
279         }
280
281         twist_publ_.publish(speed_msg);
282
283         ROS_DEBUG("Twist_PID vx vy vz wz : %2.6f, %2.6f, %2.6f, %2.6f", speed_msg.
            linear.x, speed_msg.linear.y, speed_msg.linear.z, speed_msg.angular.z);
284     }
285
286 }
```

APÉNDICE C

CÓDIGO DEL MÓDULO DE NAVEGACIÓN

C.1 Código del módulo de navegación (.h)

```
1  #ifndef Waypoint_Navigator_H_
2  #define Waypoint_Navigator_H_
3
4  #include <ros/ros.h>
5  #include "std_msgs/String.h"
6  #include "geometry_msgs/PoseStamped.h"
7  #include <math.h>
8  #include <boost/thread/mutex.hpp>
9  #include "tf/tf.h"
10 #include "tf/transform_broadcaster.h"
11 #include <iostream>
12 #include <fstream>
13
14 using namespace std;
15
16
17 namespace tfg_Waypoint_Navigator {
18
19   class Waypoint_Navigator{
20
21     public:
22
23       Waypoint_Navigator(const ros::NodeHandle nh);
24       virtual ~Waypoint_Navigator();
25       void configure();
26
27     private:
28
29       ros::NodeHandle nodeHandle;
30
31
32
```

```
33        //Params;
34
35        string positions;
36
37        //Variables
38
39        ros::Time tiempo_anterior;
40        double x[255], y[255], z[255], yaw[255];
41        int i, n;
42        geometry_msgs::PoseStamped pose_desired_msg;
43        bool finish;
44
45        double last_yaw, yaw_desired;
46
47        geometry_msgs::PoseStamped last_pose_msg;
48
49        // Subscribers and publishers
50
51        ros::Subscriber pose_sub_;
52        ros::Publisher pose_publ_;
53
54        void poseclb(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
55        void publisher_pose_desired();
56        void read_file();
57    };
58
59 }
60
61 #endif /* Waypoint_Navigator_ */
```

C.2 Código del módulo de navegación (.cpp)

```
1  #include "waypoint_navigator.h"
2
3  using namespace std;
4
5  namespace tfg_Waypoint_Navigator{
6
7      Waypoint_Navigator::Waypoint_Navigator(const ros::NodeHandle nh):
8    nodeHandle(nh)
9    {
10   configure();
11   }
12
13   Waypoint_Navigator::~Waypoint_Navigator() {
14   }
15
16   void Waypoint_Navigator::configure(){
17
18     //Init
19
20     i = 1;
21     n = 1;
22     finish = false;
23
24     //Parameters
25
26     nodeHandle.param<std::string>("positions", positions, "/home/marc/catkin_ws/
       src/waypoint_navigator/src/positions.txt");
27
28     //Advertising Topics
29
30     pose_publ_ = nodeHandle.advertise<geometry_msgs::PoseStamped>("/pose_desired/
       out", 1);
31
32     //Subscribing Topics
33
34     pose_sub_ = nodeHandle.subscribe("/kalman_filter/Ardrone/pose", 1, &
       Waypoint_Navigator::poseclb, this);
35
36     Waypoint_Navigator::read_file();
37     Waypoint_Navigator::publisher_pose_desired();
38
39   }
40
41
42   void Waypoint_Navigator::poseclb (const geometry_msgs::PoseStamped::ConstPtr&
       pose_msg){
43
44     if(finish == false){
45       last_pose_msg = *pose_msg;
46       last_yaw = tf::getYaw(pose_msg->pose.orientation);
47       Waypoint_Navigator::publisher_pose_desired();
48
49
50
51
```

```
52        //Área de tolelarancia
53
54        if(abs(pose_desired_msg.pose.position.x - last_pose_msg.pose.position.x) <
          0.2 && abs(pose_desired_msg.pose.position.y - last_pose_msg.pose.position.y)
          < 0.2 && abs(pose_desired_msg.pose.position.z - last_pose_msg.pose.position.z
          ) < 0.2 && abs(pose_desired_msg.pose.orientation.z - last_pose_msg.pose.
          orientation.z) < 0.15){
55          n++;
56          ROS_INFO("GOAL");
57          if(n <= i){
58            Waypoint_Navigator::publisher_pose_desired();
59          }else{
60            ROS_INFO("Expedicion finalizada");
61            finish = true;
62          }
63        }
64     }
65   }
66
67   void Waypoint_Navigator::publisher_pose_desired(){
68
69     //publicar punto objetivo
70
71     pose_desired_msg.header.frame_id = "/pose_desired/out";
72     pose_desired_msg.header.stamp = ros::Time::now();
73     pose_desired_msg.pose.position.x = x[n];
74     pose_desired_msg.pose.position.y = y[n];
75     pose_desired_msg.pose.position.z = z[n];
76     pose_desired_msg.pose.orientation = tf::createQuaternionMsgFromYaw(yaw[n]);
77
78     pose_publ_.publish(pose_desired_msg);
79     ROS_DEBUG("Position_desired Predict x y z o: %2.6f, %2.6f, %2.6f, %2.6f",
          pose_desired_msg.pose.position.x, pose_desired_msg.pose.position.y,
          pose_desired_msg.pose.position.z, yaw[n]);
80
81   }
82
83   void Waypoint_Navigator::read_file(){
84
85     ifstream fe;
86     fe.open(positions.c_str());
87     if(fe.is_open()){
88       ROS_INFO("The file is open");
89       while (! fe.eof() ){
90         fe >> x[i];
91         fe >> y[i];
92         fe >> z[i];
93         fe >> yaw[i];
94         i++;
95       }
96     }else{
97       ROS_INFO("Error of read de file");
98     }
99     i = i -2;
100  }
101 }
```

APÉNDICE D

CÓDIGO DEL MÓDULO DE EVITACIÓN DE OBSTÁCULOS

D.1 Código del módulo de evitación de obstáculos (.h)

```
#ifndef POTENTIAL_FIELD_H_
#define POTENTIAL_FIELD_H_

#include <ros/ros.h>
#include "std_msgs/String.h"
#include <opencv2/opencv.hpp>
#include <opencv2/video.hpp>
#include "geometry_msgs/PoseStamped.h"
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/TwistWithCovariance.h>
#include "geometry_msgs/PoseWithCovarianceStamped.h"
#include <math.h>
#include <boost/thread/mutex.hpp>
#include "tf/tf.h"
#include "tf/transform_broadcaster.h"
#include <dynamic_reconfigure/server.h>
#include <potential_field/Potential_fieldReconConfig.h>
using namespace std;

using namespace cv;

namespace tfg_POTENTIAL_FIELD {

  class POTENTIAL_FIELD{

    public:

      POTENTIAL_FIELD(const ros::NodeHandle nh);
      virtual ~POTENTIAL_FIELD();
      void configure();
```

```
31      private:
32
33        ros::NodeHandle nodeHandle;
34
35        //Dynamic reconfigure server
36
37        dynamic_reconfigure::Server<potential_field::Potential_fieldReconConfig>
        dr_srv;
38        double x, y, z, yaw, Vrepx, Vrepy;
39
40        //Params;
41
42        double x_max_vel, y_max_vel;
43        double d_min, d_max, pond;
44
45        //Variables
46
47        ros::Time tiempo_anterior;
48
49        double last_yaw, Vx, Vy;
50        bool sigmo1, sigmo2, sigmo3, sigmo4, sigmo5, activate_optitrak;
51        double sigmoidal1, sigmoidal2, sigmoidal3, sigmoidal4, sigmoidal5;
52        double alpha1, alpha2, alpha3, alpha4, alpha5;
53
54        geometry_msgs::PoseStamped last_pose_msg, last_pose_desired_msg,
        last_pose_obs1_msg, last_pose_obs2_msg, last_pose_obs3_msg,
        last_pose_obs4_msg, last_pose_obs5_msg;
55        geometry_msgs::Twist last_twist_msg;
56
57        // Subscribers and publishers
58
59        ros::Subscriber pose_sub_, pose_sub_obs1_, pose_sub_obs2_, pose_sub_obs3_,
        pose_sub_obs4_, pose_sub_obs5_, twist_sub_;
60        ros::Publisher twist_publ_, twist_rep_publ_;
61
62        void poseclb(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
63        void poseclbobs1(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
64        void poseclbobs2(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
65        void poseclbobs3(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
66        void poseclbobs4(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
67        void poseclbobs5(const geometry_msgs::PoseStamped::ConstPtr& pose_msg);
68        void twistclb(const geometry_msgs::Twist::ConstPtr& twistcov_msg);
69        void posedesired (const geometry_msgs::PoseStamped::ConstPtr&
        pose_desired_msg);
70        void calculate ();
71        void publisher_twist_out();
72        void publisher_twist_vector_repulsion();
73        void callback(potential_field::Potential_fieldReconConfig &config, uint32_t
         level);
74    };
75
76  }
77
78  #endif /* POTENTIAL_FIELD_ */
```

D.2 Código del módulo de evitación de obstáculos (.cpp)

```
#include "potential_field.h"

using namespace cv;

namespace tfg_POTENTIAL_FIELD{

  POTENTIAL_FIELD::POTENTIAL_FIELD(const ros::NodeHandle nh):
  nodeHandle(nh)
  {
  configure();
  }

  POTENTIAL_FIELD::~POTENTIAL_FIELD() {
  }

  void POTENTIAL_FIELD::configure(){

    //Set up a dynamic reconfigure server
        dynamic_reconfigure::Server<potential_field::Potential_fieldReconConfig
    >::CallbackType f;
    f = boost::bind(&POTENTIAL_FIELD::callback, this, _1, _2);
    dr_srv.setCallback(f);

    //Init

    sigmo1 = false;
    sigmo2 = false;
    sigmo3 = false;
    sigmo4 = false;
    sigmo5 = false;
    sigmoidal1 = 0;
    sigmoidal2 = 0;
    sigmoidal3 = 0;
    sigmoidal4 = 0;
    sigmoidal5 = 0;
    alpha1 = 0;
    alpha2 = 0;
    alpha3 = 0;
    alpha4 = 0;
    alpha5 = 0;
    activate_optitrak = false;

    //Parameters

    nodeHandle.param("d_min", d_min, 1.0);
    nodeHandle.param("d_max", d_max, 4.0);
    nodeHandle.param("pond", pond, 0.5);
    nodeHandle.param("x_max_vel", x_max_vel, 0.2);
    nodeHandle.param("y_max_vel", y_max_vel, 0.2);
    nodeHandle.param("activate_optitrak", activate_optitrak, false);

    //Advertising Topics

    twist_publ_ = nodeHandle.advertise<geometry_msgs::Twist>("/twist_potential/
    out", 1);
```

```
54      twist_rep_publ_ = nodeHandle.advertise<geometry_msgs::Twist>("/
        twist_potential_rep/out", 1);
55
56      //Subscribing Topics
57
58      pose_sub_ = nodeHandle.subscribe("/kalman_filter/Ardrone/pose", 1, &
        POTENTIAL_FIELD::poseclb, this);
59      pose_sub_obs1_ = nodeHandle.subscribe("/vrpn_client_node/Obs1/pose", 1, &
        POTENTIAL_FIELD::poseclbobs1, this);
60      pose_sub_obs2_ = nodeHandle.subscribe("/vrpn_client_node/Obs2/pose", 1, &
        POTENTIAL_FIELD::poseclbobs2, this);
61      pose_sub_obs3_ = nodeHandle.subscribe("/vrpn_client_node/Obs3/pose", 1, &
        POTENTIAL_FIELD::poseclbobs3, this);
62      pose_sub_obs4_ = nodeHandle.subscribe("/vrpn_client_node/Obs4/pose", 1, &
        POTENTIAL_FIELD::poseclbobs4, this);
63      pose_sub_obs5_ = nodeHandle.subscribe("/vrpn_client_node/Obs5/pose", 1, &
        POTENTIAL_FIELD::poseclbobs5, this);
64      twist_sub_ = nodeHandle.subscribe("/pid/Ardrone/twist", 1, &POTENTIAL_FIELD::
        twistclb, this);
65
66    }
67
68    void POTENTIAL_FIELD::callback(potential_field::Potential_fieldReconConfig &
        config, uint32_t level){
69
70      //Set variables to new values
71
72      d_min = config.d_min;
73      d_max = config.d_max;
74      pond = config.pond;
75
76    }
77
78    void POTENTIAL_FIELD::poseclb (const geometry_msgs::PoseStamped::ConstPtr&
        pose_msg){
79
80      last_pose_msg = *pose_msg;
81      last_yaw = tf::getYaw(pose_msg->pose.orientation);
82
83      POTENTIAL_FIELD::calculate();
84      POTENTIAL_FIELD::publisher_twist_out();
85      POTENTIAL_FIELD::publisher_twist_vector_repulsion();
86
87    }
88
89      //posiciones de obstáculos
90
91      void POTENTIAL_FIELD::poseclbobs1 (const geometry_msgs::PoseStamped::ConstPtr
        & pose_msg){
92
93      last_pose_obs1_msg = *pose_msg;
94      sigmo1 = true;
95
96      if(activate_optitrak == true){
97        double z = last_pose_obs1_msg.pose.position.y;
98        last_pose_obs1_msg.pose.position.y = -last_pose_obs1_msg.pose.position.z;
99        last_pose_obs1_msg.pose.position.z = z;
```

```
100     }
101
102   }
103
104   void POTENTIAL_FIELD::poseclbobs2 (const geometry_msgs::PoseStamped::ConstPtr&
        pose_msg){
105
106     last_pose_obs2_msg = *pose_msg;
107     sigmo2 = true;
108
109     if(activate_optitrak == true){
110       double z = last_pose_obs2_msg.pose.position.y;
111       last_pose_obs2_msg.pose.position.y = -last_pose_obs2_msg.pose.position.z;
112       last_pose_obs2_msg.pose.position.z = z;
113     }
114
115   }
116
117   void POTENTIAL_FIELD::poseclbobs3 (const geometry_msgs::PoseStamped::ConstPtr&
        pose_msg){
118
119     last_pose_obs3_msg = *pose_msg;
120     sigmo3 = true;
121
122     if(activate_optitrak == true){
123       double z = last_pose_obs3_msg.pose.position.y;
124       last_pose_obs3_msg.pose.position.y = -last_pose_obs3_msg.pose.position.z;
125       last_pose_obs3_msg.pose.position.z = z;
126     }
127
128   }
129
130
131   void POTENTIAL_FIELD::poseclbobs4 (const geometry_msgs::PoseStamped::ConstPtr&
        pose_msg){
132
133     last_pose_obs4_msg = *pose_msg;
134     sigmo4 = true;
135
136     if(activate_optitrak == true){
137       double z = last_pose_obs4_msg.pose.position.y;
138       last_pose_obs4_msg.pose.position.y = -last_pose_obs4_msg.pose.position.z;
139       last_pose_obs4_msg.pose.position.z = z;
140     }
141
142   }
143
144   void POTENTIAL_FIELD::poseclbobs5 (const geometry_msgs::PoseStamped::ConstPtr&
        pose_msg){
145
146     last_pose_obs5_msg = *pose_msg;
147     sigmo5 = true;
148
149     if(activate_optitrak == true){
150       double z = last_pose_obs5_msg.pose.position.y;
151       last_pose_obs5_msg.pose.position.y = -last_pose_obs5_msg.pose.position.z;
152       last_pose_obs5_msg.pose.position.z = z;
```

```
153     }
154
155   }
156
157   void POTENTIAL_FIELD::twistclb (const geometry_msgs::Twist::ConstPtr& twist_msg
        ){
158
159     last_twist_msg = *twist_msg;;
160
161   }
162
163   void POTENTIAL_FIELD::calculate (){
164
165     //Calcular distancia entre la posición del ardrone y cada uno de los obstá
        culos con la sigmoidal
166
167     if(sigmo1 == true){
168
169       double x_d_obs1 = last_pose_obs1_msg.pose.position.x - last_pose_msg.pose.
        position.x;
170       double y_d_obs1 = last_pose_obs1_msg.pose.position.y - last_pose_msg.pose.
        position.y;
171       alpha1 = atan2(-y_d_obs1, -x_d_obs1);
172
173       sigmoidal1 = (-(1 /(1 + (exp(-d_max*((sqrt(pow(x_d_obs1, 2) + pow(y_d_obs1,
         2)))-d_min))))))) + 1;
174
175       if(sqrt(pow(x_d_obs1, 2) + pow(y_d_obs1, 2)) >= 1.2){
176
177         sigmoidal1 = 0;
178       }
179     }
180
181     if(sigmo2 == true){
182
183       double x_d_obs2 = last_pose_obs2_msg.pose.position.x - last_pose_msg.pose.
        position.x;
184       double y_d_obs2 = last_pose_obs2_msg.pose.position.y - last_pose_msg.pose.
        position.y;
185       alpha2 = atan2(-y_d_obs2, -x_d_obs2);
186
187       sigmoidal2 = (-(1 /(1 + (exp(-d_max*((sqrt(pow(x_d_obs2, 2) + pow(y_d_obs2,
         2)))-d_min))))))) + 1;
188
189       if(sqrt(pow(x_d_obs2, 2) + pow(y_d_obs2, 2)) >= 1.2){
190
191         sigmoidal2 = 0;
192       }
193   }
194
195   if(sigmo3 == true){
196
197     double x_d_obs3 = last_pose_obs3_msg.pose.position.x - last_pose_msg.pose.
        position.x;
198     double y_d_obs3 = last_pose_obs3_msg.pose.position.y - last_pose_msg.pose.
        position.y;
199     alpha3 = atan2(-y_d_obs3, -x_d_obs3);
```

```
200
201     sigmoidal3 = (-(1 /(1 + (exp(-d_max*((sqrt(pow(x_d_obs3, 2) + pow(y_d_obs3,
        2)))-d_min))))))) + 1;
202
203     if(sqrt(pow(x_d_obs3, 2) + pow(y_d_obs3, 2)) >= 1.2){
204
205       sigmoidal3 = 0;
206     }
207   }
208
209   if(sigmo4 == true){
210
211     double x_d_obs4 = last_pose_obs4_msg.pose.position.x - last_pose_msg.pose.
        position.x;
212     double y_d_obs4 = last_pose_obs4_msg.pose.position.y - last_pose_msg.pose.
        position.y;
213     alpha4 = atan2(-y_d_obs4, -x_d_obs4);
214
215     sigmoidal4 = (-(1 /(1 + (exp(-d_max*((sqrt(pow(x_d_obs4, 2) + pow(y_d_obs4,
        2)))-d_min))))))) + 1;
216
217     if(sqrt(pow(x_d_obs4, 2) + pow(y_d_obs4, 2)) >= 1.2){
218
219       sigmoidal4 = 0;
220     }
221   }
222
223   if(sigmo5 == true){
224
225     double x_d_obs5 = last_pose_obs5_msg.pose.position.x - last_pose_msg.pose.
        position.x;
226     double y_d_obs5 = last_pose_obs5_msg.pose.position.y - last_pose_msg.pose.
        position.y;
227     alpha5 = atan2(-y_d_obs5, -x_d_obs5);
228
229     sigmoidal5 = (-(1 /(1 + (exp(-d_max*((sqrt(pow(x_d_obs5, 2) + pow(y_d_obs5,
        2)))-d_min))))))) + 1;
230
231     if(sqrt(pow(x_d_obs5, 2) + pow(y_d_obs5, 2)) >= 1.2){
232
233       sigmoidal5 = 0;
234     }
235   }
236
237   //Calcular suma de sigmoidal
238
239   Vrepx = sigmoidal1 * cos(alpha1) +  sigmoidal2 * cos(alpha2) + sigmoidal3 * cos
        (alpha3) + sigmoidal4 * cos(alpha4) + sigmoidal5 * cos(alpha5);
240   Vrepy = sigmoidal1 * sin(alpha1) +  sigmoidal2 * sin(alpha2) + sigmoidal3 * sin
        (alpha3) + sigmoidal4 * sin(alpha4) + sigmoidal5 * sin(alpha5);;
241
242   Vrepx = Vrepx * x_max_vel;
243   Vrepy = Vrepy * y_max_vel;
244
245
246
247   //Coordenadas robot
```

```
248
249   Vrepx = Vrepx * cos(-last_yaw) - Vrepy * sin(-last_yaw);
250   Vrepy = Vrepy * cos(-last_yaw) + Vrepy * sin(-last_yaw);
251
252   Vx = (Vrepx * pond) + (last_twist_msg.linear.x * (1-pond));
253   Vy = (Vrepy * pond) + (last_twist_msg.linear.y * (1-pond));
254
255   //Restringir a max/min la velocidad de salida
256
257   if(Vx > x_max_vel){
258     Vx = x_max_vel;
259   }
260   if(Vy > y_max_vel){
261     Vy = y_max_vel;
262   }
263   if(Vx < -x_max_vel){
264     Vx = -x_max_vel;
265   }
266   if(Vy < -y_max_vel){
267     Vy = -y_max_vel;
268   }
269
270   }
271
272
273   void POTENTIAL_FIELD::publisher_twist_out(){
274
275   //publicar velocidad de salida
276
277   geometry_msgs::Twist speed_msg;
278
279   speed_msg.linear.x = Vx;
280   speed_msg.linear.y = Vy;
281   speed_msg.linear.z = last_twist_msg.linear.z;
282   speed_msg.angular.z = last_twist_msg.angular.z;
283
284   twist_publ_.publish(speed_msg);
285
286   ROS_DEBUG("Twist_Final vx vy vz wz : %2.6f, %2.6f, %2.6f, %2.6f", speed_msg.
        linear.x, speed_msg.linear.y, speed_msg.linear.z, speed_msg.angular.z);
287   }
288
289
290   void POTENTIAL_FIELD::publisher_twist_vector_repulsion(){
291
292   //publicar vector de repulsión resultante
293
294   geometry_msgs::Twist speed_msg_rep;
295
296   speed_msg_rep.linear.x = Vrepx;
297   speed_msg_rep.linear.y = Vrepy;
298   speed_msg_rep.linear.x = last_twist_msg.linear.z;;
299   speed_msg_rep.linear.x = last_twist_msg.angular.z;
300
301   twist_rep_publ_.publish(speed_msg_rep);
302
```

```
303     ROS_DEBUG("Twist_Final vx vy vz wz : %2.6f, %2.6f, %2.6f, %2.6f", speed_msg_rep
          .linear.x, speed_msg_rep.linear.y, speed_msg_rep.linear.z, speed_msg_rep.
          angular.z);
304
305     }
306
307 }
```

BIBLIOGRAFÍA

[1] OptiTrack, "OptiTrack hardware," accessed: 2017-07-27. [Online]. Available: https://www.optitrack.com/hardware/ 1.1

[2] Wikipedia, "Kalman filter," accessed: 2017-07-29. [Online]. Available: https://en.wikipedia.org/wiki/Kalman_filter 2.1

[3] R. Faragher, "Understanding the basis of the Kalman filter via a simple and intuitive derivation." *Lecture notes*, pp. 128 –132, September 2012. 2.1

[4] F. Golnaraghi and B. C. Kuo, *Automatic control systems*, 9th ed. John Wiley & Sons Ltd, 2009. 2.2

[5] R. Arkin, *Behaviour-based robotics*. MIT Press, 2009. 2.3

[6] ROS, "About ROS," accessed: 2017-08-4. [Online]. Available: http://www.ros.org/about-ros/ 3.1

[7] ——, "ROS concepts," accessed: 2017-08-4. [Online]. Available: http://wiki.ros.org/ROS/Concepts 3.1

[8] OptiTrack, "Motive documentation," accessed: 2017-08-6. [Online]. Available: http://wiki.optitrack.com/index.php?title=Motive_Documentation 3.3

[9] ——, "NatNet SDK," accessed: 2017-07-11. [Online]. Available: http://optitrack.com/products/natnet-sdk/ 3.4.1

[10] R. M. Taylor, "Virtual reality peripheral network - Official repo," accessed: 2017-08-10. [Online]. Available: https://github.com/vrpn/vrpn/wiki 3.4.1

[11] P. Bovbel, "vrpn_ client_ ros," accessed: 2017-08-10. [Online]. Available: http://wiki.ros.org/vrpn_client_ros 3.4.1

[12] Wikipedia, "Mandos de vuelo," accessed: 2017-08-13. [Online]. Available: https://es.wikipedia.org/wiki/Mandos_de_vuelo 3.4.2

[13] M. Monajjemi, "ardrone_ autonomy," accessed: 2017-08-15. [Online]. Available: http://ardrone-autonomy.readthedocs.io/en/latest/index.html 3.4.3

[14] M. P. Tully Foote, "Standard units of measure and coordinate conventions," accessed: 2017-08-18. [Online]. Available: http://www.ros.org/reps/rep-0103.html 4.2

[15] Wikipedia, “Right-hand rule,” accessed: 2017-08-1. [Online]. Available: https://en.wikipedia.org/wiki/Right-hand_rule 4.2

[16] OpenCV, “cv::kalmanfilter class reference,” accessed: 2017-05-15. [Online]. Available: http://docs.opencv.org/trunk/dd/d6a/classcv_1_1KalmanFilter.html 4.3.1

[17] M. S. Fadali, *Digital control engineering: Analysis and design*. Academic Press, 2012. 4.3.2

[18] B. Gassend, “dynamic_reconfigure,” accessed: 2017-05-18. [Online]. Available: http://wiki.ros.org/dynamic_reconfigure 5.1