Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Ensayo de técnicas de guiado Visuo-Inercial para vehículos aéreos con ROS-Gazebo

Autor: José Javier González Abascal
Tutor: Carlos Vivas Venegas

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Ensayo de técnicas de guiado Visuo-Inercial para vehículos aéreos con ROS-Gazebo

Autor:
José Javier González Abascal

Tutor:
Carlos Vivas Venegas
Profesor Titular

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Ensayo de técnicas de guiado Visuo-Inercial para vehículos aéreos con ROS-Gazebo

Autor: José Javier González Abascal

Tutor: Carlos Vivas Venegas

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A todos los que me han apoyado por el camino y, en especial, a mis padres.

"La mente no es un recipiente que llenar, sino un fuego que encender." – Plutarco

"No es el conocimiento, sino el acto de aprendizaje, y no la posesión, sino el acto de llegar allí, lo que concede mayor disfrute" – Carl Friedrich Gauss

Agradecimientos

Gracias en general a todas aquellas personas con las que he podido compartir los momentos de mayor agobio. Gracias a mi familia que me ha apoyado en todo momento: a mis padres, a mis tíos y a mis abuelos. Gracias también a los amigos que han estado ahí cuando ha hecho falta. Y gracias a todas aquellas personas que he conocido durante estos 4 años y que han aportado su granito de arena para hacer de estos 4 años una experiencia inolvidable.

Por último, un saludo a esos profesores que han sabido motivar a sus alumnos, que han sabido transmitir sus conocimientos y que demuestran día a día que les apasiona su trabajo.

José Javier González Abascal

Sevilla, 2020

Resumen

El objetivo de este trabajo fin de grado es ensayar un conjunto de técnicas de estimación de la posición y orientación de un vehículo aéreo no tripulado (UAV) empleando principalmente métodos de visión monocular que son complementados empleando fusión sensorial con la unidad de medida inercial (IMU).

Se plantean dos metodologías en el problema de visión: (i) una primera en la línea de los marcadores naturales en la que el sistema no dispone de marcadores específicos conocidos *a priori* que permitan resolver el problema de modo absoluto, por lo que la solución se proporciona en términos de desplazamientos relativos con respecto al anterior, y (ii) otra basada en marcadores artificiales en la que el sistema sí que dispone de marcadores visuales que permiten el posicionamiento absoluto con respecto a unos elementos definidos con anterioridad.

Como entorno de simulación haremos uso de Gazebo, un potente software gratuito orientado al trabajo con vehículos robotizados, que cuenta con un motor de física robusto y permite la generación de gráficos de alta calidad en simultaneidad con el transcurso de la propia simulación. Para la implementación de los algoritmos, se trabajará con ROS, un producto de código abierto consistente en un conjunto de librerías y herramientas software destinadas a la programación de sistemas robóticos. Esta plataforma permite, además, trabajar en simulación con los mismos ejecutables que posteriormente serían implementados en un hardware real y cuenta con los procedimientos necesarios para lograr la correcta comunicación con Gazebo durante la simulación. Los algoritmos de procesado de imágenes se ejecutarán empleando OpenCV que es una plataforma de código abierto ampliamente documentada.

En la parte final del trabajo, se analizan los resultados de las simulaciones gráfica y numéricamente, haciendo uso del entorno ROS/Gazebo y de los algoritmos de procesado de imagen y fusión sensorial propuestos.

Abstract

The aim of this end-of-degree project is to compare several pose estimation methods for Unmanned Aerial Vehicles (UAV) mainly based on monocular vision. These monocular vision methods will be improved by fusing it with Inertial Measurement Unit (IMU) data.

Two different approaches are chosen for the monocular vision based pose estimation: (i) The first one does not require any previous knowledge about the environment as it is based on natural landmarks, so the output is a displacement between the previous pose and the current one and (ii) a second approach related to fiducial markers which takes advantage of objects or elements whose dimensions and shape are known beforehand.

Gazebo, an open-source robot simulator, will be used because of its high-quality graphics and its robust simulation engine. In order to implement the algorithms for the UAV and connect to Gazebo, ROS will be needed. Therefore, the ROS files used during the simulation can be saved for future real-life experiments. In addition, vision related algorithms will include OpenCV libraries as it is also open source.

Finally, results obtained from the simulations will be analysed and compared.

Índice

Índice de Tablas

Índice de Figuras

Notación y siglas

EKF	Extended Kalman Filter
FLANN	Fast Library for Approximated Nearest Neighbours
IMU	Inertial Measurement Unit
MAE	Mean Absolute Error
MSE	Mean Squared Error
ORB	Oriented FAST and Rotated BRIEF
RANSAC	RANdom Sample Consensus
ROS	Robot Operating System
SIFT	Scale-Invariant Feature Transform
SURF	Speeded Up Robust Features
UAV	Unmanned Aerial Vehicle

1 INTRODUCCIÓN

En los últimos tiempos, el uso de drones se ha normalizado cada vez más tanto a nivel industrial como de ocio debido, en gran parte, a las continuas mejoras que se están desarrollando y a las ventajas de su uso. Algunas de estas ventajas son el uso de un UAV (*Unmanned Aerial Vehicle*) en tareas que pudieran resultar peligrosas para personas, la posibilidad de captar imágenes con nuevas perspectivas sin la necesidad de subirse a un helicóptero o agilizar el transporte de cargas.

Algunas de las tareas anteriores podrán realizarse tanto de forma manual, es decir, con un operario controlando los movimientos del UAV como de manera automática. Este último caso sería el más sensible de todos puesto que el dron debe tener la capacidad de reconocer su estado y los estímulos que el entorno ejerce sobre él para así poder actuar en consecuencia. Es por ello por lo que las tareas que requieran una mayor independencia son las más difíciles de solucionar.

En línea con lo anterior, uno de los principales problemas que se da cuando un UAV se encuentra en un entorno no conocido es el del posicionamiento. Este caso se ha solucionado típicamente a través de la señal GPS, que permite ubicar el dron con bastante precisión, pero pudiera ocurrir que esta fuera demasiado débil como para poder utilizarla. Es en ese momento en el que habría que buscar soluciones alternativas al problema del posicionamiento.

Para obtener dicha información habrá que basarse en la información recogida por los sensores del UAV y combinarla de la manera adecuada para obtener el mejor resultado posible. Intentando resolver estos problemas se han publicado diferentes artículos con motivaciones parecidas aunque con distintas suposiciones y algoritmos. Por ejemplo, se han realizado trabajos de recopilación de métodos de posicionamiento por visión como en [1][1], pero sin estar especialmente destinados a la integración en un UAV. Otros artículos como [2] realiza pruebas para un robot con movimiento plano usando la visión monocular y la IMU, mientras que en [3] se toman imágenes a poca altura (hasta 2m) y estudian otros efectos derivados del uso del acelerómetro. También ha habido intentos de evaluar las posibilidades de los métodos de posicionamiento basados exclusivamente en visión monocular en [4] y mezclados con varias IMUs e incluso radar en [5] aunque sin tener muy en cuenta la altura estimada. Los trabajos anteriores no muestran de manera clara los resultados de dichos algoritmos para alturas variables mayores de 3m lo cual sería relevante en el caso de un UAV al aire libre o en un espacio grande que, por algún casual, no tuviera acceso a señal GPS. Por ello, se va a realizar una serie de ensayos usando algunos algoritmos de visión para un sistema de visión monocular situado en la parte inferior del UAV y así examinar los resultados de posición y orientación (los 6 grados de libertad) obtenidos.

Para realizar estas estimaciones se tomarán varias vías de investigación paralelas:

Por un lado, se incorporará al entorno algún objeto que se haya caracterizado previamente (forma y dimensiones conocidas) de tal forma que, si se detecta el objeto en la imagen, se pueda extraer la

posición del UAV gracias a las deformaciones que este elemento experimenta con la perspectiva. A este método se le llama “basado en marcadores artificiales” puesto que la referencia absoluta a partir de la cual se calcula la posición será ese objeto conocido. Este método sin embargo será incapaz de funcionar cuando dicho objeto deje de estar en el campo de visión.

Por el otro, se intentará estimar el posicionamiento calculando cómo pasar del fotograma anterior al actual. A partir de la relación entre los fotogramas se deducirá el desplazamiento que ha experimentado el UAV. En este caso, al no tener información previa sobre el entorno, se tendrán que detectar puntos que presenten alguna característica diferenciadora (como ser una esquina). Puesto que estos puntos dependen del entorno y no son predefinidos, a este método se le llama “basado en marcadores naturales” y será el principal método a desarrollar en este trabajo porque no impone restricciones de visibilidad de algún objeto concreto (si bien es necesario que la imagen no sea un color plano para que se pueda extraer información).

Para mejorar los resultados anteriores, se realizará alguna modificación al código utilizado. Una de ellas será la incorporación de un filtro de Kalman extendido para fusionar los datos proporcionados por el algoritmo de visión basado en marcadores naturales y los valores de la IMU.

Finalmente, se extraerán una serie de conclusiones relativas a los resultados obtenidos.

Para realizar todas las simulaciones, obtener de información de los sensores y su tratamiento se recurrirá a una serie de herramientas que se explican en más detenimiento en el próximo capítulo.

2 PROGRAMAS Y ELEMENTOS UTILIZADOS

Como se comentó en el capítulo anterior, se procede a indicar los elementos y programas que serán utilizados tanto para la simulación, como para la gestión de la información y demás tareas relacionadas.

Se simulará el dron ErleCopter de la empresa ErleRobotics que será el encargado de tomar las imágenes y, en su caso, de mandar también la información proveniente de la IMU. El ErleCopter se simulará en Gazebo ya que la empresa proporciona los ficheros de parámetros del UAV necesarios para la simulación. Para conectarnos con la información del ErleCopter será necesario utilizar ROS (*Robot Operating System*) y, para poder gestionar y trabajar sobre las imágenes, habrá que recurrir a las librerías de OpenCV. A continuación se da una breve indicación de cada uno de estos elementos que se utilizan a lo largo del proyecto.

Todos los programas y archivos realizados se pondrán a su disposición en el siguiente repositorio de Github: https://github.com/JJavierga/TFG_posicionamiento_por_vision. Además, en el apartado C de los anexos se indican los programas creados y su finalidad.

2.1 ErleCopter

El ErleCopter es un cuadricóptero basado en Linux desarrollado por la empresa española ErleRobotics. Esta empresa cerró en julio de 2019, resultando a su vez en el cierre de sus páginas web y foros, lo que ha dificultado la búsqueda de una fuente que explicara sin omisiones o errores el proceso de instalación de los programas requeridos y su uso. Finalmente, mezclando información de varias fuentes se consiguió poner en funcionamiento la simulación del ErleCopter dentro de Gazebo. El proceso de instalación, el funcionamiento y las fuentes vienen detalladas en el anexo A.

Ilustración 1. ErleCopter.
[https://robots.ros.org/erlecopter/]

2.2 Gazebo

Gazebo es un software gratuito de simulación de la física de sólidos en alta fidelidad orientado principalmente al desarrollo de robots y que está gestionado por la fundación Open Robotics. Aunque inicialmente fuera concebido para la simulación en exteriores y bajo diversas condiciones, en la actualidad se puede usar tanto para exterior como para interior, siendo esta última vertiente la más común entre sus usuarios. Es por esa concepción inicial que el programa recibe el nombre *gazebo*, que es un tipo de escenario exterior o cenador.

Además, su carácter de código abierto ha fomentado la integración de otros programas y elementos con él. En este caso interesará sobre todo la unión de ROS y Gazebo.

Gazebo es por tanto, el simulador utilizado en este proyecto. Para la realización de la simulación será necesario añadir otros elementos al entorno en el que se mueve el dron ya que, en la versión por defecto que viene con los archivos de simulación del ErleCopter, no hay objetos que puedan servir como puntos de referencia para la cámara del dron (el suelo es un color plano y solamente aparecen el dron y unas paredes también de color plano). Por esto último, se va a cambiar el entorno (entendiéndose como tal el "lugar" en donde va a volar el dron) tal y como se indica en el anexo A.4. Los pasos necesarios para instalar Gazebo vienen explicados también en el anexo A.

La versión utilizada es Gazebo 7 tal y como se explica en la guía de simulación del ErleCopter.

2.3 ROS

ROS (*Robot Operating System*) es un marco de desarrollo software para robots que pretende simplificar tareas complejas relacionadas con el control y simulación del robot proporcionando los servicios estándar de un sistema operativo. Además, ROS se caracteriza por estar basado en una arquitectura de grafos, los cuales se envían mensajes los unos a los otros y que son creados por un proceso maestro (explicación más detallada en capítulo 3 de este trabajo fin de grado y [6]). Cabe

destacar que, al ser un software libre, está abierto a aportaciones de código de usuario o grupos independientes, permitiéndose así agilizar su desarrollo, dando lugar a herramientas que permiten conectarlo con, por ejemplo, Gazebo.

La instalación de ROS se realiza de manera paralela a la de Gazebo (con el fin de simular el ErleCopter) y aparece descrita en el anexo A.

La versión de ROS utilizada es ROS Indigo tal y como se recomienda en la guía de instalación destinada al ErleCopter.

2.4 OpenCV

OpenCV es una biblioteca de funciones destinadas principalmente a la visión artificial y que fue desarrollada inicialmente por Intel en 1999. Es una de las bibliotecas de visión artificial más utilizadas a día de hoy lo que, unido con su carácter de código abierto, facilita la búsqueda de información sobre su funcionamiento. Incorpora funciones tantos de estimación de posición de la cámara, como de reconocimiento facial, de gestos, segmentación… Para su utilización se recomienda el material web y los libros [7] y [8].

A través de las funciones de OpenCV se conseguirá extraer la información necesaria de las imágenes capturadas por la cámara. El proceso de instalación viene detallado en el anexo B ya que la versión incluida en ROS no nos vale al no incorporar directamente el módulo de SURF.

Por las funciones que contiene, se utilizará la versión 3.1.0 de OpenCV.

3 ESTIMACIÓN DE LA POSICIÓN CON IMÁGENES POR MARCADORES NATURALES

En este capítulo se plantea el desarrollo teórico de los pasos necesarios para la obtención de los movimientos de la cámara mediante la comparación entre fotogramas. Para realizar esta tarea se utilizará el concepto de homografía. La homografía es una transformación que permite convertir imágenes proyectadas en un sensor en imágenes proyectadas en un sensor en otra posición. Es decir, es una transformación que permite pasar del fotograma anterior al fotograma siguiente. Sin embargo, lo que se desea no es pasar de la imagen anterior a la actual, sino calcular cuál es esa transformación. Para ello, será necesario tomar un conjunto de puntos de las dos imágenes que sean característicos por el contraste o forma de su vecindad y compararlos para ver cuáles se corresponden entre sí. Para realizar dicha comparación será necesario tener una medida de las propiedades de ese punto y su entorno que hacen que sea diferente al resto. Una vez definida y calculada esta medida para cada punto, se podrá realizar la comparativa entre ellos y, a partir de aquellos que presenten mayor similitud, estimarse una homografía. Finalmente, solamente faltaría extraer de ella las rotaciones y traslaciones de la cámara y, seguidamente, cambiar de base a la del UAV.

A continuación, se indica esquemáticamente el proceso para la obtención del movimiento de la cámara de un fotograma al siguiente, el cual también se muestra en la Ilustración 2:

- Detectar puntos que destaquen (por ejemplo, por el contraste en su vecindad) en la imagen anterior y la actual (llamados marcadores naturales, *keypoints* o puntos de interés).
- Calcular alguna entidad (descriptores) que sirva para describir cómo son dichos puntos.
- Comprobar qué descriptores de la imagen anterior son muy parecidos a los de la actual.
- A través de esos puntos relacionados por la similitud de sus descriptores, obtener una transformación que convierta los puntos del fotograma anterior en los del fotograma actual (homografía).
- Extraer de esa transformación el movimiento de la cámara.

Cada una de estas tareas se puede afrontar con diferentes algoritmos. En este caso nos basaremos en una combinación de algoritmos ya incorporados por separado en OpenCV 3.1.0: *Fast Hessian*, *SURF*, *FLANN(Kd-tree)* y *RANSAC*). Estos pasos serán explicados a continuación tanto de manera teórica para entender su comportamiento como de forma más práctica orientada a la implementación.

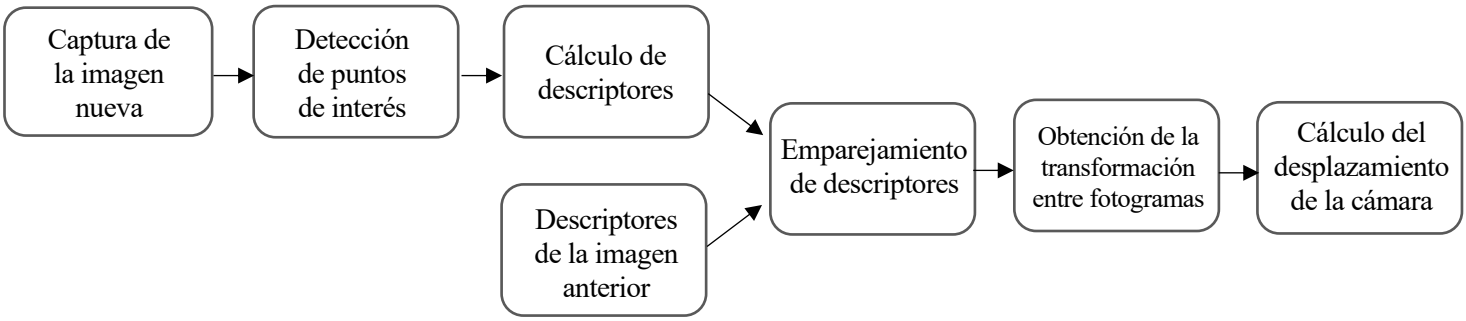

Ilustración 2. Diagrama de las operaciones a realizar para obtener una estimación de los movimientos de la cámara a partir de las imágenes.

3.1 Detección de puntos de interés

En primer lugar habrá que averiguar qué puntos de la imagen pueden ser representativos como pueden ser las esquinas o bordes. Existen varias formas de realizar esta tarea: detección de borde (*corner detection)*, detección de manchas (*blob detection*) y detección de regiones (*región detection).* Para este caso se ha recurrido a un método de detección de manchas basado la función hessiano llamado *Fast Hessian Detector*. Como su propio nombre indica esta opción se basa en el hessiano, pero realizando una serie de simplificaciones que permiten al algoritmo ejecutarse en un menor tiempo. Como el hessiano indica la "aceleración" del nivel de intensidad de los píxeles en una dirección del espacio, mientras mayor sea este valor, más rápido cambia dicho valor de gris, por lo que si obtenemos valores pequeños será que esa zona será una superficie, mientras que para valores grandes podrá ocurrir que sea en un borde por ejemplo.

Previo a la explicación será necesario indicar el significado de los términos octava y nivel:

- Octavas (*Octaves*): Conjunto de imágenes en las que se ha utilizado un tamaño de máscara (parámetro L) determinado a la hora de calcular el hessiano. Una imagen puede tener diferentes octavas en caso de que cada una de ellas utilice diferentes tamaños de máscara. Habitualmente cada nueva octava suele redimensionarse a la mitad para mantener la complejidad de cálculos entre escalas.

- Nivel (*Layer*): Cada *octave* puede tener a su vez diferentes niveles, cada uno de los cuales tiene un valor diferente de desenfoque gaussiano (una varianza diferente aplicada).

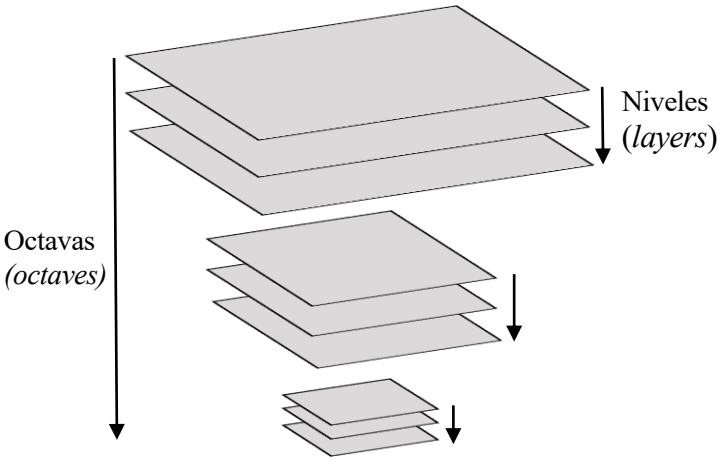

Ilustración 4. Representación gráfica de octaves y layers.

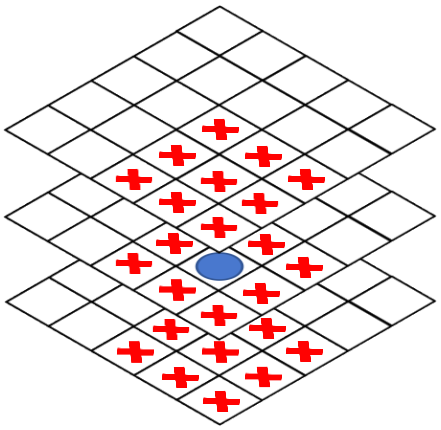

Ilustración 3. Representación gráfica de la vecindad de 3x3x3 entre 3 capas diferentes.

Partiendo de la imagen original, se crean diferentes octavas cada una de las cuales tiene una resolución la mitad del anterior y, a cada octava, se le aplican desenfoques gaussianos de distinta varianza para dar un conjunto de capas. En cada uno de los elementos de las distintas capas puede calcularse el hessiano a través de máscaras y, a continuación, calcular su determinante de manera aproximada según la siguiente expresión sacada de [9]:

$$DoH^L(u) := \frac{1}{L^4} * \left(D_{xx}^L u \cdot D_{yy}^L u - \left(w D_{xy}^L u \right)^2 \right)$$

Donde L es un factor de escala relacionado con el tamaño de la máscara, w es un peso de ponderación que suele valor 0.912, D_{ij} es la componente (i,j) del hessiano y u el valor de la imagen discreta. El lector interesado puede consultar más detalles en el artículo SURF [9]. Finalmente, solo falta seleccionar los máximos locales del determinante en una vecindad de 3x3x3, es decir, comparar el valor del determinante no solamente con los pixeles vecinos, sino también con aquellos de los niveles adyacentes. Para hacer el algoritmo más robusto y resistente a ruido, se incorpora un umbral mínimo por debajo del cual, aunque el hessiano sea un máximo local, no se tiene en cuenta.

3.2 Descriptores locales

Una vez elegidos los puntos de interés, será necesario obtener una serie de características propias de estos puntos para que, en la fase de comparación, podamos tener una forma de ver las correspondencias entre ellos. A este conjunto de características propias de la región en torno a un punto es a lo que se denomina descriptor y se representa por un vector cuyas componentes no son más que las propias características y el cual es deseable que sea invariante a transformación geométricas y robusto ante ruido, cambios de iluminación o contraste. En nuestro caso, se utiliza un descriptor invariante a rotaciones y escala llamado SURF (*Speeded Up Robust Features)*[9], basado en SIFT [10], pero de mayor velocidad de cómputo. Si se quiere información sobre estos algoritmos también se puede leer el capítulo 4 de [11] o [12], aunque si se quiere buscar otros algoritmos posibles tanto para la detección como para la descripción, se recomienda leer [13].

El descriptor SURF se obtiene a partir de tomar una región orientada en torno al punto de interés que, a su vez, se divide en 16 subregiones, dando lugar a una cuadrícula de 4x4.

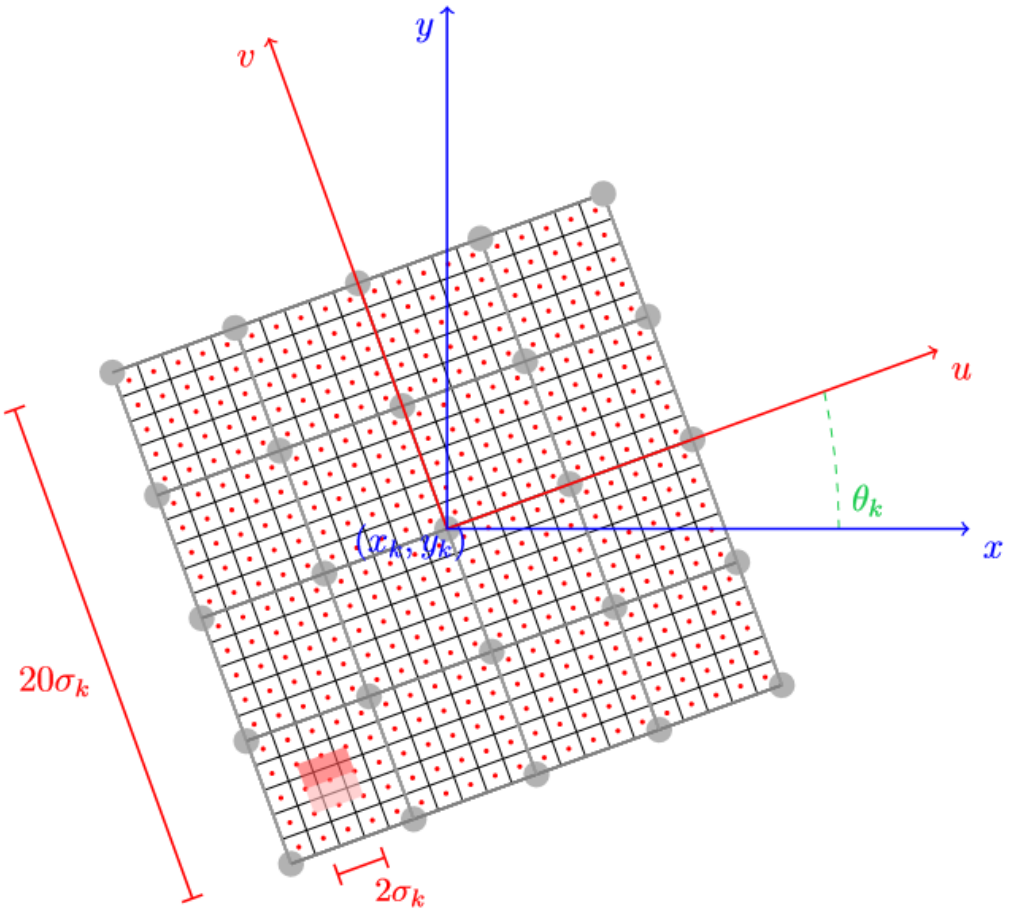

Ilustración 5. Representación de la rejilla escalada y orientada. La rejilla, dividida en 16 subregiones, se utiliza para aplicar SURF a la vecindad del punto de interés $\boldsymbol{X_k}$: $(\boldsymbol{x_k}, \boldsymbol{y_k}, \boldsymbol{L_k})$ con una orientación $\boldsymbol{\theta_k}$. Sacado de la referencia [9].

En cada una de estas subregiones se calculan 4 valores dando lugar a:

$$\forall (i,j) \in [\![1,4]\!]^2 \quad , \quad \mu_k(i,j) = \begin{pmatrix} \sum_{(u,v)\in R_{i,j}} d_x(u,v) \\ \sum_{(u,v)\in R_{i,j}} d_y(u,v) \\ \sum_{(u,v)\in R_{i,j}} |d_x(u,v)| \\ \sum_{(u,v)\in R_{i,j}} |d_y(u,v)| \end{pmatrix}$$

La dos primeras componentes son la suma de los gradientes de esa región en las direcciones x e y, mientras que las dos últimas son la suma de los valores absolutos de los gradientes en las dos direcciones. Adicionalmente, podría realizarse una normalización de dicho descriptor para obtener mejores resultados antes cambios de contraste.

Uniendo las características de las 16 subregiones existentes se obtiene el descriptor SURF, que resulta ser un vector de 64 componentes. Toda la información detallada sobre este algoritmo y el hessiano rápido puede encontrarse en [9], del cual se ha extraído la Ilustración 5.

La detección de puntos de interés y cálculo de sus descriptores se puede realizar de una sola vez en OpenCV. En primer lugar hará falta crear un puntero de tipo *Feature2D* que almacenará el algoritmo a utilizar. Posteriormente, se crean los objetos Mat que almacenarán los descriptores y el vector de puntos de interés (tipo *KeyPoint*). Finalmente, se accede a la función de detección de puntos y cálculo de descriptores. En nuestro caso, al utilizar SURF como descriptor, el detector predeterminado es el Hessiano rápido.

Puesto que se desea un algoritmo eficaz, pero al mismo tiempo lo más rápido posible, se realizan una serie de simulaciones del comportamiento global del algoritmo para valores entre 100 y 1500 del umbral mínimo de detección de puntos (recuérdese del apartado 2.1). Mientras mayor sea dicho umbral, menos puntos se detectarán y, por ende, menos descriptores habrá que calcular posteriormente, por lo que buscamos un valor que proporcione un buen resultado en el menor tiempo posible. Los resultados muestran que, para valores del umbral superiores a 1000, el bajo número de puntos afecta negativamente al comportamiento global de la estimación de la posición. Por ello, se toma este valor límite como el nuevo umbral, consiguiéndose así una reducción de en torno al 30% del tiempo de ejecución con respecto al valor por defecto.

3.3 Comparador de descriptores

Una vez obtenidos los descriptores de los puntos de interés, habrá que compararlos con aquellos obtenidos en otra imagen para ver cuáles son parecidos y cuáles no. Existen diferentes formas de afrontar este problema como algoritmos de emparejado de fuerza bruta (*Brute force matching*), basados en *Kd-trees*… En nuestro caso se aplicará el algoritmo implementado en la librería FLANN (*Fast Library for Approximated Nearest Neighbours*) que, por defecto, recurre a *Kd_trees* ya que funcionan mejor que la fuerza bruta conforme aumenta el número de elementos a analizar.

Sin embargo, no todos los emparejamientos obtenidos van a ser correctos, por ello, a continuación sería aconsejable realizar un pequeño filtrado. La forma implementada en nuestro caso es detectar el punto a mínima distancia de su pareja y aquel que se encuentra a mayor distancia y extraer de ahí el abanico de posibles distancias. Finalmente, nos quedamos con los puntos que se encuentran en el primer tercio de dicha amplitud.

En OpenCV, solo hará falta crear un vector de tipo *DMatch* que almacene los emparejamientos, crear un objeto de tipo *FlannBasedMatcher* y ejecutar el comando.

Como también se comentó en el apartado 2.2, es recomendable filtrar el resultado para eliminar aquellos emparejamientos que no arrojen buenos resultados. En la página web de OpenCV [26] sugieren quedarse con aquellos puntos cuya distancia entre descriptores se encuentre entre la distancia

entre descriptores para el mejor emparejamiento y esa misma multiplicada por un factor. Esto quiere decir que si el fator es, por ejemplo, 3, y el mejor emparejamiento se encuentra a 0.1, sólo nos quedaremos con aquellos puntos cuya distancia sea menor de 0.3.

Sin embargo, aparte de tener que sintonizar el valor de ese factor, este método da problemas en el caso de que haya un descriptor inusualmente parecido entre las dos imágenes. En algunas ocasiones se dan puntos cuya distancia es menor de 0.001, por lo que o el factor es muy grande, o el algoritmo se quedará únicamente con unos pocos puntos que pueden no ser representativos de la muestra o incluso ser insuficientes puntos para posteriores cálculos (menos de 4 puntos).

Una opción sería que en esos casos no hubiera estimación y se desechara la medida, pero entonces se estaría diciendo que cuando se tienen puntos muy parecidos la medida no vale, cuando ese caso es en realidad cuando la transformación entre las dos imágenes puede parecerse más.

La opción que se ha implementado es tomar el abanico total de distancias entre emparejamientos, esto es la diferencia entre la distancia del mejor emparejamiento y del peor, y quedarse con una porción de esta. En nuestro caso se ha optado por tomar aquellos emparejamientos cuyas distancias se encuentren a un tercio de la amplitud de distancias. Es decir, si el mínimo es 0.1 y el máximo es 1.3, se elegirán aquellos puntos que se encuentren a una distancia menor de 0.5 . Aunque de esta forma se podría estar incorporando un mayor número de puntos que con el otro método, se aseguraría un número suficiente de ellos ya que estos suelen estar dispersos. Además, se eliminan así esos posibles casos en los que una distancia mínima muy pequeña limite la cantidad de puntos.

3.4 Encontrar transformación entre imágenes

El siguiente paso es ver qué transformación permite pasar de los puntos de interés de la primera imagen a los de la segunda. En nuestro caso, esto no sería más que un cambio de perspectiva, es decir, una homografía. Para encontrar dicha perspectiva se recurre al algoritmo RANSAC ya que suele ser más rápido que otros y sin dejar de ser robusto.

3.4.1 RANSAC

RANSAC (*RAndom SAmple Consensus)* es un método iterativo que, como su propio nombre indica, se basa en el "consenso" de los diferentes datos. Esto significa que, partiendo de un subconjunto inicial de datos, hace una hipótesis sobre la distribución general, la cual es posteriormente puesta a prueba con el resto de valores. Si hay suficientes puntos que se ajusten a dicho modelo, se llega a un consenso y la siguiente hipótesis será formada por aquellos valores que forman parte del consenso actual. En caso contrario, es descartada y se prueba con otro subconjunto. Este proceso se repite una serie de veces, es decir, es un método iterativo, por lo que su coste computacional aumenta conforme aumenta el número de iteraciones (y de *outliers*).

3.4.2 Obtención de rotaciones, traslaciones y normal de la matriz de homografía

Una homografía se representa como una matriz de 3x3 que permite pasar de la imagen inicial a la final, en coordenadas homogéneas, de la siguiente forma:

$$\begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = \begin{pmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix}$$

Por lo que habrá que extraer de ahí la información que necesaria, es decir, la rotación y la traslación de una imagen a la otra teniendo en cuenta que, a la hora de calcular la homografía, la traslación se realiza de manera proporcional a un factor de escala que indica la distancia de la cámara al plano objeto. Por ello, cualquier traslación obtenida por homografía vendrá, de base, sin factor de escala incluido.

Para realizar este paso deberán conocerse los parámetros intrínsecos de la cámara y las deformaciones que esta aplica. Esta tarea puede calcularse de forma experimental mediante procedimientos de calibración de cámara ampliamente estudiados y descritos en la literatura. En nuestro caso, al tratarse de un entorno simulado, el problema pasa a ser el de acceder a los parámetros intrínsecos de la cámara que aplica el simulador de Gazebo.

Para obtener la matriz de homografía se puede recurrir a la función *findHomography( )* de OpenCV. Esta función recibe las dos imágenes e incorpora como tercer parámetro el algoritmo a utilizar que, en nuestro caso, será el anteriormente explicado RANSAC. A partir de esta matriz se podría obtener la traslación y rotación con la función *DecomposeHomographyMat( )*, pero, como se acaba de mencionar, haría falta antes la matriz intrínseca de la cámara.

Nosotros estamos en el segundo caso, una simulación. Ateniendo a los *topics* activos durante la simulación del Erle-Copter, nos podemos fijar en que hay uno que se llama */erlecopter/bottom/image_raw* que comparte las imágenes que graba la cámara inferior y otro llamado */erlecopter/bottom/camera_info*. Si se ejecuta *rostopic echo /erlecopter/bottom/camera_info*, devuelve por la pantalla una serie de valores tal y como se ve en la Ilustración 6.

```
javier@JJavierGA: ~
header:
  seq: 64
  stamp:
    secs: 374
    nsecs:  92500000
  frame_id: erlecopter_bottomcam
height: 360
width: 640
distortion_model: plumb_bob
D: [0.0, 0.0, 0.0, 0.0, 0.0]
K: [374.6706070969281, 0.0, 320.5, 0.0, 374.6706070969281, 180.5, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [374.6706070969281, 0.0, 320.5, -0.0, 0.0, 374.6706070969281, 180.5, 0.0, 0.0
, 0.0, 1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
```

Ilustración 6. Salida comando rostopic echo /erlecopter/bottom/camera_info

La cámara utiliza el modelo de distorsión Plumb Bob (o Brown-Conrady), el cual es un modelo que combina distorsión radial y tangencial, quedando definido por 5 parámetros. Estos cinco valores están almacenados en el vector D dando lugar a un vector de ceros. Esto quiere decir que la simulación supone una lente ideal.

Otro parámetro que proporciona el *topic* es la matriz K, que tiene forma de matriz intrínseca de la cámara. Al surgir la posibilidad de que realmente signifique otra cosa, se han buscado referencias y se ha encontrado que hay otros trabajos como [14] que avalan que, en efecto, los parámetros anteriores son correctos. Por lo que:

$$Matriz\ intrínseca: K = \begin{pmatrix} 374.67 & 0 & 320.5 \\ 0 & 374.67 & 180.5 \\ 0 & 0 & 1 \end{pmatrix}$$

Una vez obtenida, y gracias a la función *DecomposeHomographyMat( )*, obtenemos las cuatro soluciones posibles:

$$Solución\ 1:\ R_a, t_a,\ n_a$$
$$Solución\ 2:\ R_b, t_b,\ n_b$$
$$Solución\ 3:\ R_a, -t_a, -n_a$$
$$Solución\ 4:\ R_b, -t_b, -n_b$$

3.4.3 Reducción de soluciones de la homografía

La reducción de 4 a una única solución no es algo trivial tal y cómo se explica en este artículo [15]. Dos de estas soluciones pueden eliminarse utilizando la restricción de visibilidad. Esta quiere decir que la cámara no puede estar mirando en dirección opuesta a los objetos. Para pasar de 2 a la solución final, hay varias opciones. La primera opción es buscar aquella solución cuya normal sea más cercana a [0,0,1] ya que el dron va a estar medianamente horizontal y ésta tomará un valor próximo. Otra opción es imponer que la solución sea aquella más consistente con el resultado anterior, es decir, aquella cuya rotación suponga un menor cambio con respecto a la rotación anterior. Esta última opción es la que se ha escogido. En resumen, el problema podría solucionarse con las siguientes dos simplificaciones:

- El dron nunca va a volar invertido. Esto implica que la cámara siempre estará apuntando hacia el suelo o, visto de otra manera, siguiendo la referencia de la normal establecida en el artículo anterior y mostrada en la Ilustración 7 sacada de ese artículo (por defecto apunta en la misma dirección que la cámara), la componente Z de la normal debe ser siempre positiva. Una Z negativa indicaría que el dron está del revés. Esto deja solamente dos soluciones restantes. La solución 1 y la 2.

- Principio de mínimo cambio. Se impone que la solución sea consistente si es la que minimiza el cambio en la rotación con respecto a la posición anterior.

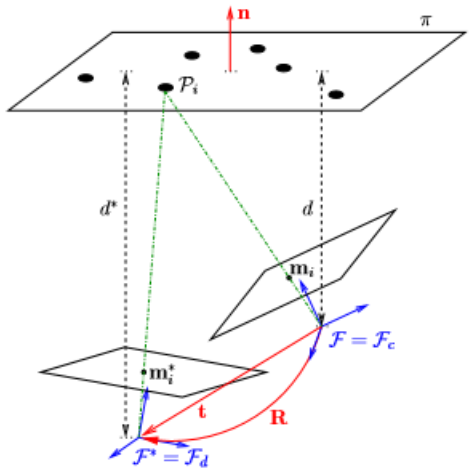

Ilustración 7. Sistemas de referencia inicial (F), deseado (F) y notación relacionada.*

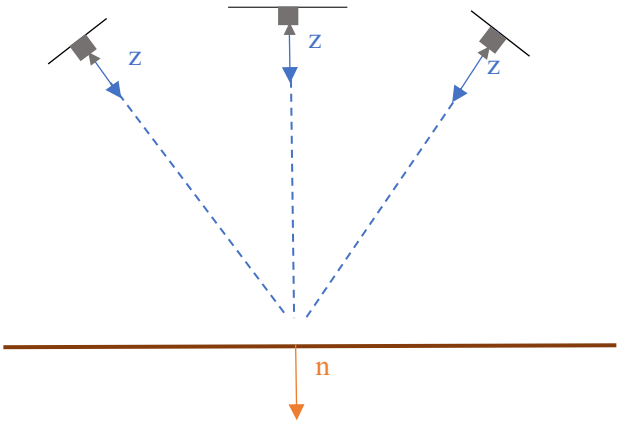

Ilustración 8. Ejes Z de la cámara del dron y normal del suelo cuando el vuelo no es invertido.

Para conseguir calcular cuál es la solución última aplicando la segunda suposición, se parte de que se conoce la matriz de rotación anterior (R_{12}) que pasa de la imagen 1 a la imagen 2 y hay varias opciones para la actual (R_{23}^i).

Existe una forma de comparar ambas matrices de rotación a través de sus ejes de giro y ángulo de rotación. Para ello, hay que indicar que toda matriz de rotación puede descomponerse como un giro de un ángulo θ en torno a un eje u. Esto último es trivial si se piensa que toda matriz de rotación tiene a lo sumo 3 grados de libertad (tres giros en torno a tres ejes), por lo que tiene que haber una manera de representar dicha información en solo tres variables. Una de ellas es el vector de rotación, el cual tiene la dirección del eje de rotación, el sentido indicado por el ángulo y, como módulo, el ángulo girado. Este vector puede obtenerse a través de la fórmula de Rodrigues como puede leerse en el artículo [16].

En definitiva, se desea comparar u_1 proveniente de R_{12} con los diferentes u_2^i resultantes de cada R_{23} y quedarse con aquel que sea más cercano. Sin embargo, R_{12} representa el giro que realizan los puntos de la imagen 1 a la imagen 2, es decir:

$$R_{12} \cdot imagen_1 = imagen_2$$

Por lo que $u_1$estará expresado en base de la imagen 1 y los u_2^i estarán expresados en la base de la imagen 2. Será necesario por tanto expresar ambos vectores en la misma base. Esto se consigue de manera sencilla dándose cuenta de que, si los ejes de 2 son los ejes de 1 multiplicados por una matriz de rotación, los ejes han girado mientras que el vector sigue apuntando hacia la misma dirección, o, visto al revés, el vector ha girado en sentido contrario a los ejes. Es decir, las nuevas coordenadas del vector de rotación vendrán dadas por la inversa del giro de los ejes:

$$(R_{12})^{-1} \cdot u_1|_{\mathcal{B}_1} = u_1|_{\mathcal{B}_2}$$

Por lo que ya podrían compararse $u_1|_{\mathcal{B}_2}$ con u_2^i y obtener aquel que minimice la distancia:

$$u_2 = \arg\min \quad |u_2^i - ((R_{12})^{-1} \cdot u_1)|$$

Y, una vez conocida la solución, se puede saber a qué matriz de rotación corresponde, quedando finalizado el problema de asignar una matriz de rotación al paso de la imagen anterior a la actual.

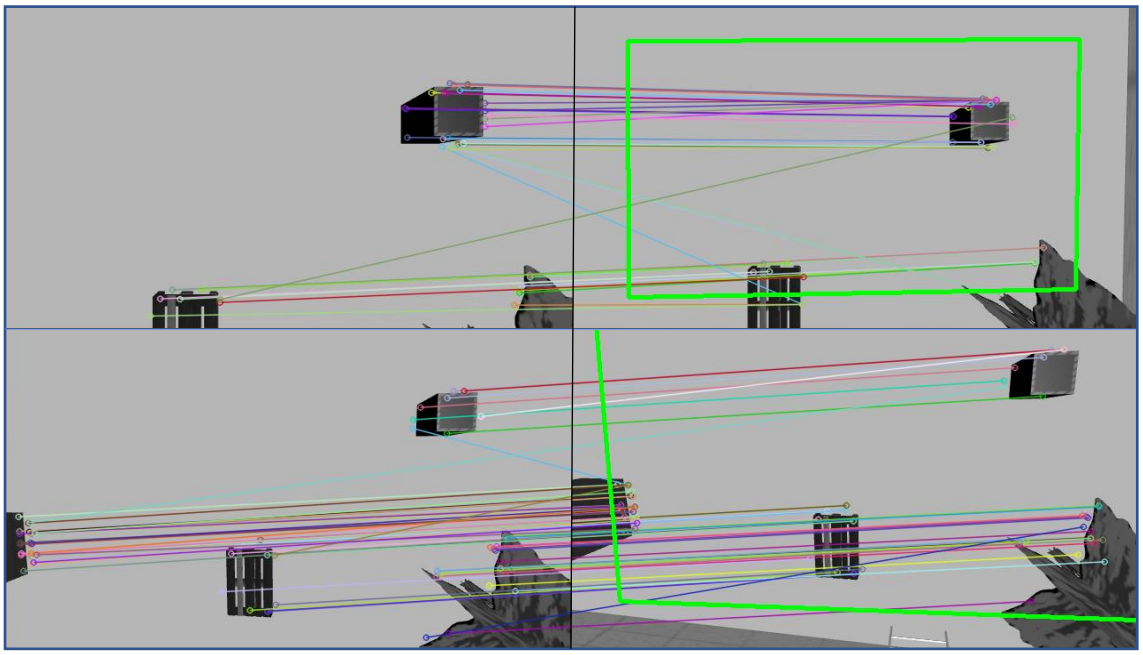

Ilustración 9. Detección de homografía entre dos fotogramas no consecutivos obtenidos por la cámara. Las imágenes de la izquierda son los primeros fotogramas a comparar y, las de la derecha, los segundos fotogramas. El cuadrilátero verde se corresponde con el resultado de aplicar la homografía calculada a las esquinas del primer fotograma y unir los puntos.

3.4.4 Determinación del factor de escala

El factor de escala, como se ve en la Ilustración 7**,** es la distancia desde el plano deseado a aquel que contiene los puntos de interés. En nuestro caso este plano no está definido ya que los puntos de interés pueden estar tanto en el plano del suelo como en del tejado, a la altura de una papelera o a la de cualquier otro objeto de la imagen. Entonces, ¿cómo se puede saber este valor? La única opción es estimarlo y esto puede realizarse de diversas formas:

- Suponiendo que el suelo ocupe una parte importante de la imagen, se puede inferir que la mayoría de los puntos detectados pertenecen al plano del suelo. Si el dron despegase inicialmente hasta una altura conocida, se sabría la altura inicial y, el resto de factores de escala necesarios, se tomarían de las futuras estimaciones de la altitud del dron.
- También se puede usar algún elemento de referencia cuyas dimensiones sean conocidas y así poder inferir la distancia a este mediante proporciones. Fijándose únicamente en la deformación de este objeto se tendría un posicionamiento absoluto con respecto a él. Sin embargo, esto no permitiría al UAV salir del rango de visión del objeto.

- Otra opción sería realizar un filtro de Kalman para estimarlo tal y como se hizo en [17], pero tardaría mucho tiempo en converger (mínimo 15 segundos según indican) por lo que no sería del todo viable en nuestro caso.

La opción elegida inicialmente es la de suponer que los puntos se encuentran en el suelo y que se conoce la altura real del UAV en el instante tras el despegue. Esto puede hacerse ayudándose de las proporciones de la plataforma de despegue en las imágenes entrantes y así poder saber con bastante precisión la altura del UAV en el instante inicial

Todos los pasos anteriores han proporcionado información sobre las imágenes, no sobre la cámara ni el UAV. Tanto las matrices de rotación como las traslaciones eran entre imágenes como ya se comentó, no entre posiciones de la cámara, así que ahora toca resolver la pregunta sobre cómo obtener el cambio de posición y de orientación de la cámara y del UAV a partir de esos datos.

3.5 Trasvase de información de la imagen a la cámara

Anteriormente se comentó que los resultados de la homografía eran las modificaciones con respecto a las imágenes, ahora toca hacernse la pregunta de si estas modificaciones entre imágenes se traducen de manera directa a modificaciones en la posición de la cámara. Para ello, lo más sencillo es comprobar cómo varían ambas para un ejemplo sencillo:

1. Tenemos una imagen con una flecha apuntando hacia arriba.

2. Giramos 90º la imagen en torno al eje Z de la cámara, es decir, la imagen sufre una rotación de 90º en el sentido de las agujas del reloj (recordemos que el eje Z apunta hacia la imagen desde la cámara y no al revés). Esto implica que la flecha que inicialmente apuntaba hacia arriba ahora apunta hacia la derecha en la imagen.

3. ¿Qué cambio de la orientación de la cámara tendría ese mismo efecto en la imagen final? Pues si giramos la cámara 90º en torno a ese mismo eje Z… ¡La flecha apuntaría hacia la izquierda! En cambio, si giramos -90º, apuntaría hacia la derecha, por lo que debe de girar el ángulo opuesto al de la imagen.

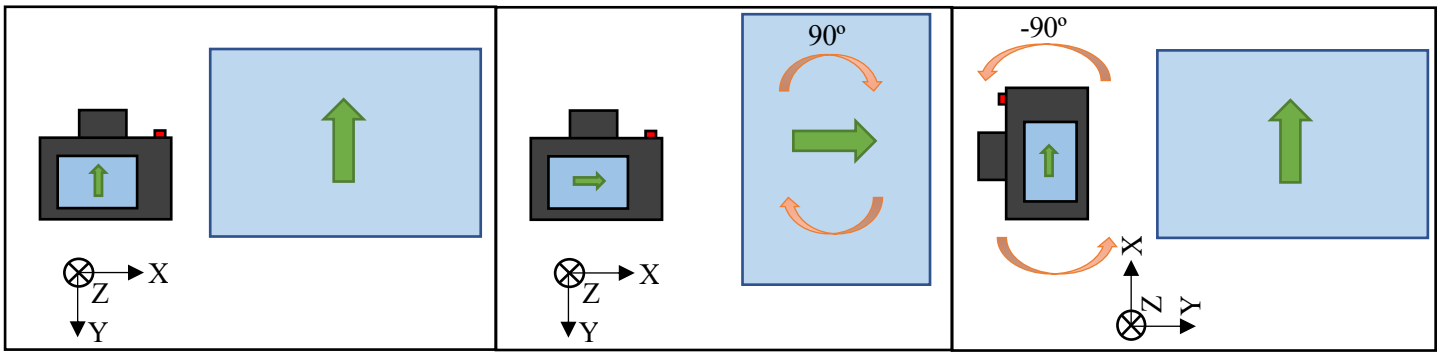

Ilustración 10. De izquierda a derecha, cada uno de los pasos comentados anteriormente en la analogía.

Esta simple prueba puede generalizarse de tal forma que la rotación que hay que aplicar a la cámara es la inversa de aquella que experimenta la imagen. Por lo que si la homografía proporciona una matriz R entre imágenes, la matriz entre orientaciones de la cámara será la inversa de R. Aunque el ejemplo se haya realizado para rotaciones, se puede realizar igualmente para traslaciones. Los valores de traslación proporcionados por la homografía son los opuestos de las traslaciones de la cámara, es decir, si un objeto se mueve hacia la derecha en la imagen (suponiendo que este esté realmente quieto) significa que la cámara se está desplazando hacia la izquierda), es decir, la traslación de la cámara será la opuesta a la obtenida.

3.5.1 Cambio de base de UAV a cámara y viceversa

A continuación, vamos a ver cómo relacionar los ejes propios del UAV y los de la cámara. Para ello, se ha recurrido a las imágenes de la simulación y la documentación de los propios programas.

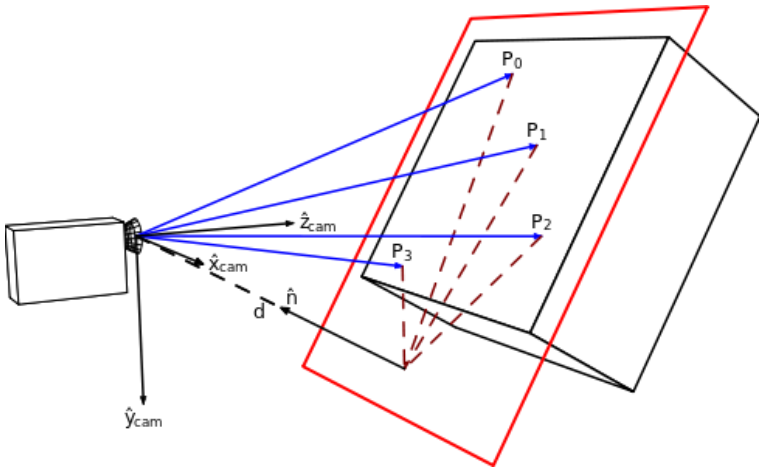

Ilustración 11. Sistema de referencia de una cámara.

Los ejes del UAV pueden verse en Gazebo seleccionando en la barra de herramientas → *view* → *link frames*. Estos ejes coinciden con aquellos especificados por la norma REP 103 [29] ya que siguen el convenio ENU, en el que el eje Z apunta hacia arriba, el eje X hacia el Este y el eje Y hacia el norte. Por otro lado, la cámara toma también los ejes que la norma específica para cámaras y que se muestran en la Ilustración 11 (*Homography-transl.svg: Per Rosengrenderivative work: Appoose / CC BY (https://creativecommons.org/licenses/by/3.0)*), por lo que el eje Z apunta hacia delante, el eje X a la derecha de la cámara y el eje Y hacia debajo de la cámara. Ambos ejes pueden verse representados en la Ilustración 12**.**

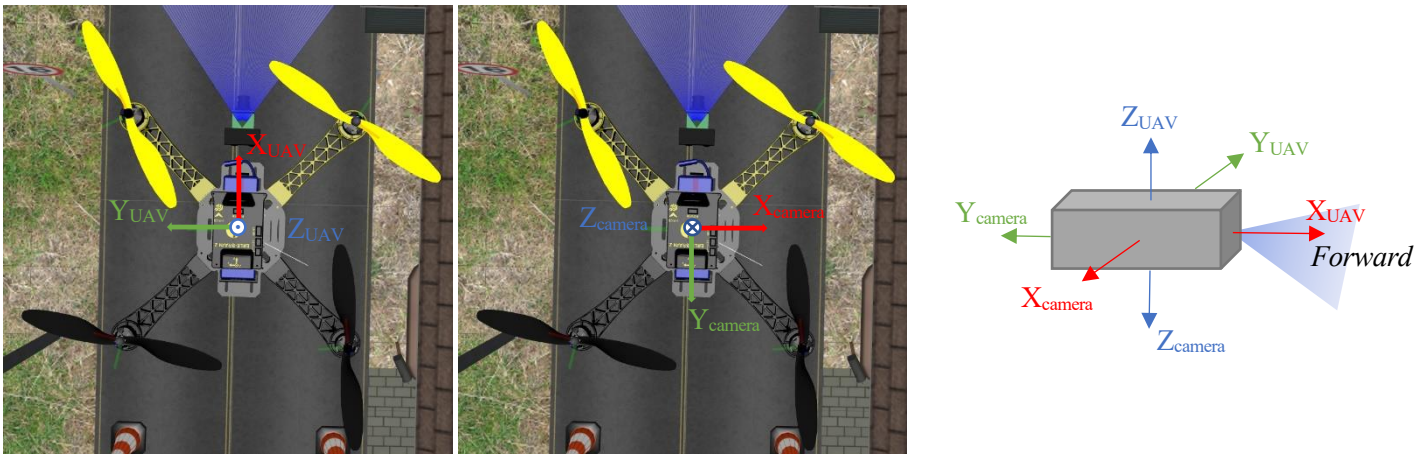

Ilustración 12. Ejes de referencia del UAV y de la cámara. Las dos primeras imágenes los muestran sobre una vista cenital del UAV. La última imagen muestra un esquema en 3 dimensiones de dichos ejes.

Basándose en esta información recogida, se deducen los giros que hay que realizar para pasar de los ejes del UAV a los de la cámara. Una forma de hacerlo es rotando 90º en Z y sobre el nuevo eje Y', rotar 180º. Como se actúa sobre ejes móviles esto se traduce en:

$$Rot_{UAV \rightarrow camera} = Rot_z(\pi/2) \cdot Rot_y(\pi)$$

De igual forma, si se quiere pasar de los ejes de la cámara a los ejes del UAV, habría que realizar otra rotación. En este caso ambas coinciden y valdría rotar primero en Z_{camera} 90º y, después, 180º en torno al nuevo eje Y'_{camera}, por lo que:

$$Rot_{camera \rightarrow UAV} = Rot_z(\pi/2) \cdot Rot_y(\pi)$$

Es decir, para este caso las matrices de rotación son la misma ya que ella misma es su propia inversa.

3.5.2 Ángulos de Euler a partir de una matriz de rotación

Una vez obtenida la matriz de rotación, habrá que extraer de ella los giros del UAV: *yaw, roll* y *pitch*. Estos ángulos son los giros sobre los ejes móviles de las rotaciones en torno a los tres ejes, siendo la rotación en Z la primera aplicada, después en Y' y, por último, en X''. También puede verse en sentido contrario como el giro en ejes fijos primero sobre X, luego sobre Y y por último sobre Z:

$$R = R_z(\varphi) \cdot R_y(\theta) \cdot R_x(\psi) = \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix}$$

Sin embargo, dicha descomposición no es única tal y como se explica en [18], [19] y [20] y, por tanto, habrá que realizar alguna suposición. Partiendo de ese artículo anteriormente citado, se obtienen

una serie de fórmulas que permiten obtener los ya citados ángulos.

$$\theta_1 = -\sin^{-1}(R_{31})$$
$$\theta_2 = \pi - \theta_1$$
$$\phi_1 = atan2\left(\frac{R_{21}}{\cos(\theta_1)}, \frac{R_{11}}{\cos(\theta_1)}\right)$$
$$\phi_2 = atan2\left(\frac{R_{21}}{\cos(\theta_2)}, \frac{R_{11}}{\cos(\theta_2)}\right)$$
$$\psi_1 = atan2\left(\frac{R_{32}}{\cos(\theta_1)}, \frac{R_{33}}{\cos(\theta_1)}\right)$$
$$\psi_2 = atan2\left(\frac{R_{32}}{\cos(\theta_2)}, \frac{R_{33}}{\cos(\theta_2)}\right)$$

Se tienen múltiples soluciones de entre las cuales es necesario escoger aquella que represente a nuestro UAV. La determinación de la solución real se consigue analizando el significado de cada ecuación y comparándolo con las condiciones de funcionamiento reales. En la realidad, se compararán fotogramas que están muy próximos en el tiempo, por lo que es más que razonable pensar que no le va a dar tiempo a girar 90º cada, como máximo, 200 ms. Esta limitación de movimiento permite eliminar una de las dos soluciones obtenidas y quedarse con una única terna de ángulos. Pero, pensando un poco más, se da uno cuenta de que en condiciones normales el dron no debería tener tampoco un *pitch* mayor de 90º, por lo que, como *asin()* devuelve valores entre $-\pi/2$ y $\pi/2$, no tiene sentido que la solución sea su complementario y, en consecuencia, se elegirían los ángulos con subíndice 1.

3.5.3 Resultado final de la posición de la cámara

Uniendo los cambios en los marcos de referencia y la información de la homografía se obtiene que, usando como marco de referencia aquel del dron, la rotación del mismo entre dos fotogramas se calcularía pasando primero a coordenadas de la cámara, aplicando posteriormente la rotación de la cámara y, finalmente, volviendo a las coordenadas de UAV:

$$Rot_{UAV,1\to2} = Rot_{UAV\to camera} \cdot \left(R_{homography,1\to2}\right)^{-1} \cdot Rot_{camera\to UAV}$$

Donde $R_{homography}$ es la matriz obtenida con la homografía y $Rot_{UAV,1\to2}$ es la matriz que indica el giro de los ejes del UAV de la posición 1 a la 2.

Ahora bien, no pueden obtenerse los valores de los ángulos directamente de esa matriz porque puede ser que los ejes 1 del UAV no coincidan con los iniciales. Recuérdese que lo que realmente se busca es saber los ángulos del dron con respecto a la posición inicial de reposo donde todos son nulos. Supóngase que en vez de pasar del fotograma 1 al 2, pasase del 4 al 5, los ejes 4 tienen una orientación diferente de los ejes iniciales (ejes 0) por lo que cualquier supuesto ángulo de Euler entre 4 y 5 calculado directamente de $R_{homography,4\to5}$ será una combinación de los iniciales y, por tanto, no podrá sumarse directamente. Es decir, para poder calcular los ángulos de Euler del UAV reales, habrá que saber la variación con respecto a la orientación inicial y no extraerlos directamente de la variación anterior (aunque se podría hacer desarrollando otra fórmula diferente). Si la cámara fuera la única fuente de información, entonces habría que acumular todas las rotaciones que va realizando la cámara:

$$R_{UAV,0\to n} = Rot_{UAV\to camera} \cdot (R^{-1}_{homography,0\to 1} \cdot \ldots \cdot R^{-1}_{homography,n-1\to n}) \cdot Rot_{camera\to UAV}$$

Donde la matriz $R_{UAV,0\to n}$ sería aquella de la que se podrían extraer los ángulos de Euler que llevan a la posición n.

$$R_{UAV,0\to n} = Rot_z(yaw) \cdot Rot_y(pitch) \cdot Rot_x(roll) = R_{UAV,0\to 1} \cdot \ldots \cdot R_{UAV,n-1\to n}$$

Para los desplazamientos del fotograma 1 al fotograma 2, habría que tener en cuenta que la traslación está expresada en la base del fotograma 1 pero con la dirección del fotograma 2 al 1, por lo que si se quiere la traslación en la base del fotograma 2, habrá que multiplicar adicionalmente por la inversa de la rotación y por -1 para pasar que apunte de la posición 1 a la 2. Esto se ve mejor en 2D donde, si los ejes giran un ángulo α, para pasar de la base inicial a las coordenadas en la nueva base esto equivaldría a que los ejes iniciales estuvieran quietos y el vector girara un ángulo $-\alpha$. Se obtienen así las traslaciones en la base 1 del UAV y en la base 2 del UAV:

$$t_{UAV,1,1\to 2} = -Rot_{camera\to UAV} \cdot t_{homography,1\to 2} \cdot d^*$$
$$t_{UAV,2,1\to 2} = -Rot_{camera\to UAV} \cdot R_{homography,1\to 2} \cdot t_{homography,1\to 2} \cdot d^*$$

Donde d* es el factor de escala que se comentó en el apartado 2.4.4 y $t_{UAV,1,1\to 2}$ es la traslación del instante 1 al 2 en la base del UAV en el momento 1.

Igualmente, si la cámara fuera la única fuente de información de la posición, para estimarla habría que sumar todos los desplazamientos, desde el inicio hasta el final, expresados en una misma base. Lo más lógico es usar como base la posición inicial ($t_{UAV,0,0\to n}$). En cada instante, la nueva posición será la posición anterior más la nueva traslación expresada en la nueva base:

$$t_{UAV,0,1\to n} = t_{UAV,0,0\to n-1} + \left(R_{UAV,0\to n-1}\right)^{-1} \cdot t_{UAV,n-1,n-1\to n}$$

Donde el último término puede expresarse según se ha visto anteriormente como:

$$t_{UAV,0,n-1\to n} = -Rot_{UAV\to camera} \cdot \left(R_{\text{hom}\_acum,n-1}\right) \cdot t_{homography,n-1\to n} \cdot d^*$$

Siendo $R_{hom\_acum,n-1} = R^{-1}_{homography,0\to 1} \cdot \ldots \cdot R^{-1}_{homography,n-2\to n-1}$ la matriz que indica el giro desde la orientación inicial de la cámara hasta la orientación n-1 de la misma.

4 ESTIMACIÓN DE LA POSICIÓN CON IMÁGENES POR MARCADORES ARTIFICIALES

En el capítulo anterior se abordó una manera de conseguir una estimación de la posición a través de las imágenes proporcionadas por la cámara sin ningún elemento concreto de referencia. Ahora bien, también se puede plantear una solución más limitada en el espacio basada en el uso de objetos en la imagen cuyas dimensiones y forma se conozcan previamente, es decir, mediante el uso de marcadores artificiales. De esta forma, calculando la deformación en la imagen de dichos objetos, se podrá estimar cuál es la posición de la cámara con respecto a este. Utilizando este método, tendremos la limitación de que estos objetos tengan que estar siempre en el campo de visión de visión del dron, pero, a cambio, no será necesario realizar ningún tipo de suposición de factor de escala como se ha comentado para el apartado anterior. Así, en este capítulo se tratarán los siguientes temas:

- Marcadores ArUCo y uso.
- Integración de marcadores ArUCo en Gazebo.
- Posibilidades de los marcadores ArUCo dentro de la problemática de este TFG.

4.1 Marcadores ArUCo

Los marcadores ArUCo son un tipo de marcadores desarrollados inicialmente en la universidad de Córdoba por Rafael Salinas y Sergio Garrido [21] y tienen un especial interés en la estimación de la posición a través de imágenes ya que nos permiten utilizar el conocimiento de su tamaño para estimar medidas.

La apariencia de estos marcadores es la de las imágenes adjuntas a continuación en la Ilustración 13. Tal y como se ve, tienen forma cuadrada con un borde negro y se distinguen los unos de los otros por la combinación de cuadrados blancos y negros en su interior. Este número de cuadrados en el interior puede variar desde 4x4 hasta 5x5, 6x6 o 7x7 según el diccionario elegido.

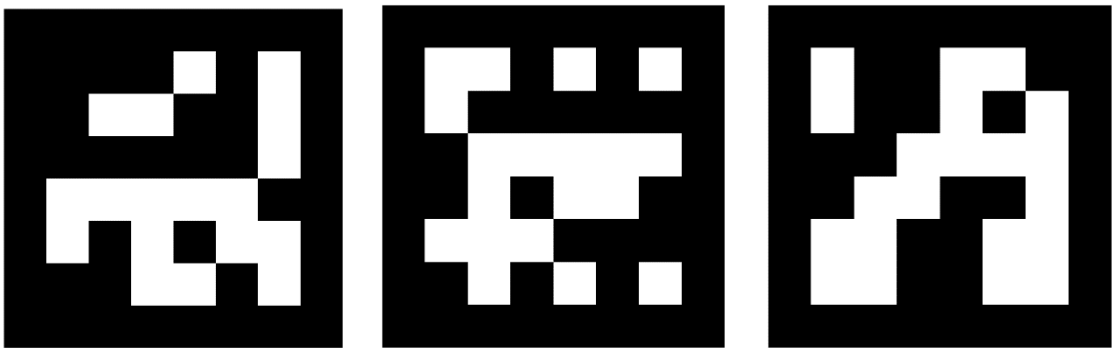

Ilustración 13. Ejemplo de marcadores ArUCo de 6x6 bits.

El uso de estos marcadores está extendido, lo que facilita la búsqueda de información en internet y cuentan además con su propio módulo dentro de OpenCV. Este módulo incorpora muchas funciones, pero las que serán de mayor interés son las de detección de marcadores en la imagen y la de estimación de la posición de la cámara con respecto a los marcadores.

Haciendo referencia a esto último, la detección de los marcadores en la imagen puede conseguirse con la función *cv::aruco::detectMarkers*. Esta se encarga de pasar filtros a la imagen para obtener los contornos y quedarse con las figuras que tengan 4 vértices (posibles cuadrados deformados); posteriormente, elimina la perspectiva de los cuadriláteros y busca si hay cuadrados blancos en el interior que representen la codificación interna de dicho marcador. Si encuentra algún candidato con codificación, entonces habrá encontrado un marcador.

Por otro lado, la estimación de la posición se realiza sabiendo las medidas reales del marcador. Sabiendo las medidas reales, la proyección en la cámara y los parámetros de esta, se puede estimar cual es la posición del marcador con respecto a la cámara (siendo los ejes de esta aquellos comentados anteriormente, es decir, los de la norma REP-103). El resultado obtenido no es la posición de la cámara con respecto al marcador, sino la posición del marcador con respecto a la cámara. Igualmente sucede con la rotación, esta es la que convierte los ejes de la cámara en los del marcador. De igual forma, si hubiera varios marcadores, proporcionaría la posición de los diferentes marcadores con respecto a la cámara. Esto lleva a plantear el problema de pasar de los valores obtenidos a los deseados (posición de la cámara con respecto al marcador). Este problema se soluciona haciendo un cambio de coordenadas que, en teoría es simple, pero que en la práctica puede ser problemático:

$$t_{marker,marker\rightarrow camera} = -(R_{camera\rightarrow marker})^{-1} \cdot t_{camera,camera\rightarrow marker}$$

Donde $R_{camera\rightarrow marker}$ es la rotación proporcionada por el algoritmo para pasar de los ejes de la cámara a los del marcador y $t_{camera,camera\rightarrow marker}$ es la traslación del algoritmo que debe ir en sentido contrario (del marcador a la cámara) y cambiada de referencia.

También cabe comentar que la rotación proporcionada no es directamente aquella del UAV desde la posición inicial, por lo que habrá que multiplicar dicha rotación por $Rot_{camera\rightarrow UAV}$ para obtener la rotación real del dron.

$$R_{marker\rightarrow UAV} = (R_{camera\rightarrow marker})^{-1} \cdot Rot_{camera\rightarrow UAV}$$

Para el caso de que haya varios marcadores habrá que realizar una media de las rotaciones proporcionadas por cada uno de ellos y tener en cuenta los desplazamientos en la posición de los marcadores entre sí para obtener una estimación de la traslación global de la cámara congruente.

4.2 Marcadores ArUCo en Gazebo

Gazebo incorpora un marcador ArUCo de fábrica, sin embargo, ¿qué pasa si se quieren utilizar más marcadores? Pues la solución pasa por crear nuestro propio modelo en Gazebo del marcador que se quiera utilizar. Este proceso puede simplificarse al tener ya un marcador hecho, por lo que servirá de ejemplo para crear los demás.

En primer lugar, se eligen los marcadores a utilizar. Una opción para ello sería ir a esta página web [https://chev.me/arucogen/] e indicar el identificador, tamaño y diccionario deseado. Para simplificar el proceso y no tener que trabajar con un fichero .svg, se puede hacer una captura de pantalla del marcador. El objetivo final sería una imagen cuadrada y en la que los tamaños relativos del borde y cuadrados internos se mantengan, por ello, el recorte no debe intentar ser perfecto ya que podrían eliminarse alguna de las primeras filas o columnas del marcador de manera no intencionada. En consecuencia, es recomendable dejar un margen entre el recorte y el marcador y, posteriormente, procesar la imagen en Matlab u Octave para quedarse únicamente con el marcador. Además, hay un pequeño detalle de importancia, el sistema de detección de marcadores presenta problemas en el caso en que no haya un buen contraste entre el límite negro del bode del marcador y el fondo. Por ello, a este marcador ya recortado se le añade un borde blanco, asegurándonse así que mejora la detección. Este programa de recorte y adición del marco se encuentra también en el repositorio creado y se llama screenshot2marker.m .

Ahora, yendo a la carpeta que guarda los modelos de Gazebo *~/.gazebo/models* (con Ctrl+H aparecen las carpetas ocultas, es decir, *.gazebo*) y abriendo la carpeta Aruco0 (aquella relativa al modelo Aruco de Gazebo), se ve que hay algunas carpetas más y archivos. Si se quiere crear un nuevo modelo con la imagen obtenida anteriormente habrá que copiar y pegar la carpeta de Aruco0, cambiarle el nombre a ArucoMarker1 por ejemplo y, abriendo los archivos que no tengan ~ en su nombre, cambiar todos los Aruco0 por ArucoMarker1 (incluidos nombres). Finalmente, en *material/textures/* estará la imagen del marcador en .png, esta debe cambiarse por la que se ha obtenido anteriormente, la cual debe tener como nombre ArucoMarker1.png. En nuestro caso se han realizado los modelos para un diccionario de 6x6 y se han tomado los valores: 1, 2, 3, 4, 5, 6, 8 y 23. Estos modelos se han introducido en un mundo que se ha creado a partir de otros archivos de Gazebo y que viene explicado en el anexo sobre instalación de Gazebo.

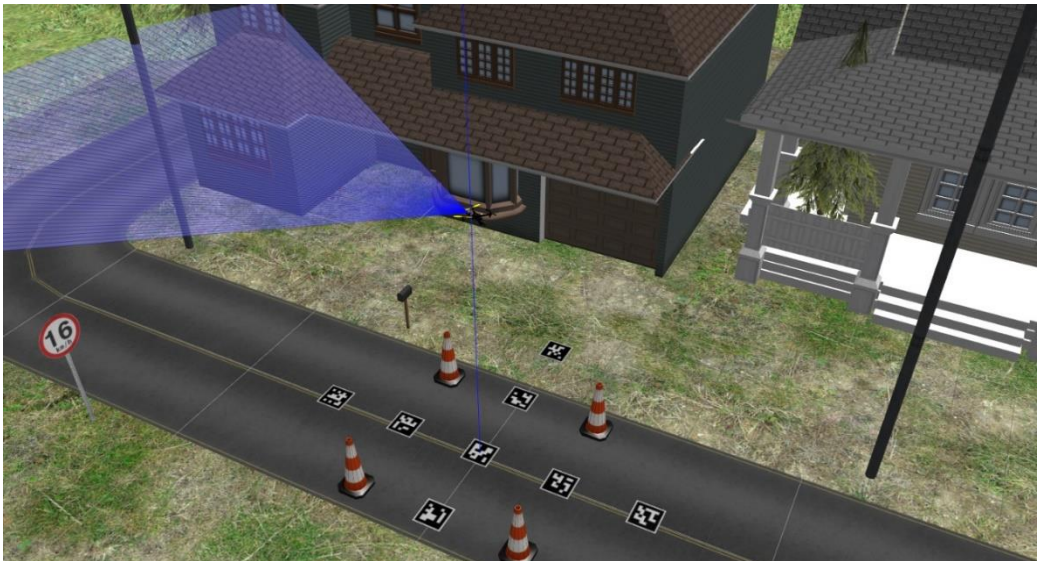

Ilustración 14. Imagen del mundo donde se simula el dron con 9 marcadores ArUCo en el suelo.

Si se abre ahora Gazebo, deberían poderse utilizar estos marcadores que tendrán un ancho de 0.5m. Hay que recordar sin embargo que se ha añadido un borde, por lo que el tamaño del marcador real se ha reducido. Para resolver esto, solo habría que ver los píxeles que tiene nuestra imagen del marcador (en mi caso 836x836), restarle los píxeles del borde cuyo tamaño elegimos en la función de Matlab (40 por lado, en total 80) y esos serán los píxeles de la imagen que ocupan el marcador (756x756). Por proporcionalidad, se puede sacar que sin 0.5m son 836px, entonces los 756px ocupan en total 0.45215m. Este será el parámetro de longitud del marcador que habrá que pasarle a *estimatePoseSingleMarkers()*.

4.3 Uso de marcadores ArUCo en este proyecto

Los marcadores ArUCo se utilizarán en este proyecto como un medio para la determinación de la posición de la cámara mediante marcadores artificiales. Sin embargo, en el capítulo anterior se dejó en el tintero la estimación de la altura inicial del dron. Este problema podría resolverse, por ejemplo, usando un marcador ArUCo justo en la base de despegue del dron, de tal forma que, a la hora de despegar, se puedan obtener datos válidos ya que, como resulta obvio, el método basado en marcadores naturales no podrá aportar una estimación durante el principio del despegue ya que las imágenes serán colores prácticamente planos y negros debido a la baja distancia entre cámara y suelo. Puesto que el método comentado en el capítulo 3 va sumando los desplazamientos calculados, al no detectar puntos durante un periodo de tiempo, no detectará esos desplazamientos existentes, por lo que no sería viable su uso durante el despegue.

De esta forma, en los apartados de simulación se procederá a implementar los marcadores en un primer capítulo destinado a marcadores artificiales para así realizar una comparativa y entender su comportamiento en un entorno simulado. Mientras que, en un segundo capítulo, se utilizará la mejor versión obtenida del método de marcadores artificiales para establecer las condiciones iniciales del dron, es decir, para obtener cual será la posición y ángulos iniciales justo en el momento en el que termine el despegue y se empiece a utilizar el método basado en marcadores naturales. En resumen, se usarán marcadores artificiales para:

- Comparar métodos basados en marcadores artificiales, por lo que no pueden perderse de vista.

- Establecer la posición y ángulos iniciales del UAV, a partir de los cuales empezará a funcionar el método de estimación de posición basado en marcadores naturales, por lo que no habría limitaciones de visionado de objetos (más allá de la necesidad de que las imágenes obtenidas por la cámara no sean colores planos).

5 CREACIÓN DE PROGRAMAS EN ROS

Como ya se indicó al principio, se va a utilizar ROS para poder trabajar con los datos que nos envían los sensores del dron. En nuestro caso necesitaremos trabajar con la información proveniente de la cámara y la IMU. En este capítulo se explica de manera breve en qué consiste ROS, sus principales funciones y algunos detalles propios de la implementación de algunos de los programas realizados para las simulaciones del ErleCopter y que se encuentran disponibles en el repositorio indicado en el primer capítulo. No se pretende que tenga un carácter global, sino orientado a entender los códigos y tareas a realizar en nuestro problema concreto, por ello se recomienda ver [6] y [22] para más información. En el anexo D se exponen algunos de los códigos realizados a lo largo del proyecto.

5.1 Conceptos generales

ROS es un marco para el desarrollo software de robots cuyo objetivo es simplificar la programación de la gran variedad de tareas que diferentes tipos de robots pueden realizar basándose en una arquitectura de grafos. Para entender mejor esta estructura hay que conocer los 3 conceptos en los que se basa:

- *Nodes* : Es cada uno de los nodos que forman el grafo en el que se basa ROS. Estos nodos no son más que procesos que se comunican con el resto de tareas a través de *topics* por lo que se envían mensajes y se sincronizan con *services.*

- *Topics*: Son una especie de buses imaginarios a través de los cuales intercambian información los nodos. Es decir, los nodos pueden enviar información (actúan como *publishers*) a través de un *topic* a varios nodos que esperan dicha información (actúan como *subscribers*). Esta información enviada, es decir, los mensajes, deben tener un tipo determinado fijado previamente.

- *Services*: Son una forma de sincronización entre nodos en las que se envía una petición y se recibe una respuesta.

Este grafo lo crea un proceso *master* que es el que se encarga a su vez de crear los nodos e interconectarlos, pero, una vez creados estos, no interactúa con ellos durante sus comunicaciones.

Como este proyecto está enfocado especialmente al tratamiento de la información procedente de sensores (IMU y cámara) no se va a recurrir a los servicios ya que no hará falta, en un principio, realizar ningún tipo de sincronización. Es decir, bastará con manejar la información procedente de los *topics* en los que los sensores irán trasvasando sus datos.

5.2 Funciones y comandos habituales y de interés

En este apartado se van a subrayar algunas funciones y comandos de ROS que se van a utilizar y que resulta conveniente conocer. Para ello se van a separar en varios bloques según estén relacionadas con: compilación, *topics*, nodos y ejecución. Nótese que para utilizar estas funciones en *scripts* será necesario incluir "ros/ros.h" en el fichero y que, antes de lanzar el primer nodo, habrá que ejecutar en algún otro terminal el *master*, lo que puede hacerse con el comando *roscore* (aunque si se ejecutan los nodos desde un fichero .launch no sería necesario).

5.2.1 Nodos

Cada nodo va a realizar una serie de tareas indicadas en su código. Para crear un ejecutable que haga de nodo, será necesario que en el main se utilice *ros::init(argc,argv,"nombredelnodo")* antes que cualquier otra función de ROS. Los argumentos argc y argv son las entradas típicas del main, siendo argc el número de argumentos y argv el vector de cadenas que guarda dichos argumentos. Posteriormente, se declara el manejador del nodo usando *ros::NodeHandle nombre_manej_nodo* pudiendo referirnos al nodo a partir de ahora con ese nombre indicado (lo cual será necesario para la subscripción a los *topics*).

5.2.2 Topics

Los *topics* son una manera de enviar información de un nodo a otro. En este sentido se puede enviar o recibir un mensaje, teniendo así que escribir una serie de líneas de código para suscribirse a un *topic* y otra diferente para publicar en él. A continuación se indican estas órdenes a la hora de escribir código y algunos comandos de terminal útiles para depurar el código relacionado.

5.2.2.1 Subscripción

Para subscribirte a un *topic* será necesario utilizar un objeto *ros::Subscriber* y utilizar la función *subscribe* del manejador del nodo declarado anteriormente.

```
ros::Subscriber nombre_del_subscriber = nombre_del_nodo.subscribe
("topic_al_que_se_subscribe", tamaño_cola, función_a_ejecutar)
```

Donde la *función_a_ejecutar* se ha debido declarar previamente y será aquella que se ejecute cada vez que lleguen mensajes por el *topic,* el cual va entrecomillado porque tenemos que pasarlo como cadena. El tamaño de la cola indica el número máximo de mensajes que puede haber esperando a ser utilizados por la función a ejecutar de tal forma que, conforme vayan llegando más, se vayan borrando los más antiguos.

A continuación, se utiliza la función *ros::spin()*, la cual entra en un bucle infinito hasta que cancelen el nodo. Conforme le van llegando mensajes, el nodo ejecuta las funciones que se le han indicado (que se suelen llamar *callbacks).* Existen también otras variantes como *ros::spinOnce()* que se ejecutan una vez.

La forma de esa función *callback* a la que se llama de manera recurrente suele ser:

```
void chatterCallback(const Tipo_de_mensaje::ConstPtr& msg)
```

Donde *msg* es un puntero al mensaje que ha llegado por el *topic* y que por lo general tiene forma de estructura, por lo que para poder utilizar los datos que contiene sería necesario utilizar -> en la función.

En el caso en el que el mensaje recibido sea una imagen y queramos operar con ella como si fuera un tipo *Mat* (el tipo que se utiliza para trabajar en OpenCV con matrices), será necesario utilizar una función adicional que sirva de puente entre ROS y OpenCV.

```
Mat img = cv_bridge::toCvShare(msg, "mono8")->image;
```

Donde *img* sería la variable de tipo *Mat* que guarda la imagen tomada pasada a blanco y negro. Si se quisiera la imagen en color habría que sustituir *mono8* por *bgr8* (para 8 bits por canal).

Esta función se incluye dentro de <image_transport/image_transport.h> y <cv_bridge/cv_bridge.h>.

5.2.2.2 Publicación

En el caso en el que queramos enviar un mensaje, habrá que crear primero un publicador, esto se consigue con la línea de código *ros::Publisher nombre_publisher* y asociar a ese publicador un nodo que publique (a través de su manejador), un *topic* donde publicarlo, el tipo de mensaje y un tamaño de la cola. Esto se puede sintetizar en:

```
ros::Publisher nombre_publisher = nombre_manej_nodo.advertise
<tipo_mensaje>("topic_al_que_se_publica", tamaño_de_la_cola);
```

De esta forma, cada vez que queramos publicar algo por el *topic* simplemente habrá que crear una variable (*mensaje_a_enviar*) con el tipo especificado, rellenarla, y publicarla con la siguiente línea:

```
nombre_publisher.publish(mensaje_a_enviar);
```

5.2.2.3 Comandos de terminal útiles

Una buena forma de debuguear los mensajes que se mandan o reciben es usando comandos de terminal. En esta línea se pueden usar comandos como:

- *rostopic list* para mostrar los *topics* existentes.
- *rostopic echo nombre_del_topic* para mostrar el contenido de los mensajes que se están enviando por *nombre_del_topic*.
- *rostopic type* si quieres saber qué tipo de mensaje se utiliza.
- *rostopic show tipo_de_mensaje* para ver qué elementos componen la estructura del mensaje.

Adicionalmente existen algunos paquetes que permiten representar las relaciones en términos de los *topics* que conectan los diferentes nodos como por ejemplo *rqt_graph: rosrun rqt_graph rqt_graph*

5.2.3 Compilación

Existen diferentes formas de organizar los archivos que vas a compilar. La explicada aquí es la que se ha utilizado para los ficheros del repositorio y consiste en utilizar un fichero *CMakeLists.txt* en el que se especifican las dependencias de los programas a compilar. Si se quisiera agregar un programa utilizando las librerías de OpenCV y ROS habituales, bastaría con escribir el nombre de tu archivo en el mismo formato que en el documento del repositorio, es decir, añadir el ejecutable, sus dependencias y las librerías necesarias. También podría cambiarse el nombre del proyecto según se desee. Lo ideal sería guardar este archivo *CMakeLists.txt* en una carpeta llamada *src* junto a los archivos a compilar que se compilarán escribiendo por terminal *catkin_make* (suponiendo que el terminal está en la carpeta inmediatamente superior a *src*). Esto generará un par más de carpetas: *build* y *devel*.

5.2.4 Ejecución

Una vez compilados, existen varias formas de ejecutar esos programas dependiendo de si se quieren ejecutar varios a la vez o solamente uno. Sin embargo, antes que nada habrá que ejecutar la siguiente línea (si estamos en la carpeta inmediatamente superior a *src* en el terminal):

```
source devel/setup.bash
```

- Si quieres ejecutar solo uno, vale con utilizar la siguiente estructura por terminal:

```
rosrun nombre_proyecto nombre_ejecutable
```

- Si se quiere en cambio ejecutar varios a la vez, habría que crear primer un archivo .launch. Este archivo también tiene la posibilidad de pasarle parámetros (que pueden expresarse como ficheros) a esos ejecutables a través de *rosparam*. Un fichero tipo .launch que tiene un ejecutable al que se le pasa un archivo como argumento tiene la siguiente forma:

```
<launch>
  <node pkg="nombre_proyecto" type="nombre_ejecutable" name="nombre_nodo"/>
  <node pkg="nombre_proyecto2" type="nombre_ejecutable2" name="nombre_nodo2">
<rosparam command="load" file="$(find nombre_proyecto)/fichero_YAML.yaml" />
  </node>
<launch/>
```

Para ejecutar estos archivos, simplemente habría que indicar lo siguiente por terminal:

```
roslaunch nombre_proyecto nombre_launch.launch
```

Además, en este caso no habría hecho falta haber usado *roscore* antes en otro terminal.

5.3 Rosbag

Este último elemento es una herramienta de ROS que es muy útil para almacenar información sobre las distintas simulaciones que se han realizado y que puede facilitar enormemente el trabajo en una situación real en la que se tenga un tiempo limitado para la realización de pruebas. Rosbag es una herramienta que permite almacenar los mensajes que se envían durante un periodo de tiempo, pudiendo reproducirse posteriormente el comportamiento grabado. De esta forma, en nuestro caso se puede realizar una simulación y, con esas imágenes y valores de la IMU en forma de mensajes, comprobar luego cómo funcionan los algoritmos sin tener que abrir Gazebo de nuevo. Durante la reproducción, los mensajes se envían en los mismos periodos de tiempo aproximadamente tal y como indican los elementos *stamp* del elemento *header* de la estructura que forma el mensaje.

Para guardar todos los mensajes enviados se debe correr en un terminal:

```
rosbag record -a
```

Si se quisieran guardar solo algunos *topics*:

```
rosbag record -O nombre_archivo_bag nombre_topic_1 nombre_topic_n
```

Finalmente, si se quisieran reproducir dichos archivos, valdría con ejecutar la siguiente línea estando en la misma carpeta que el archivo:

```
rosbag play nombre_archivo.bag
```

Ojo, la reproducción no empezará hasta que se inicie el *master*, es decir, se haga *roscore* o se ejecute algún archivo .launch.

5.4 Robot_simulation

En el proyecto también se probará a utilizar el paquete de *robot_localization* el cual tiene varias funciones que se encargan de crear nodos que implementan un filtro de Kalman extendido (EKF) o un *unscented Kalman filter* (UKF) a tantas fuentes de información como se deseen. En nuestro caso se optará por utilizar un filtro de Kalman extendido como se detallará en el apartado correspondiente.

Para poder realizar esto, será necesario un fichero .launch con la misma estructura comentada anteriormente en la que se referencie a un fichero .yaml con los parámetros para la creación de ese filtro de Kalman. La escritura de este fichero .yaml es lo que puede requerir un mayor trabajo y posiblemente requiera de un proceso iterativo de reajuste de los parámetros, por ello, se recomienda mirar las referencias [28] y, en especial, el tutorial proporcionado en [27]. También puede recurrirse al archivo EKF_template.yaml que se encuentra en la capeta *params* del paquete robot_localization y que explica cada campo a rellenar. Sin embargo, aquí se comentarán por encima los aspectos más importantes:

- *frequency*: Frecuencia de cálculo de la salida del filtro de Kalman.
- *sensor_timeout*: Tiempo máximo en segundos que puede estar un sensor sin enviar información antes de considerarse averiado y dejar de fusionarse.
- *map_frame, odom_frame base_link_frame* y *world_frame*: Para entender estos conceptos se recomienda leer la norma REP 105 [30]. *Grosso modo*, puede decirse que sirven para indicar el marco de referencia de las medidas que estamos tomando y explicitar con qué nombres de nuestra simulación se corresponden. En esta línea el marco *base_link_frame* está unido al robot móvil mientras que los otros se refieren a sistemas globales (que pueden tener deriva temporal o no).

Para indicar las fuentes de medidas, este paquete permite cuatro tipos de mensajes: mensajes de tipo IMU (*geometry_msgs::imu*), mensajes con velocidades (*geometry_msgs::TwistWithCovarianceStamped*), mensajes con posiciones (*geometry_msgs::PoseWithCovarianceStamped*) y mensajes de tipo odometría que serían la combinación de los dos anteriores (*nav_msgs::Odometry*). Ojo, cuando se habla de posición, se refiere al equivalente en inglés como *pose*, es decir, se incluye tanto la ubicación en X, Y, Z como los ángulos que forma dicho sistema de referencia con respecto al original. Según el mensaje que sea, si se quiere configurar uno de tipo odometría habrá que usar odom0; si fuera *twist*, twist0 y, si hubiera varios del mismo tipo, odom1, odom2 … Para cada una de estas fuentes habrá que configurar:

- *Topic* que envía información de las medidas. Ejemplo:

  ```
  odom0: /Camera_estimation
  ```

- Medidas que queremos fusionar de las que se reciben por el *topic*. Para ello se realiza una matriz cuyas filas representan respectivamente: posición, orientación, velocidad lineal, velocidad angular y aceleración lineal. Si se quiere fusionar un elemento, simplemente se pone a verdadero. Ejemplo de fusión de velocidad angular y lineal:

```
odom0_config:  [false,  false,  false,
                false,  false,  false,
                true,  true,   true,
                true,  true,  true,
                false,  false,  false]
```

- Si se quiere que se tengan en cuenta los incrementos entre medidas en vez de las medidas en sí, se activa *odom_differential*. También existe *odom_relative* que avisa de que las medidas obtenidas por un sensor son relativas a la primera.
- Por último, también hay una cola que indica cuántos posibles mensajes vas a fusionar de una tirada.

Estos comandos anteriores valen tanto para *odomx* como para *twistx*, *posex* o *imux*, siendo la x el número de la fuente que le hemos dado.

Finalmente, habría que especificar las matrices de covarianza de ruido del proceso y la de la estimación inicial. Estos dos últimos parámetros serán los que hay que ajustar en cada situación particular para intentar obtener un comportamiento óptimo.

6 RESULTADOS EXPERIMENTALES ARUCO

En los capítulos anteriores se comentó el posible uso de marcadores absolutos artificiales y cómo funcionaban. En este, se llevará a cabo una serie de simulaciones en las que el UAV se desplazará entre varias posiciones teniendo visible en todo momento al marcador o los marcadores de referencia. Para ello, se utilizarán mundos creados como se indica en el anexo A y se incorporarán los marcadores necesarios que se crearon según el apartado 4.2.

En un principio se probará la configuración básica de un único marcador en el mismo punto de despegue y, a continuación, se irán realizando cambios ya sea añadiendo más marcadores o utilizando la información proveniente de la IMU con el objetivo de obtener mejores resultados.

Finalmente, se compararán las diferentes alternativas de posicionamiento simuladas y se detectarán los puntos débiles y limitaciones de los marcadores ArUCo.

6.1 Simulación para un solo marcador

Inicialmente, se ha probado a realizar la prueba de funcionamiento de la estimación de la posición del dron a partir de un único marcador como se ve en la Ilustración 15. Esta prueba se ha realizado con el programa *pure_marker.launch.* Este método se basa solamente en las estimaciones proporcionadas por los algoritmos que utiliza la librería ArUCo, sin incluir información de la IMU. Los resultados se muestran en las siguientes gráficas (Ilustración 16):

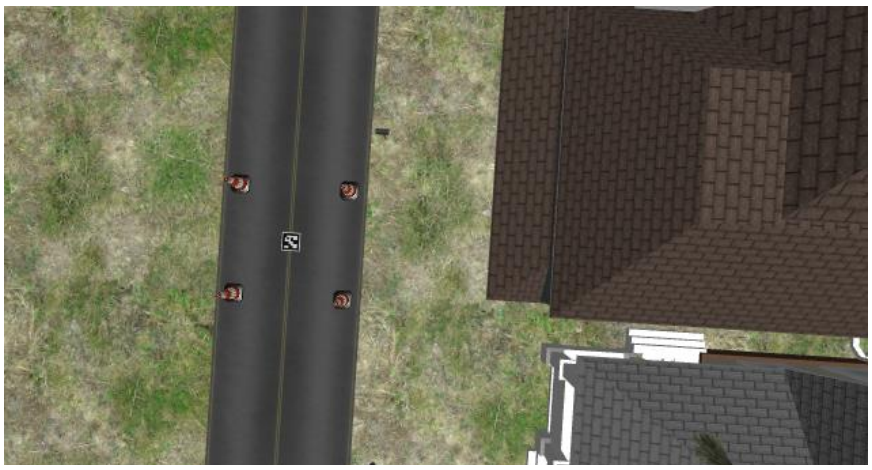

Ilustración 15. Imagen del mundo para la simulación de un marcador.

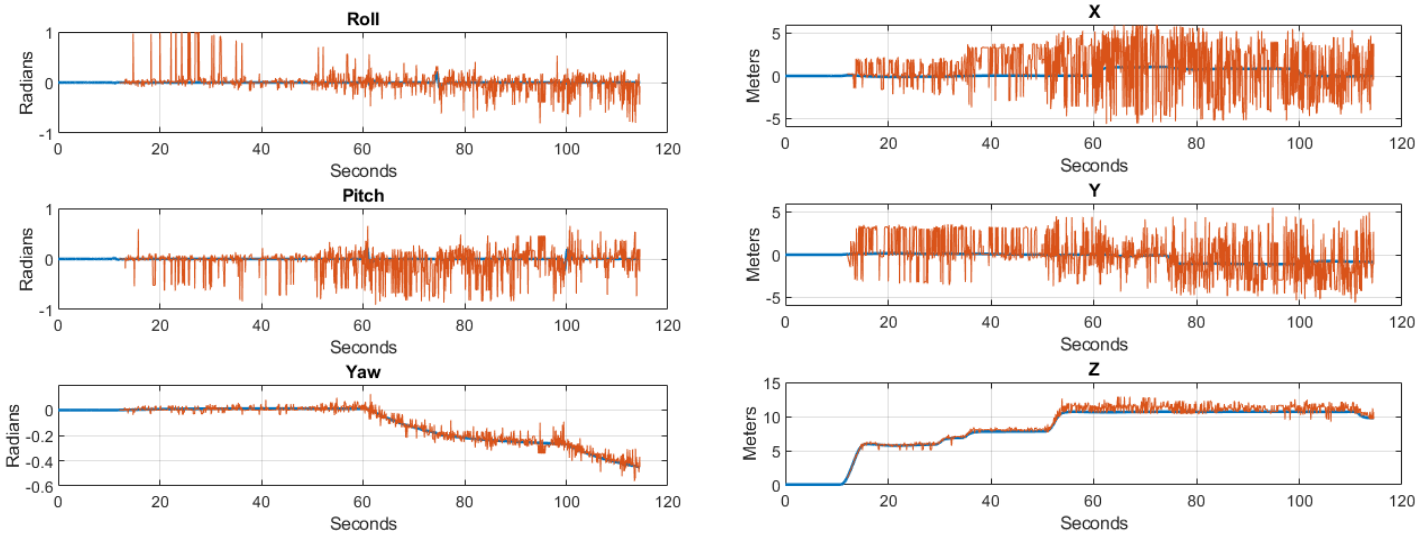

Ilustración 16. Resultados de la simulación para un marcador ArUCo. En rojo el valor estimado y en azul el valor real.

Viendo los resultados es evidente que hay algo que está funcionando mal, especialmente en el caso de la orientación, la cual parece tener un ruido que difumina cualquier posible valor real. Como el dron está prácticamente siempre en horizontal, a excepción de los momentos en los que se mueve, los valores de los ángulos de incidencia de la cámara con respecto al marcador son bastante pequeños. Según los artículos [23] y [24] los marcadores ArUCo presentan su peor comportamiento en la estimación de la orientación para ángulos de incidencia pequeños. Esto se debe a que un ángulo de incidencia nulo es una posición inestable en la que la proyección del marcador debería ser un cuadrado perfecto. En el momento en el que haya la más mínima desviación de esta proyección, esta se traduce en un cambio brusco de la estimación de la orientación. Aunque estos artículos obtengan valores del error bastante menores que los conseguidos aquí, cabe recalcar que su cámara presenta una resolución 10 veces mayor (2Mpx frente a 230kpx), lo cual puede afectar bastante en los resultados si las distancias entre la cámara y el marcador son suficientemente grandes como es nuestro caso. En general, estos investigadores recomiendan que los ángulos de incidencia sean en torno a 30º. Esto no es algo viable en nuestro caso si queremos usar el marcador como plataforma de despegue, sin embargo, hay otra alternativa ya que en los artículos también se indica una mejora del comportamiento cuando se usan diferentes marcadores.

6.2 Simulación para varios marcadores

Se van a realizar pruebas con diferente número de marcadores. Mientras más marcadores mejor debería ser nuestra estimación, así que se va a probar con 3 disposiciones diferentes: 2 marcadores, 4 marcadores y 8 marcadores adicionales al central, haciendo un total de 3, 5 y 9 marcadores totales. El programa que realiza esto es *pure_several_markers.launch*, el cual está en el repositorio creado.

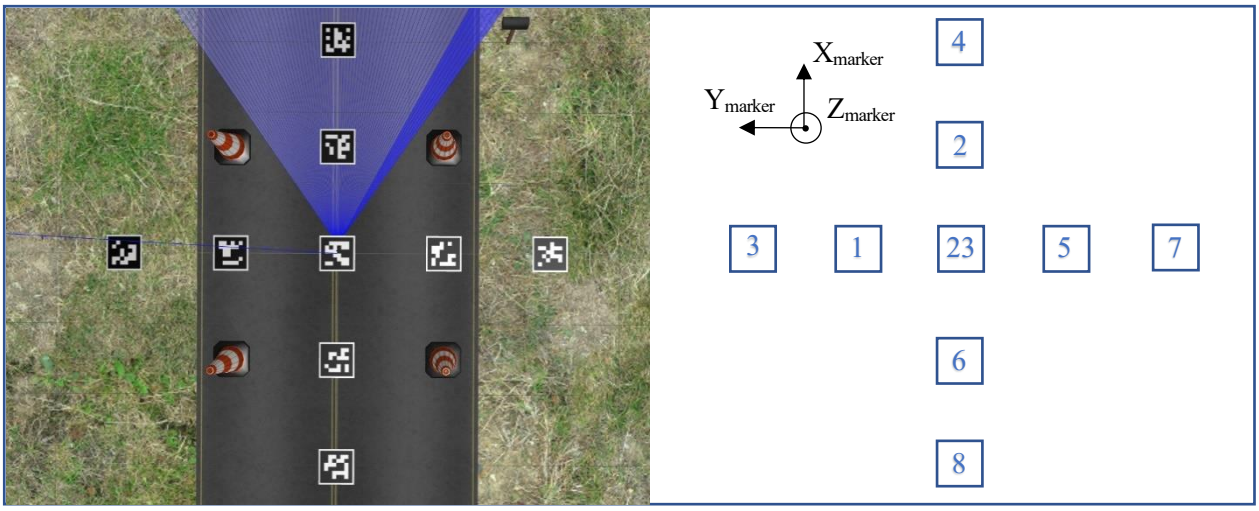

Ilustración 17. Imagen de los marcadores en el mundo y esquema con los identificadores de cada uno. El sistema de referencia indicado es el del mundo.

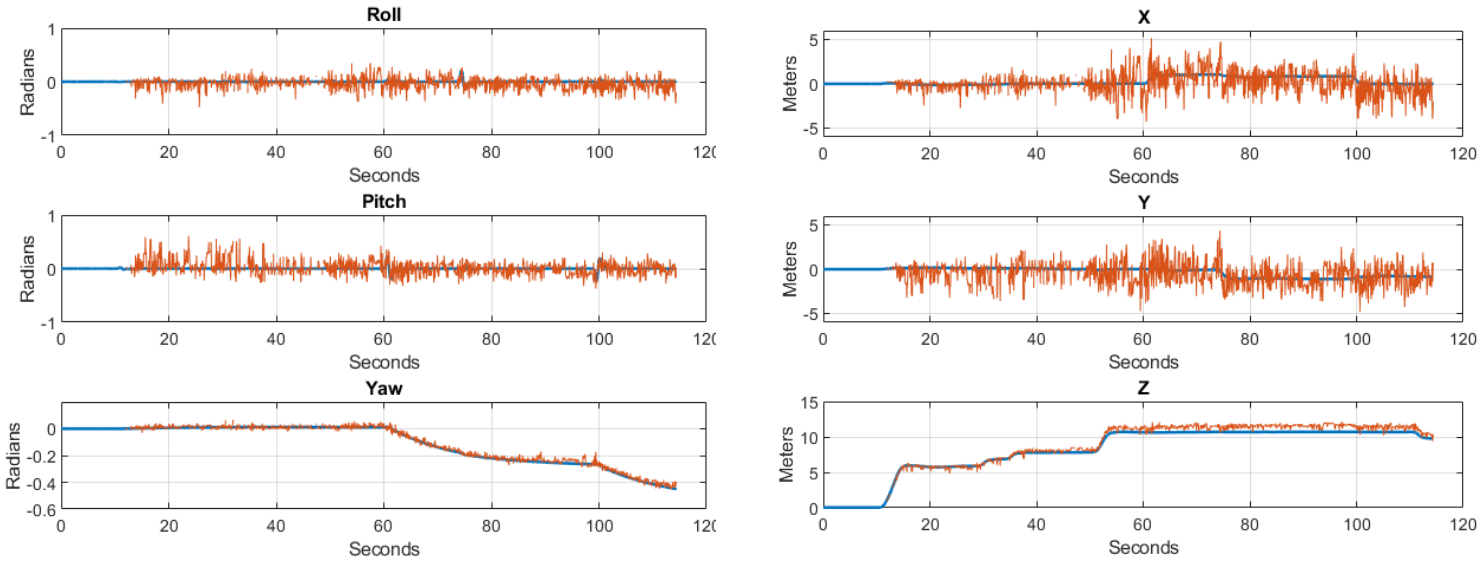

Ilustración 18. Resultados de la simulación para un total de 3 marcadores ArUCo (el central y dos adicionales). En azul se ve el valor real y, en rojo, la estimación obtenida.

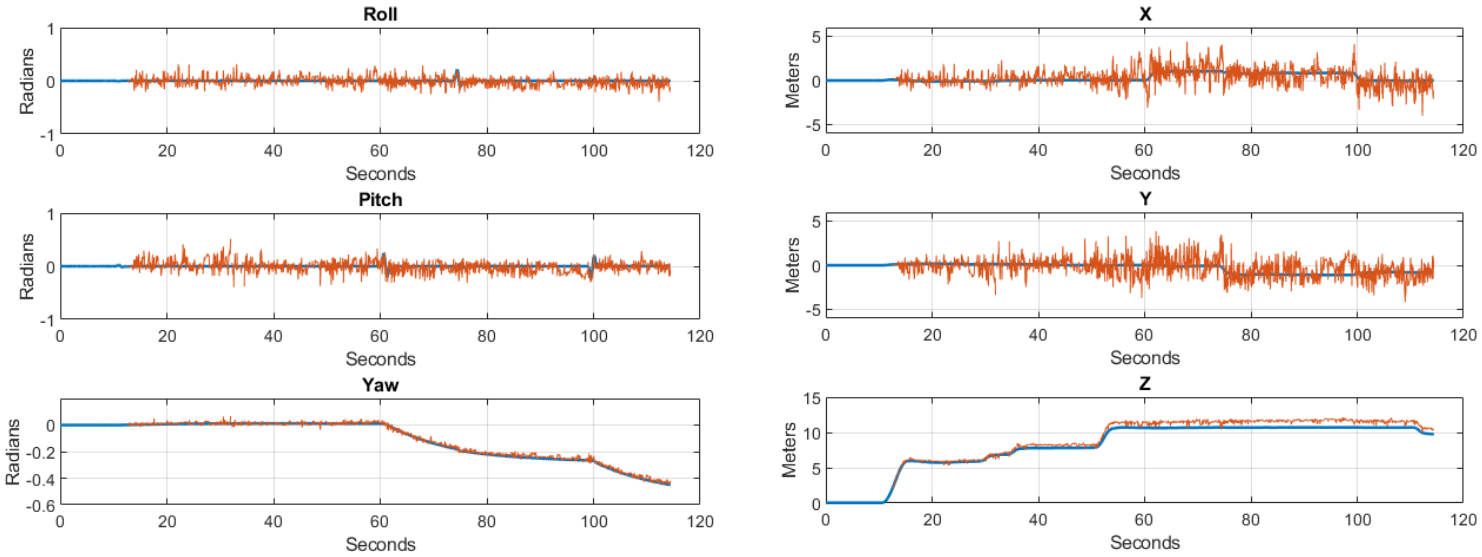

Ilustración 19. Resultados de la simulación para un total de 5 marcadores ArUCo. La línea azul es el valor real y, la roja, la estimación obtenida.

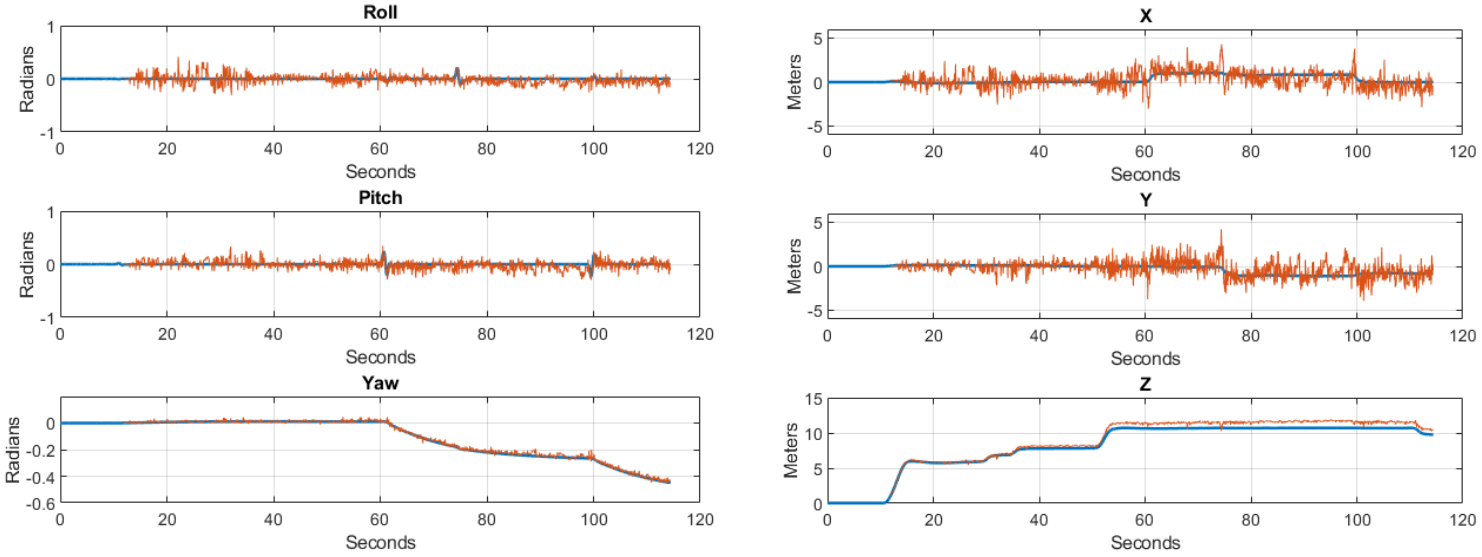

Ilustración 20. Resultados de la simulación para un total de 9 marcadores ArUCo. La línea azul es el valor real y, la roja, la estimación obtenida.

Si comparamos los resultados obtenidos en la simulación, se ve cómo el comportamiento mejora considerablemente de 1 a 2, no siendo tan visible a simple vista la mejora de 2 a 4 y de 4 a 9, aunque puede notarse si se fija un poco. Se podría intentar mostrar las diferentes gráficas superpuestas, pero la variación en torno al punto real hace que las líneas se solapen, haciendo que no se vea nada en la gráfica y dificultando la interpretación más que si se vieran las gráficas por separado. Por ello, y para poder cuantificar las diferencias en las estimaciones, se va a calcular el error cuadrático medio y el error absoluto medio. Las simulaciones son las mismas, pero el número de estimaciones difiere en cada uno de los modelos puesto que no se tarda lo mismo en procesar 9 marcadores que 1. Por ello, la media no se realizará para el mismo número de marcadores si bien la diferencia entre el que más medidas toma y el que menos difiere solamente en un 4% del que más y, como son valores medios, no debería afectar a la interpretación de los resultados.

	Roll [rad]	**Pitch [rad]**	**Yaw [rad]**	**X [m]**	**Y [m]**	**Z [m]**
1 marcador	0.1004	0.1430	0.0215	2.0606	1.5247	0.4490
3 marcadores	0.0816	0.0972	0.0127	0.8596	0.9405	0.5526
5 marcadores	0.0630	0.0787	0.0101	0.6529	0.7768	0.6301
9 marcadores	0.0613	0.0588	0.0090	0.6299	0.6032	0.5893

Tabla 1. Error absoluto medio según el número de marcadores.

	Roll [rad^2]	**Pitch [rad^2]**	**Yaw [rad^2]**	**X [m^2]**	**Y [m^2]**	**Z [m^2]**
1 marcador	0.0409	0.0605	0.0009	6.8616	4.1922	0.3730
3 marcadores	0.0124	0.0178	0.0003	1.4471	1.5023	0.4306
5 marcadores	0.0070	0.0106	0.0002	0.7983	1.0284	0.5182
9 marcadores	0.0069	0.0060	0.0001	0.6908	0.6875	0.4573

Tabla 2. Error cuadrático medio según el número de marcadores.

Los resultados obtenidos son congruentes con los vistos en las gráficas de la simulación, conforme se va aumentando el número de marcadores disminuye el valor de los medidores del error. El único caso en que se nota un ligero empeoramiento es en el eje Z. Esto es debido a que, conforme el UAV se aleja del marcador el número de píxeles que ocupa en el sensor es menor, aumentando las posibilidades de dar peores estimaciones y, además, al tener que calcular distancias más grandes, pequeños errores en las mediciones de las proporciones del marcador se ven amplificados. Juntando estos dos efectos, se tiene un algoritmo que da una estimación superior al valor real y, conforme aumenta el número de marcadores, una disminución de la desviación con respecto al valor central. Al tener una mayor variación para menos marcadores, hay un conjunto de valores que se desvían por debajo de la media de la estimación que hacen que se acerque mejor a la altura real si bien esta desviación no sería algo deseable. Estos temas de las limitaciones de los marcadores se tratarán un poco más en profundidad en las conclusiones del capítulo.

Siendo un poco más críticos, hay que reconocer que aunque mejore la estimación de la orientación, esta sigue dejando que desear. Es por ello por lo que se van a plantear varios métodos que lleven a un mejor resultado. Es más, como para el cálculo de la posición tiene que hacer la inversa de rotación actual, una mala estimación de la orientación actual tiene como resultado un empeoramiento de la estimación de la posición, por lo que esta va a ser nuestra prioridad a mejorar. En esta línea se plantean tres posibles soluciones:

- Determinación de la matriz a mano suponiendo que los ángulos del dron varían poco.
- Uso de las medidas del giroscopio de la IMU.
- Uso de las medidas del magnetómetro de la IMU.

6.3 Simulación con un marcador y suponiendo giros pequeños

Como el UAV en la simulación está la mayor parte del tiempo horizontal y su parte delantera suele apuntar hacia el Norte (aunque va girando poco a poco con el tiempo). Se podría realizar una aproximación a mano de la matriz de rotación que pasa de unos ejes a otros. Esto no nos solventaría el problema de la orientación, pero sí mejoraría el valor de la posición. Suponiendo que el dron ha girado solo ángulos pequeños con respecto al inicial, esta matriz será igual a intercambiar los ejes X e Y y multiplicar por -1 todos los ejes. Es decir, las traslaciones en Z serán ahora de signo contrario y las de X e Y estarán intercambiadas y cambiadas de signo (Ilustración 22). Es decir, realizando el cambio en la ecuación del capítulo 3.1 quedaría:

$$t_{marker,marker\rightarrow camera} = -Rot \cdot \left(-t_{camera,camera\rightarrow marker}\right) = Rot \cdot t_{camera,camera\rightarrow marker}$$

Donde $$Rot = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

El resultado de la posición será válido sólo cuando el UAV esté en posición horizontal y apuntando hacia el Norte. En el momento en el que se mueva o acumule una deriva de giro en torno al eje Z, esta aproximación cometerá errores puesto que los ángulos no serían nulos y la suposición ya no sería cierta. Este efecto se muestra claramente en las gráficas (Ilustración 21) para las que se ha usado *handmade_marker.launch*.

Se ha mostrado únicamente las posiciones ya que, bajo esta suposición los ángulos de giro son nulos, por lo que no tiene mucho sentido mostrarlos (aunque en su defecto podría cogerse la estimación de los marcadores que ya sabemos que no es muy buena).

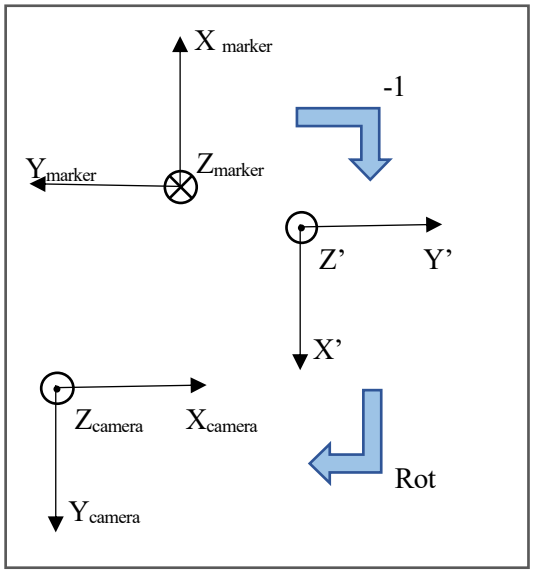

Ilustración 22. Esquema de los cambios realizador por la matriz de rotación calculada a mano. Los ejes finales de la cámara coinciden con los reales si no hay ningún giro de la cámara.

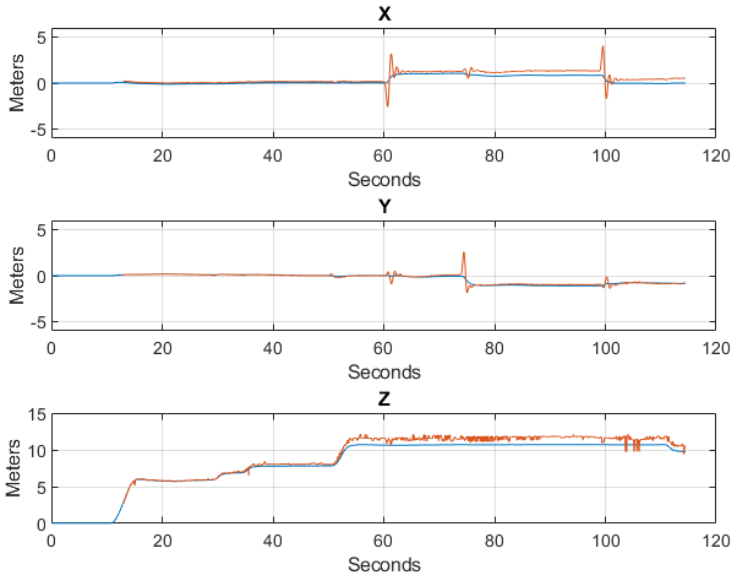

Ilustración 21. Resultados en la posición al utilizar la matriz heurística de cambio de base.

El principal problema de este método es que el dron no va a estar siempre apuntando hacia el Norte (quien dice Norte dice una dirección concreta), sino que tiene una pequeña deriva en el movimiento y conforme va avanzando la simulación, comienza a girar en torno a Z ya que los motores no son exactamente iguales.

6.4 Simulación con un marcador y giroscopio

Como el método anterior no siempre es válido y, además, no nos soluciona el problema de querer tener una buena estimación de la orientación, se prueba a utilizar la IMU del UAV para conseguir la orientación y utilizar esta a su vez para calcular la inversa. En este caso se va a probar con el giroscopio, es decir, habrá que integrar en el tiempo las medidas de la velocidad angular para así poder obtener los ángulos totales. Es decir, se estimará el valor de $(R_{camera \to marker})^{-1}$ del capítulo 2.5.3 mediante la matriz de rotación que nos proporcionan los ángulos estimados con la integración de la medida en el tiempo. La relación entre las matrices de rotación sería:

$$(R_{camera \to marker})^{-1} = R_{world \to UAV} \cdot Rot_{UAV \to camera}$$

Donde $R_{i,world \to UAV}$ es la matriz de rotación que obtenemos al multiplicar los giros indicados por los ángulos proporcionados por la integración.

Los resultados obtenidos con *IMUinv_integration_marker.launch* consiguen dar una solución al problema de estimar los transitorios de movimiento y, además, como se basa en la medida de orientación de la IMU, nos daría unos valores de orientación a utilizar. Sin embargo, presentan un problema: la deriva con el tiempo. La integración de los valores de la velocidad angular lleva consigo

la integración de unos errores que se irán acumulando con el tiempo, haciendo que si pasa tiempo suficiente y, aunque no haya ningún movimiento en el UAV, se perciba en las gráficas una tendencia lineal de los ángulos como se ve en Ilustración 23. Es decir, es como si al valor que debería estimar se le sumara una recta, por lo que, si ha pasado suficiente tiempo, el valor que proporciona este algoritmo carecería de sentido. Para intentar minimizar este efecto (al menos a corto/medio plazo) se calibra la IMU. Esto consiste en recoger los valores aportados por la IMU cuando el dron está en posición de reposo en el suelo y calcular el *offset* de las medidas del giroscopio y acelerómetro a través de la media de esos valores. Dicho offset será restado a las medidas que provea la IMU en el futuro. Realizando esta tarea se obtienen los *offsets* de la Tabla 3.

X_acc [m]	-0.1878
Y_acc [m]	-0.2297
Z_acc [m]	0.1152
Roll_vel [rad]	-5.426e-4
Pitch_vel [rad]	5.847e-3
Yaw_vel [rad]	-8.303e-4

Tabla 3. Offsets del giroscopio y acelerómetro.

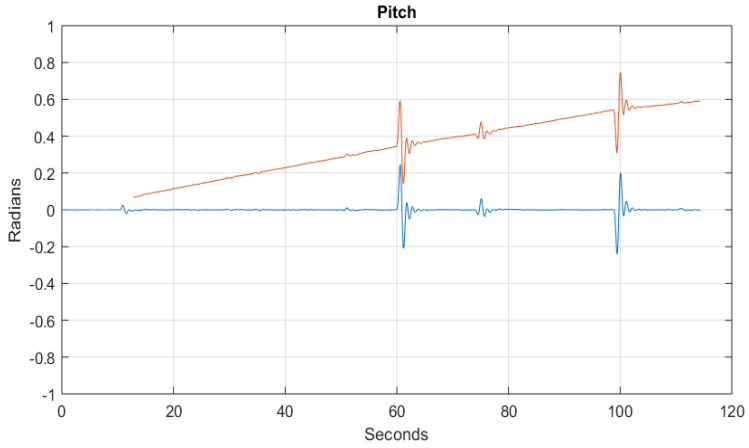

Ilustración 23. Ejemplo de deriva temporal en un ángulo.

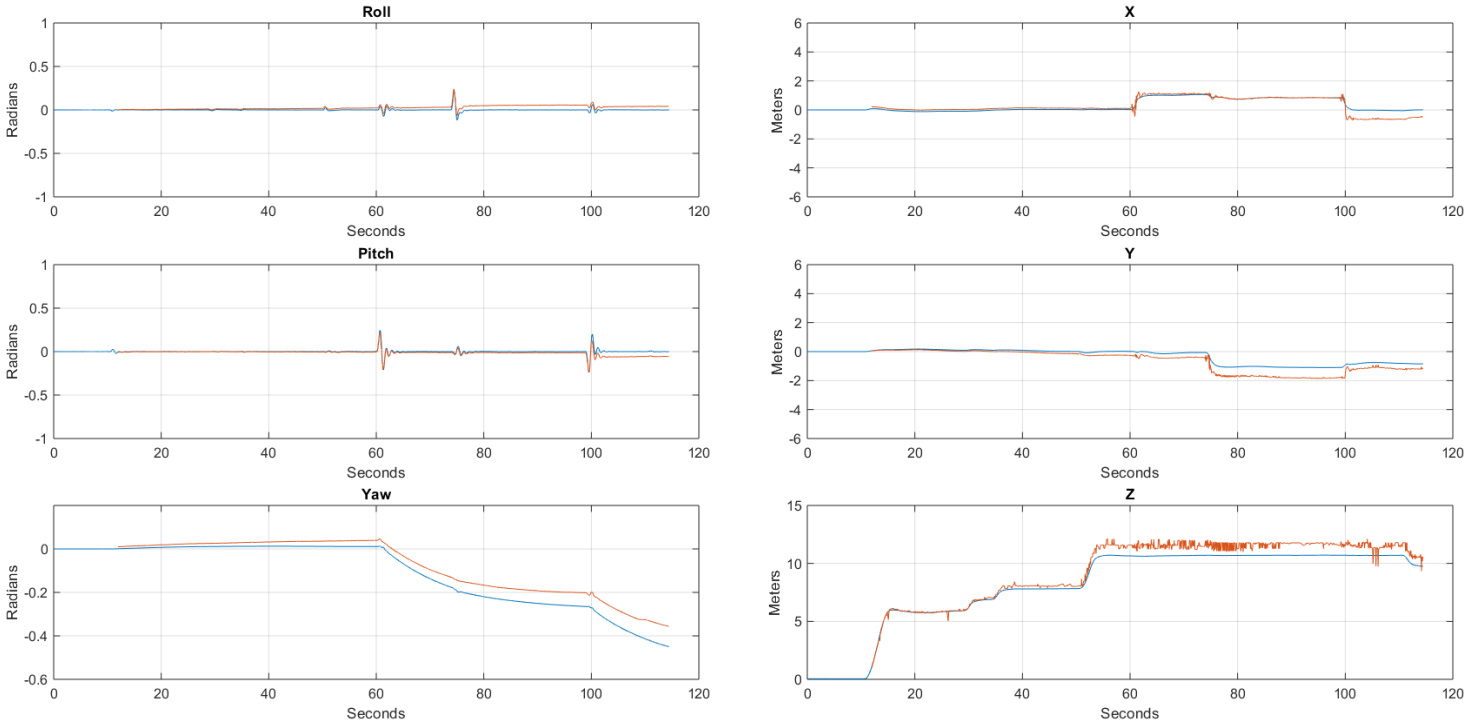

Ilustración 24. Resultados del uso de la orientación proveniente del giroscopio calibrado. En rojo el valor estimado y, en azul, el real.

6.5 Simulación con un marcador y magnetómetro

La IMU del ErleCopter también proporciona una estimación de los ángulos basada en las medidas de un magnetómetro. Se va a probar a comprobar el funcionamiento de esta otra opción, para ello se utiliza el fichero *IMUinv_magnetometer_marker.launch*, para el que se realiza la misma transformación del vector de traslación indicada en el apartado anterior.

El valor proporcionado de la orientación es prácticamente el real, los resultados son bastante buenos. Si bien, cabe destacar que en la realidad las medidas de los magnetómetros no tienen tal fiabilidad y son dependientes de la cantidad de ruido electromagnético que haya en los alrededores, por lo que no se debe esperar que este resultado sea exactamente replicable fuera de una simulación. Es por ello que, aunque el resultado sería ideal, en principio se seguiría utilizando la integración de la medida del giroscopio para el resto de resultados. En la simulación no tiene mucho sentido fusionar a través de un filtro de Kalman la medida del magnetómetro y la del giroscopio porque el magnetómetro es muchísimo más fiable y no lo mejoraría apenas, pero, en la realidad, sí se podría proponer el uso de un filtro de Kalman extendido para intentar conseguir una buena estimación de la orientación.

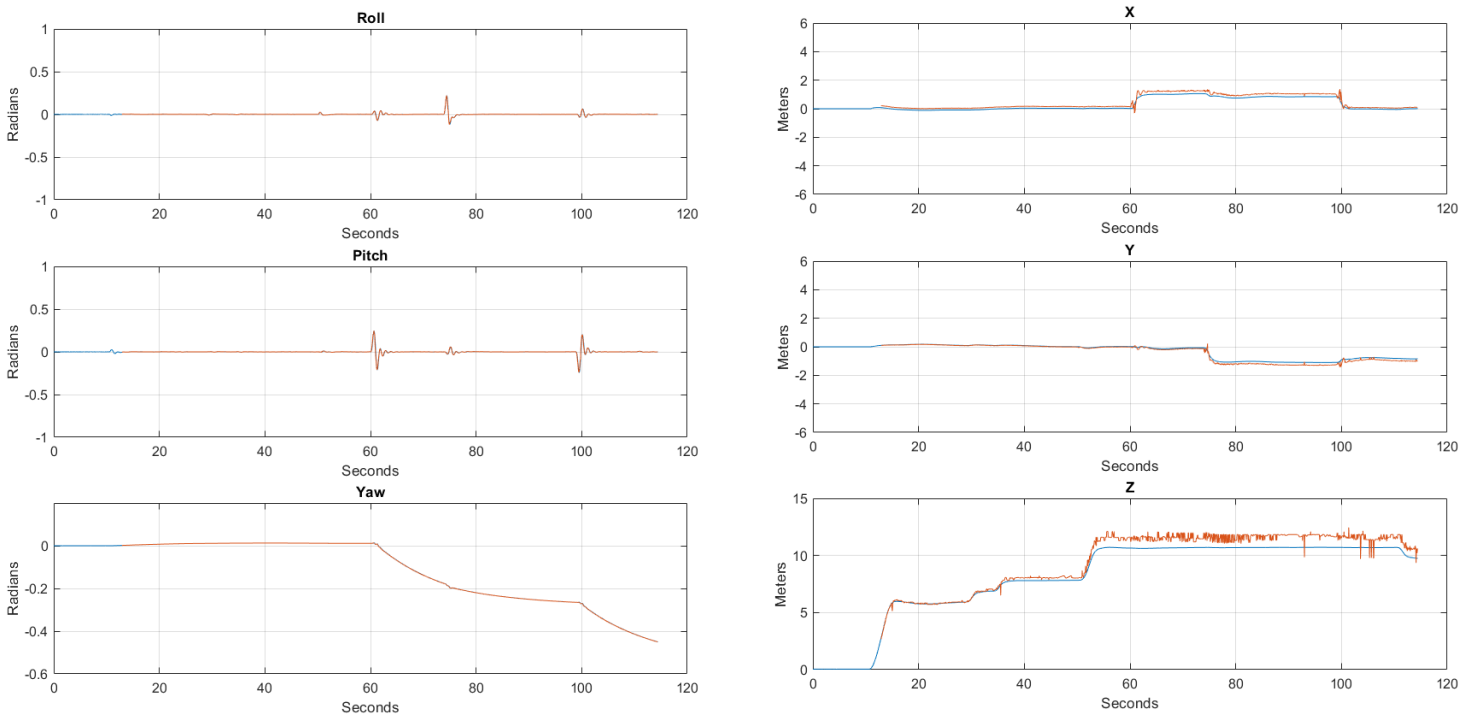

Ilustración 25. Resultados del uso de la orientación proveniente del magnetómetro.

6.6 Conclusiones

A modo de resumen se adjunta una tabla en la que se ha calculado el error cuadrático medio y el absoluto medio para las medidas de la posición. Se han omitido los resultados de la orientación puesto que para el caso del uso de la IMU ese valor no lo proporciona ArUCo, por lo que no tendría mucho sentido cuando el objetivo de este capítulo es comparar las distintas alternativas a la hora de utilizar estos marcadores. Adicionalmente, también se han reescalado las gráficas (Ilustración 26) en las que se usan tanto la IMU como la aproximación a mano para ver mejor las diferencias en los resultados.

	X [m]	Y [m]	Z [m]
1 marcador solamente	2.0606	1.5247	0.4490
9 marcadores	0.6490	0.6032	0.5893
1 marcador + rotación heurística	0.2161	0.0934	0.6782
1 marcador + giroscopio	0.1931	0.3588	0.6668
1 marcador + magnetómetro	0.0498	0.0947	0.7045

Tabla 4. Error absoluto medio para los algoritmos utilizados.

	X [m^2]	Y [m^2]	Z [m^2]
1 marcador solamente	6.8616	4.1922	0.3730
9 marcadores	0.7304	0.6875	0.4573
1 marcador + rotación heurística	0.1601	0.0467	0.6228
1 marcador + giroscopio	0.0964	0.1857	0.6003
1 marcador + magnetómetro	0.0053	0.0147	0.6572

Tabla 5. Error cuadrático medio para los algoritmos utilizados.

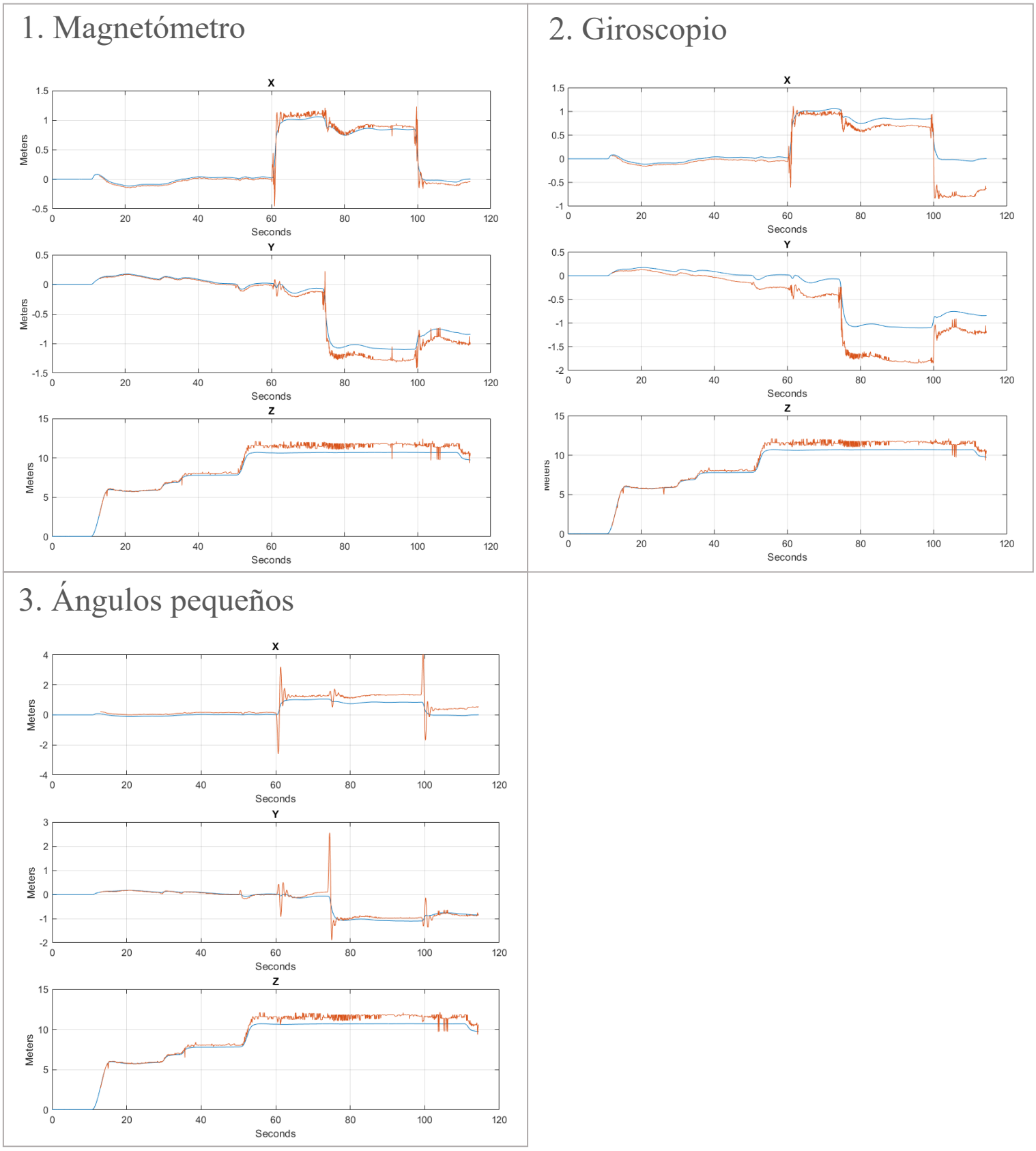

Ilustración 26. Gráficas de posición reescaladas para poderse comparar mejor. Referidas a: 1. Magnetómetro, 2. Giroscopio, 3. Suposición ángulos pequeños

Con los datos de las tablas Tabla 4 y Tabla 5 y las gráficas de la Ilustración 26, se puede realizar una comparativa a la luz de los resultados obtenidos.

Aunque aumentando el número de marcadores se mejora la estimación de la posición, no es pensable que sería una idea útil teselar el suelo con marcadores para que el dron se pueda mover y, a la vez, ubicarlo bien. Teselar el suelo no es una opción simple o que tenga una puesta en práctica rápida ya que además deberían considerarse la distancia entre los marcadores. Frente a esta, aparecen las otras 3 opciones anteriormente nombradas.

El giroscopio en este caso presenta unos valores aceptables en comparación con los vistos anteriormente. Concretamente, si se compara su comportamiento con aquel de los 9 marcadores, se ve cómo el giroscopio proporciona una mejor medida aunque no llegue a ser tan preciso en Z.

Con respecto a la estimación a mano de la matriz de rotación necesaria para cambiar de base la estimación de posición del dron, da unos resultados bastante buenos en términos generales según el MSE (*Mean Squared Error*) y el MAE(*Mean Absolute Error*). Es más, los resultados obtenidos podrían competir con aquellos del giroscopio. Esto se debe a que, como se comentó anteriormente, los mayores errores de este algoritmo es cuando el dron se mueve, lo cual es visible en la gráfica. Estos grandes errores están presentes en pequeños intervalos de tiempo, por lo que ese error se camufla en la media del error medio absoluto, pero puede verse un poco mejor en el error cuadrático medio ya que, al elevar al cuadrado los errores, penaliza más aquellos errores grandes, lo que hace que en media se difuminen menos. Es decir, este acercamiento nos valdría si no nos importara tener una mala estimación cuando esté en movimiento ya que, cuando pare, volverá a dar un buen resultado. Adicionalmente y como se comentó en su apartado hay que asegurar que el giro en Z no aumenta mucho, porque eso afectaría también al resultado.

Para la estimación basada en el magnetómetro los resultados son bastante buenos, basta con echarle un vistazo a la gráfica reescalada. Las estimaciones en X e Y son las mejores de los algoritmos probados, esto se debe probablemente a que el ruido electromagnético en la simulación es muy pequeño como ya se comentó. En la realidad lo suyo sería fusionar esta medida con el giroscopio usando un filtro de Kalman extendido por ejemplo.

Este último método, al presentar unos valores de los ángulos tan cercanos a los reales, nos permite extraer información sobre las condiciones ideales de funcionamiento del dron, es decir, vamos a examinar la gráfica obtenida con el magnetómetro.

Por un lado, se tiene que las primeras detecciones del dron no se dan desde el momento cero, eso se debe a que la simulación comienza con el dron en el suelo y, por tanto, la cámara no tiene ni rango de visión completo del marcador ni capacidad de enfoque. Cuando alcanza los 2.5m aproximadamente es cuando se empiezan a obtener datos. Por el otro, conforme se va alejando empeoran los valores proporcionados como se ve a partir de alcanzar el dron los 10m. No solamente se aleja del valor real, sino que aumenta la desviación con respecto al valor medio en esa ubicación. Este efecto de la altura en la estimación se va a estudiar con un ensayo mostrado en la Ilustración 27 usando el magnetómetro (ya que interesa la mayor precisión posible en la simulación).

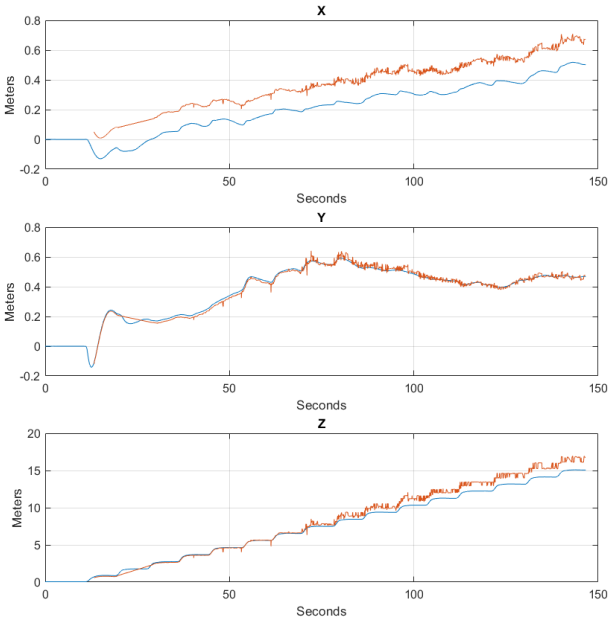

Ilustración 28. Resultado de comprobación del comportamiento con la altura.

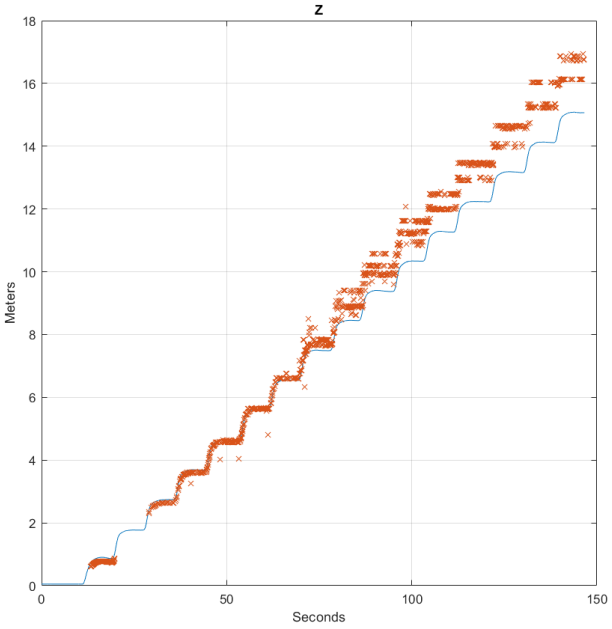

Ilustración 27. Puntos de altura obtenidos en el estudio.

Inicialmente, la estimación de la posición está formada por líneas rectas aunque apenas se distinguen en la gráfica a no ser que se acerque lo suficiente. Esto no es más que un artificio debido a la unión de los puntos que se han calculado, es decir, hubiera que representar los puntos como cruces, se vería más claro el comportamiento real (mostrado en la Ilustración 28). En esa nueva gráfica se aprecia como para alturas inferiores a 2.5m aproximadamente no hay detecciones consistentes en el tiempo debido a que el algoritmo empleado no está pensado para elementos tan grandes (en tamaño relativo a la proyección). Conforme se va alejando empieza a detectar más a menudo el marcador, obteniéndose así resultados continuos en el tiempo.

Sin embargo, la cámara también tiene un límite superior el cual depende fuertemente de la resolución. Conforme se va alejando la cámara del marcador, menor será el número de píxeles que ocupará este en el sensor y, por tanto, pequeñas distorsiones en la toma de la imagen pueden acarrear cambios más bruscos en la estimación o incluso la no detección. Esto último es fácilmente visible en la Ilustración 28 en la que se ve cómo el número de cruces disminuye para las últimas alturas y cómo estos valores tienen una mayor varianza o desviación con respecto a la media. Si hubiera que poner un límite a partir del cual dejaría de presentar resultados óptimos, se podría elegir los 8m por inspección ocular, ya que a partir de esa altura la estimación empieza a perder calidad de manera notoria. Adicionalmente a la resolución, pequeños errores en las medidas del marcador o de la proyección de este en la cámara se amplifican conforme aumenta la altura, dando ese aparente error permanente en las gráficas.

Es por este motivo por lo que en las gráficas realizadas a lo largo de este capítulo se ha procurado no superar los 10m de altura, pero superando a la vez ese límite óptimo para poder llegar hasta esta conclusión de manera natural.

Una última nota. Seguramente se habrá dado cuenta de que hay un offset de en torno a 0.15m en X entre la posición según la cámara y la del marcador. Esto se debe a que aunque el marcador se encuentre en el punto (0,0,0) de la simulación al igual que el dron, la cámara no se encuentra en el centro del dron, sino que estaría desplazada 0.15m en el eje X. Esto querría decir que habría que pasar la posición de la cámara al centro del dron, lo cual no sería más que restar 0.15m a las medidas obtenidas en X y sin modificar la orientación (no se está modificando la posición relativa de los ejes). En la Ilustración 29 se muestra la diferencia en el resultado si lo hubiéramos tenido en cuenta para la Ilustración 27. Aunque esto no se haya indicado hasta ahora, ha sido tenido en cuenta a la hora de calcular los MAE y MSE en X y dibujar el resto de las gráficas en X (menos esta última que se ha utilizado como prueba una vez teníamos certeza de que estábamos usando la orientación correcta).

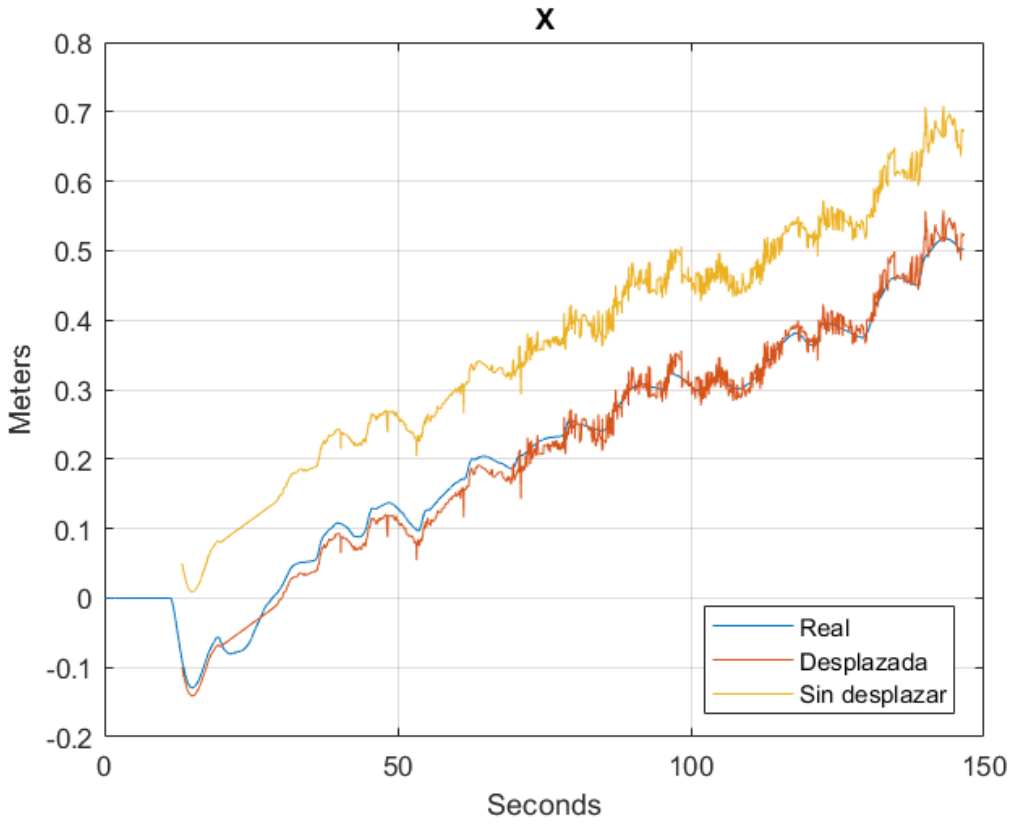

Ilustración 29. Desplazamiento en X para la comprobación de la altura teniendo en cuenta el offset de posición de la cámara.

Recuérdese que no solamente se iba a estudiar el uso de marcadores ArUCo como una forma de obtener la posición global del UAV, sino también como un medio para obtener la estimación de la altura inicial del UAV a partir de la cual ir calculando los desplazamientos. Es por ello que, sabiendo que la medida a partir de 8m empieza a empeorar, se va a elegir un punto algo inferior, los 7m, para el despegue por defecto del UAV. Es decir, el UAV despegará desde encima de la base (el marcador) e irá midiendo la altura gracias al marcador hasta llegar a la altura de 7m, en la que empezará a funcionar el programa encargado del posicionamiento relativo.

7 RESULTADOS EXPERIMENTALES PARA EL CASO CON MARCADORES NATURALES

En este capítulo se procederá a mostrar, comparar y analizar los resultados obtenidos por el método basado en la homografía, es decir, en el que no hay referencias absolutas. Para ello, se van a realizar simulaciones en dos entornos diferentes (entendiéndose como entorno el conjunto de objetos/elementos/modelos que aparecen visualmente en la simulación): uno en el que los objetos serán de pequeña altura para que los puntos detectados se encuentren más o menos en el suelo y otro en el que se utilice cualquier tipo de objeto de cualquier altura como, por ejemplo, una casa.

En esta línea se probará para cada uno de ellos cuál sería el comportamiento si tuviera acceso a la altura y orientación reales (necesaria para cambiar a la base global la traslación) y en el caso en el que estos datos tuvieran que estimarse a partir de las estimaciones anteriores.

Finalmente, se plantearán 3 modificaciones a estos algoritmos con el fin de poder mejorar el resultado obtenido:

- Sustitución del algoritmo SURF y por el algoritmo ORB.
- Forzar a que sean cero las estimaciones de X e Y cuando las variaciones de giro en X e Y sean prácticamente nulas. Esto se basa en la suposición de que si el dron apenas gira en X e Y, será porque el dron está en posición horizontal (en vuelo continuo sería poco probable que mantenga un rango de valores de ángulos tan acotado).
- En vez comparar cada fotograma con el anterior, se van a usar fotogramas de referencia (cambiantes en el tiempo) de tal forma que se compararán con estos, postergando en el tiempo los efectos derivados de la suma de variaciones.

7.1 Sobre la estimación inicial

Recordemos que, para empezar a utilizar la homografía, se hacía necesario establecer un punto de referencia, es decir, una posición y orientación inicial. Este punto no se pone en el suelo porque la cámara sería incapaz de calcular desplazamientos entre una imagen negra y otra a 1 metro. Además, la falta de puntos de interés en las primeras imágenes haría que posibles irregularidades en la imagen den lugar a estimaciones completamente desorbitadas. Por ello mismo y tal y como se comentó en el último párrafo del capítulo anterior, se va a emplear un marcador ArUCo como referencia absoluta durante el despegue hasta que el UAV supere los 7m de altura, momento en el cual comenzará a funcionar el algoritmo basado en la homografía y ya no dependerá de que se vea con suficiente nitidez

ese marcador. Puesto que la estimación inicial debería ser lo más parecida posible a la real ya que se quieren ver los errores del método basado en la homografía, se utilizará la medida del magnetómetro. En la realidad, se utilizaría el ya sugerido filtro de Kalman para giroscopio y magnetómetro.

7.2 Simulaciones en un entorno plano

Para estas simulaciones se utilizará el fichero .world del repositorio llamado *empty_plano_un_aruco.world* y que puede verse en la Ilustración 30. La pequeña dispersión en la altura de sus elementos debe ser un factor que juegue a nuestro favor ya que la altura será una buena estimación del factor de escala.

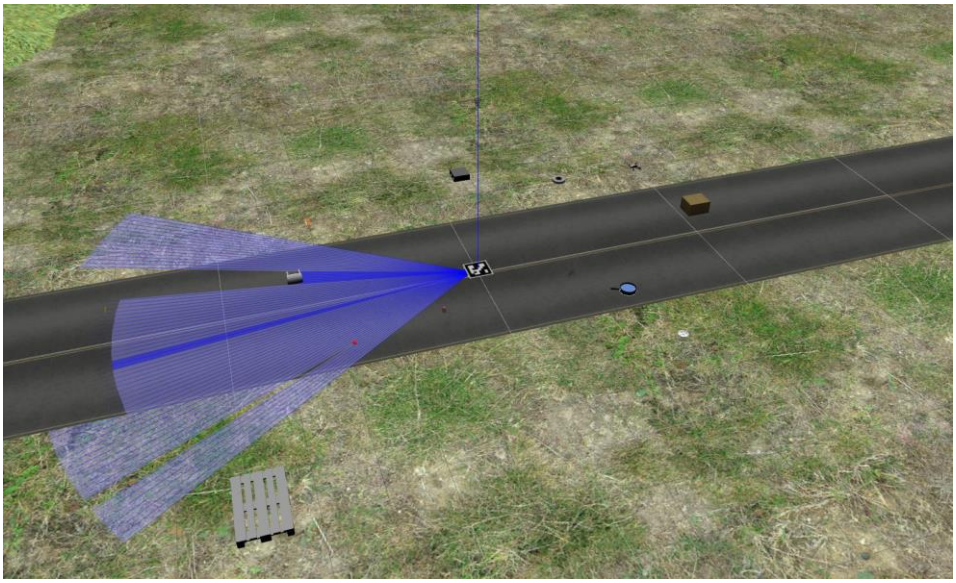

Ilustración 30. Imagen del mundo plano.

7.2.1 Orientación y altura real

Si ejecutamos el archivo *perfect.launch*, se realizará una simulación en la que se irá comparando cada fotograma con el anterior para calcular así los desplazamientos. Sin embargo, se recurrirá al valor real de la altura y orientación del dron para así estimar cuál sería el comportamiento del algoritmo en un entorno ideal. Esto, evidentemente, no sería implementable en la realidad, porque precisamente se desea conocer esos valores, pero permite saber cómo de buena sería la estimación si en cada instante tuviera como entradas los valores reales.

Como la homografía lo que proporciona en realidad son los desplazamientos y variaciones de los ángulos, se van a mostrar dos conjuntos de gráficas, una primera asociada a las variaciones medidas por nuestro algoritmo y una segunda que muestra la suma de esas desviaciones (los valores globales) para así tener una visión de cómo de bien hace su trabajo el algoritmo a nivel tanto de estimación individual de cada variación como de estimación del conjunto del movimiento.

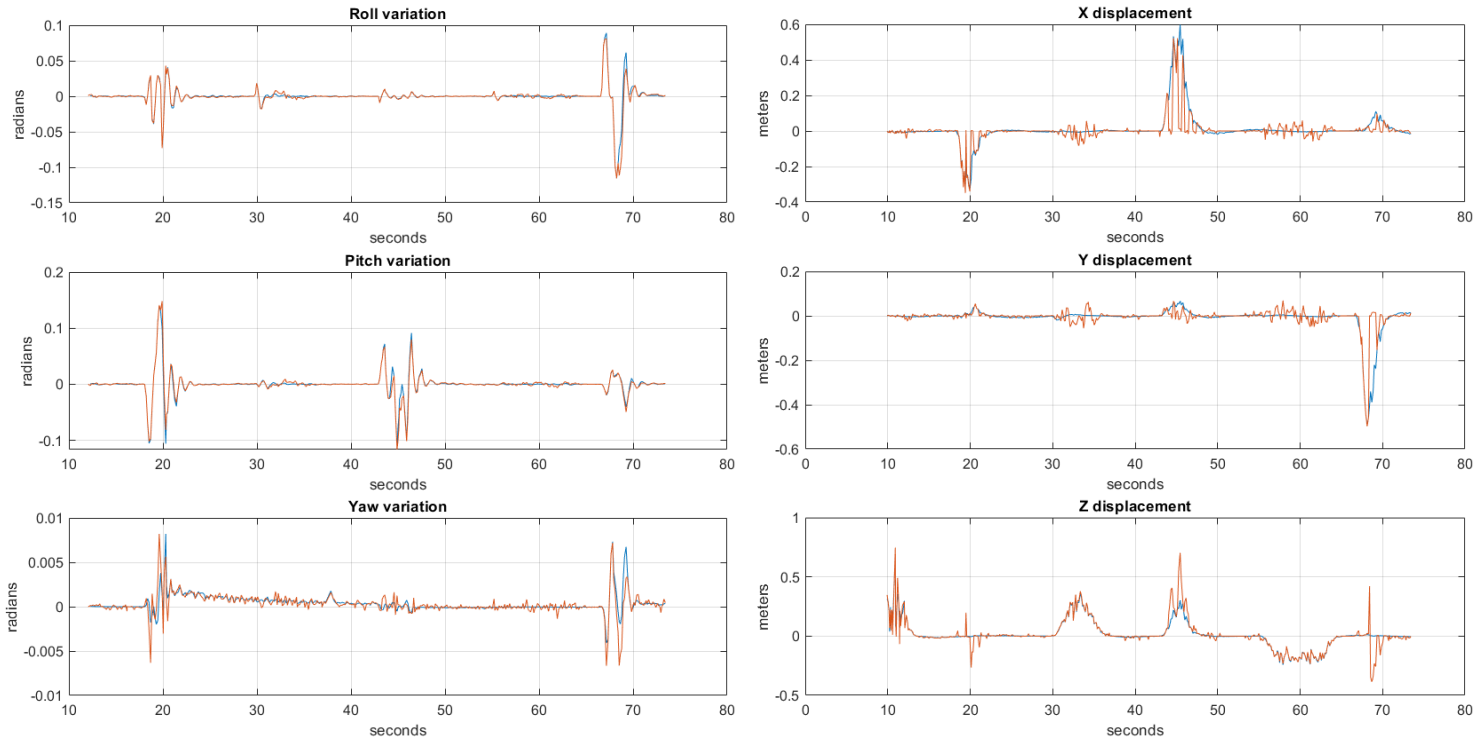

Ilustración 31. Resultados de las variaciones en las posiciones y ángulos de un fotograma a otro en la simulación en el mundo plano para valores reales de entrada. En azul se muestra el valor real y en rojo, el valor estimado.

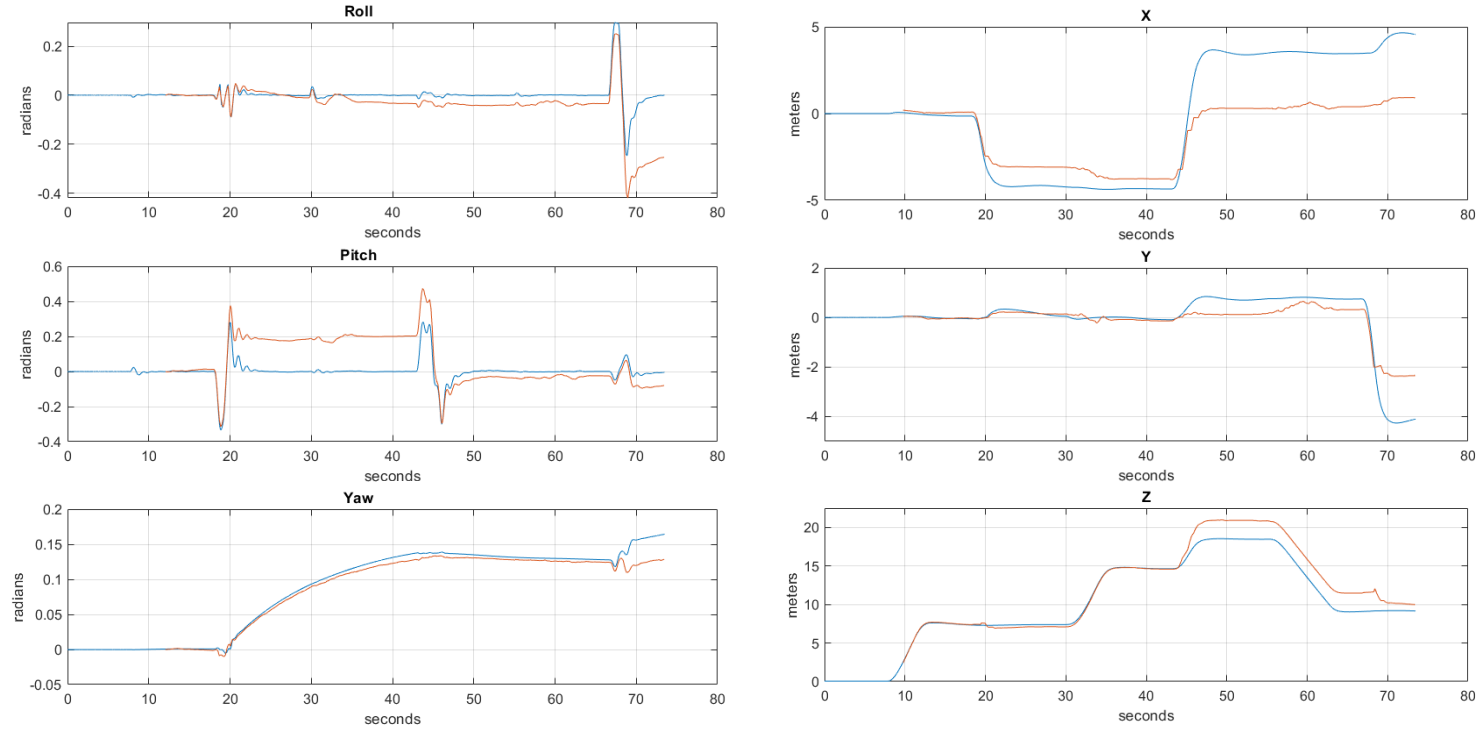

Ilustración 32. Resultados de las posiciones y ángulos acumulados en el mundo plano para valores reales de entrada. Al igual que siempre, en azul se muestra el valor real y, en rojo, el estimado.

Se ve claro como las estimaciones en la Ilustración 31 no son exactas, sin embargo si se nota una correspondencia entre los cambios reales y los estimados. Para el caso de la estimación de los ángulos, se ve en la gráfica de variaciones que uno de los principales problemas que aparenta la estimación es que sobredimensiona los cambios bruscos de ángulos, lo cual puede afectar negativamente al comportamiento global haciendo que tras giros bruscos y aunque el dron este en equilibrio, se indique que existen giros en X e Y tal y como se muestra en la Ilustración 32.

7.2.2 Orientación y altura estimadas

Ahora, se procede a comprobar el funcionamiento si se realimentan los valores estimados de orientación y altura para calcular la siguiente estimación. Para ello se utiliza *mixed.launch*. Se espera por tanto un comportamiento peor que en el caso anterior tanto en las traslaciones como en las orientaciones.

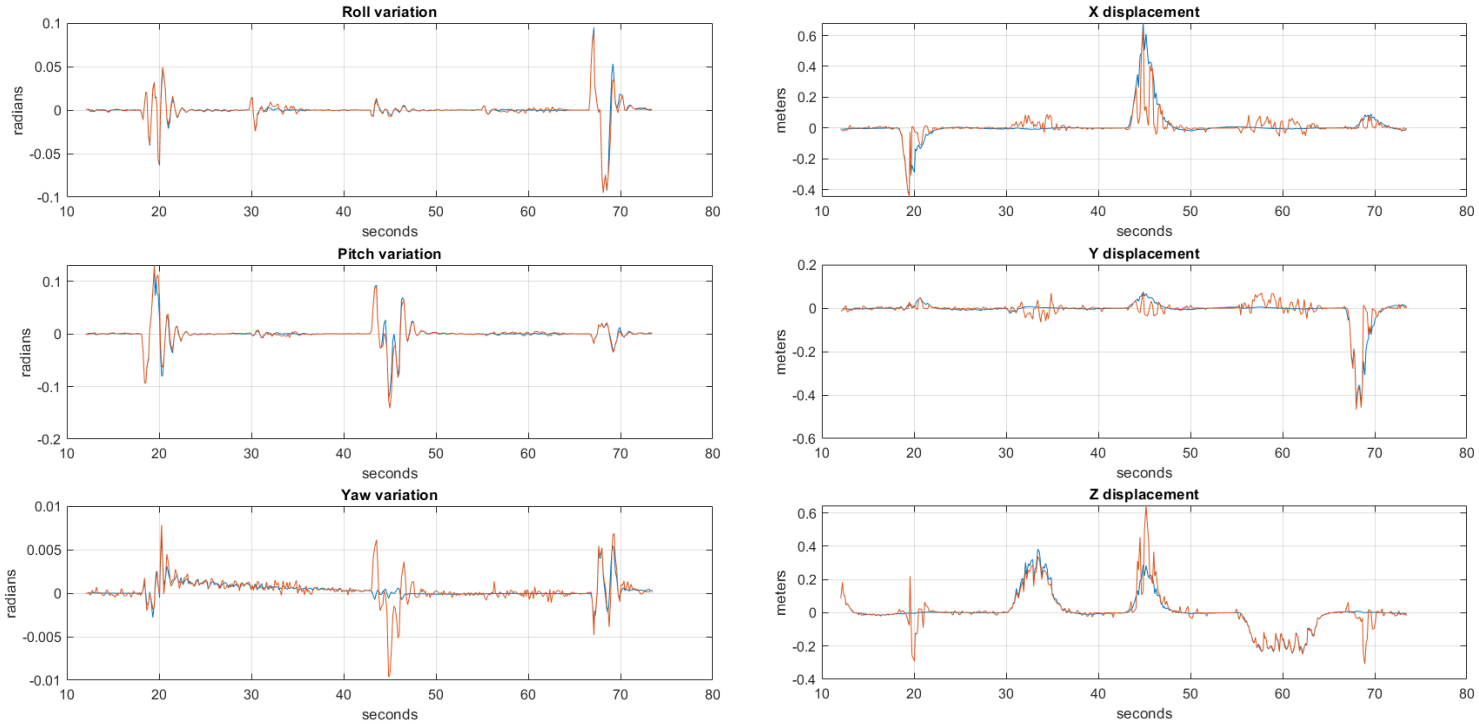

Ilustración 33. Resultados de las variaciones de ángulos y posiciones en la simulación en el mundo plano realimentando valores estimados. En rojo, el valor estimado y, en azul, el real.

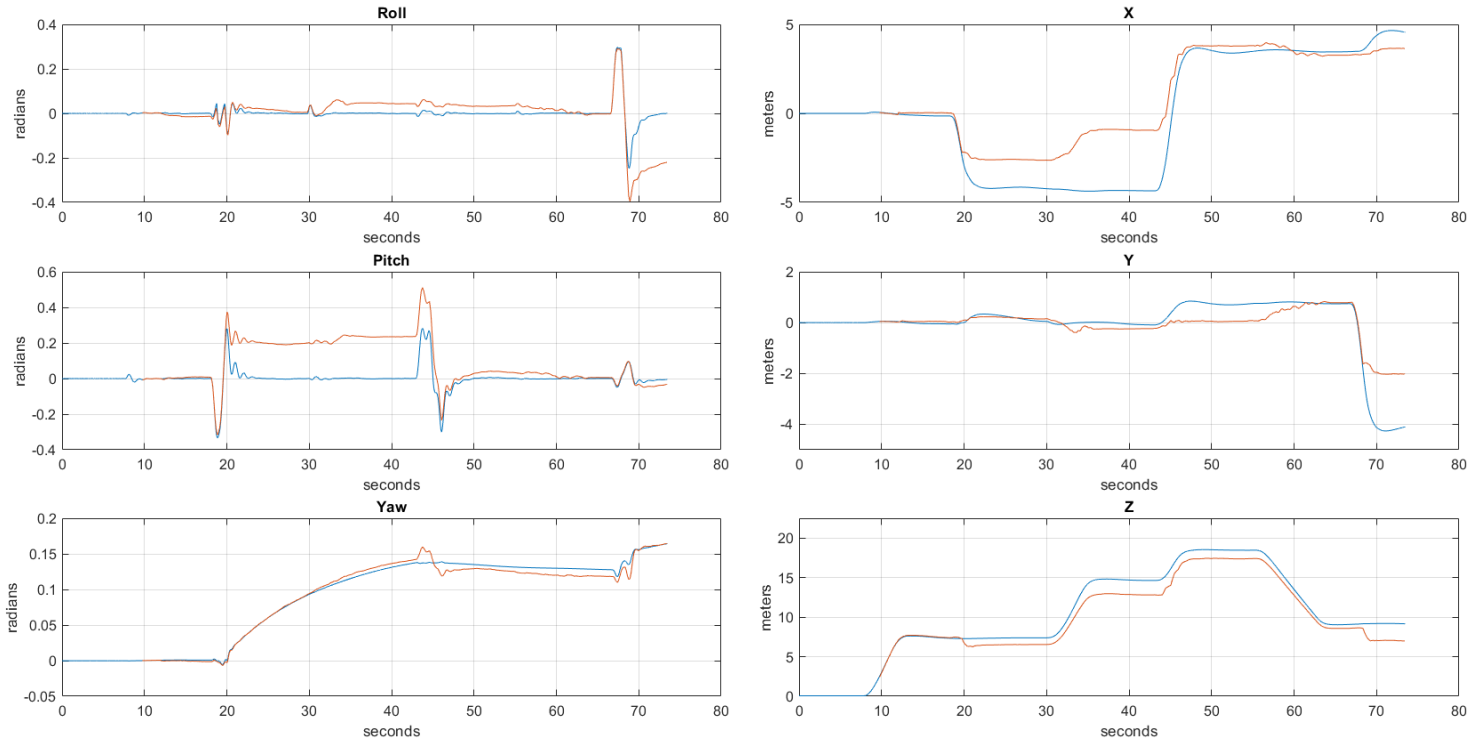

Ilustración 34. Resultados de la acumulación de ángulos y posiciones en la simulación del mundo plano realimentando valores estimados. En rojo, el valor estimado y, en azul, el real.

Para empezar, vamos a ver la Ilustración 33. En esta nueva gráfica se ve el efecto que tiene una estimación pasada que no sea coincidente con la real. En torno al segundo 45 se ve como aparece un giro en torno al eje Z que realmente no existe y que es debido a una estimación en el giro en Y que no se corresponde con el valor real. Como ya se indicó antes, un problema de este método es que una mala estimación de los ángulos puede dar lugar a ángulos residuales que deberían ser realmente nulos y, como la estimación de la posición depende de una buena estimación de la orientación (hay que realizar un cambio de base) la posición se ve afectada. Este efecto puede verse en la gráfica de posición de la Ilustración 34, donde se ve que la gráfica difiere de aquella obtenida en la situación óptima. Sin embargo, estas diferencias no parecen traducirse en un peor comportamiento puesto que esos defectos o excesos en las medidas de los ángulos dan lugar a estimaciones más cercanas a la realidad en algunos casos si bien esto es parte de la casualidad.

En resumen, se obtiene un comportamiento parecido a falta de posibles interacciones entre los ángulos y posiciones.

7.3 Simulaciones en un entorno con alturas variadas

En el anterior apartado se realizaron las simulaciones en un entorno con objetos de poca altura para intentar que la condición de que los puntos detectados se ubicaran en el mismo plano se cumpliera lo máximo posible. Ahora bien, ¿qué es lo que pasa cuando se utiliza un entorno en donde esta premisa se incumple? Esto es lo que se va a probar ahora, en un entorno mostrado en la Ilustración 35, más parecido a un caso real.

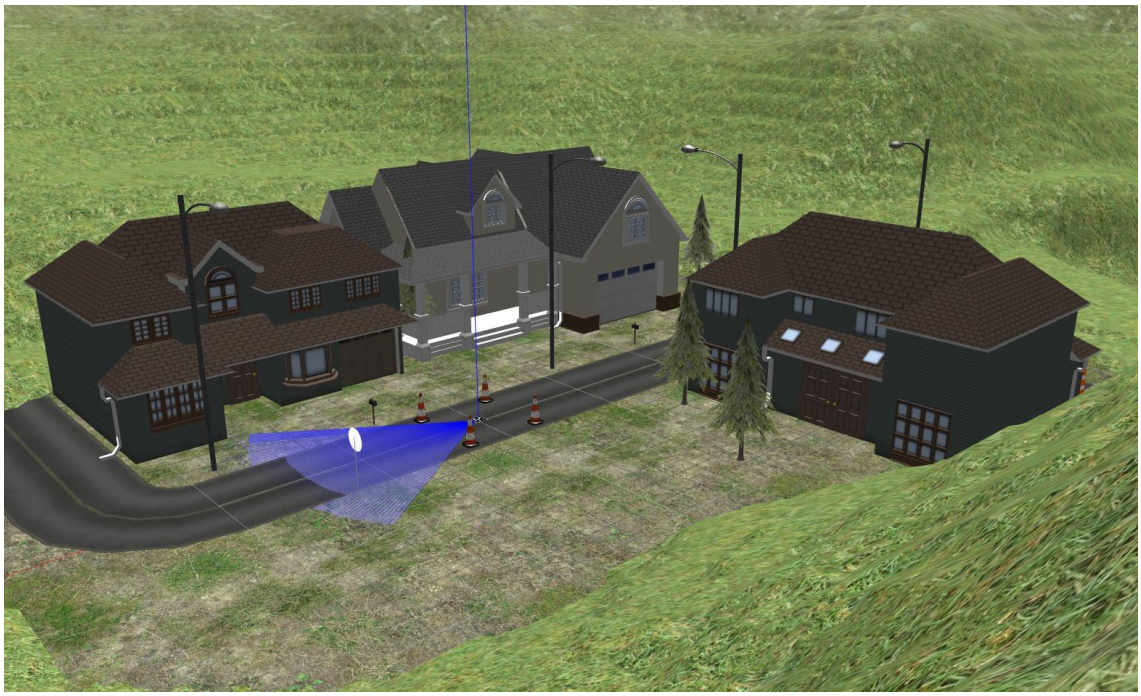

Ilustración 35. Imagen del entorno de simulación con diferentes alturas.

Para este entorno, hay conos de tráficos, farolas, casas, buzones… Es decir, objetos de altura variada. Cabe mencionar que el UAV también sobrevolará las casas, lo cual dará una idea de cómo puede afrontar estos desfases de altura de los puntos que pueden llegar a ser 5m. Es previsible por tanto, que los resultados empeoren con respecto al caso plano.

7.3.1 Orientación y altura real

Al igual que se hizo antes, se va a empezar con el caso óptimo en el que se utiliza la altura y orientación real (desconocidas en la realidad) para así tener una idea de cómo de buena en principio podrían ser nuestras estimaciones si bien es verdad que, como se ha visto antes, no hay una gran diferencia entre los resultados. El fichero que hay que correr es el mismo que en el caso anterior: *perfect.launch* .

Los resultados, mostrados en la Ilustración 37 y la Ilustración 36, muestran el mismo problema que para el caso del mundo plano: los ángulos residuales. Además, da la sensación de que se agravan las interacciones entre los cambios de desplazamiento. Esto puede deberse a que, como los puntos no se encuentran en el mismo plano, al moverse el dron, lo puntos realizan movimientos que no serían los típicos de un punto contenido en el plano del suelo y, por tanto, infiera que se está desplazando el dron en una dirección que no es.

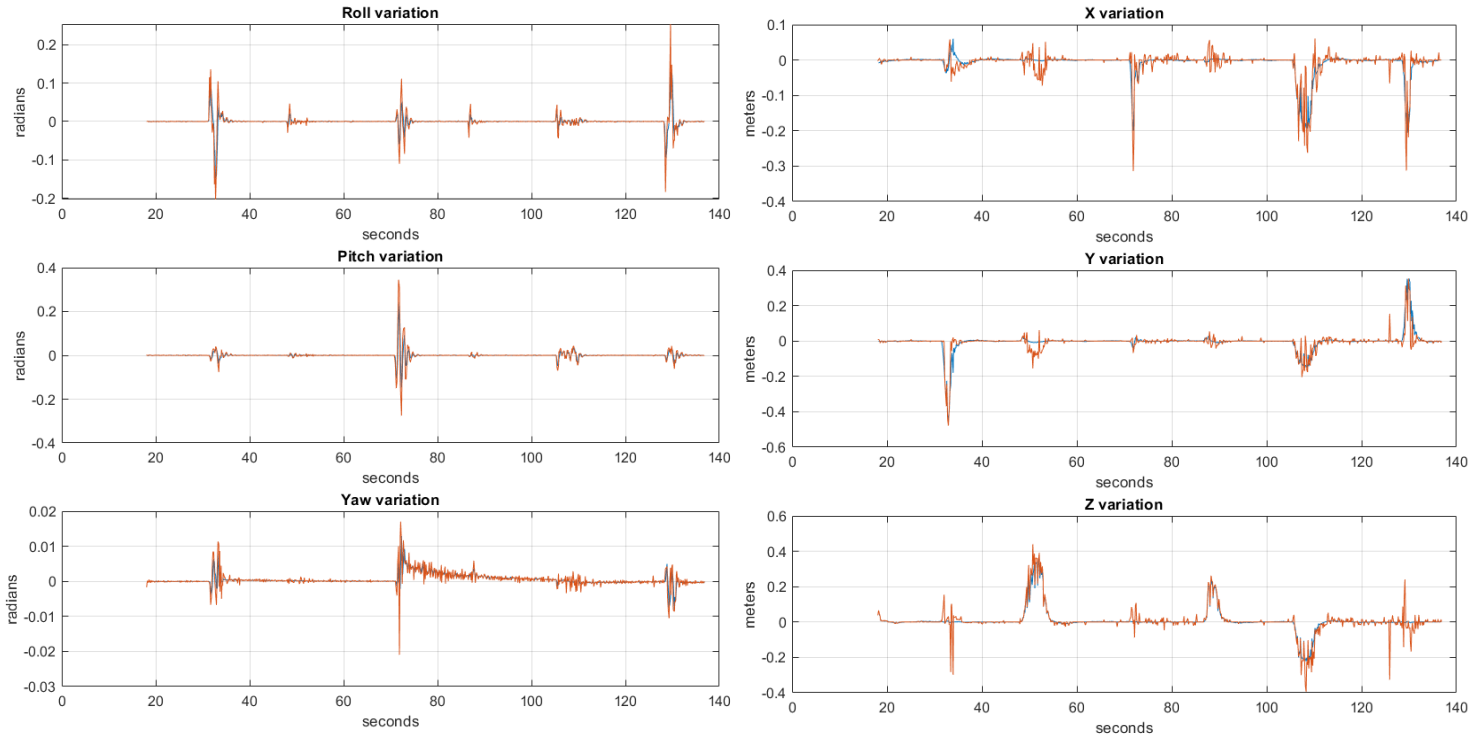

Ilustración 37. Resultados de las variaciones de posición y ángulos para la simulación del mundo a distintas alturas. En azul el valor real y, en rojo, la estimación.

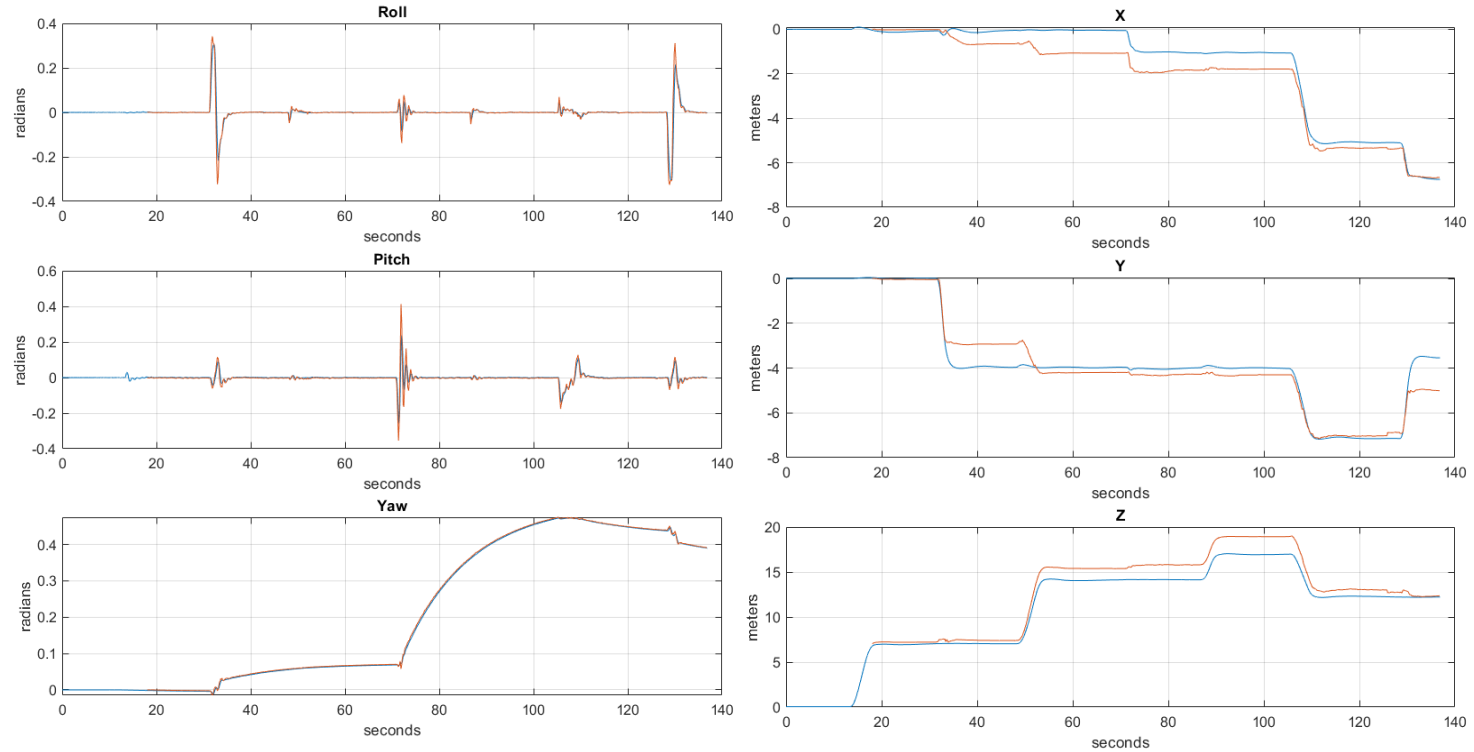

Ilustración 36. Resultados de las posiciones y ángulos estimados para la simulación del mundo a distintas alturas. En azul el valor real y, en rojo, la estimación.

7.3.2 Orientación y altura estimada

Ahora, se va a realizar la misma simulación pero realimentado a la entrada los valores de salida obtenidos. De esta forma se obtiene la Ilustración 38 y la Ilustración 39.

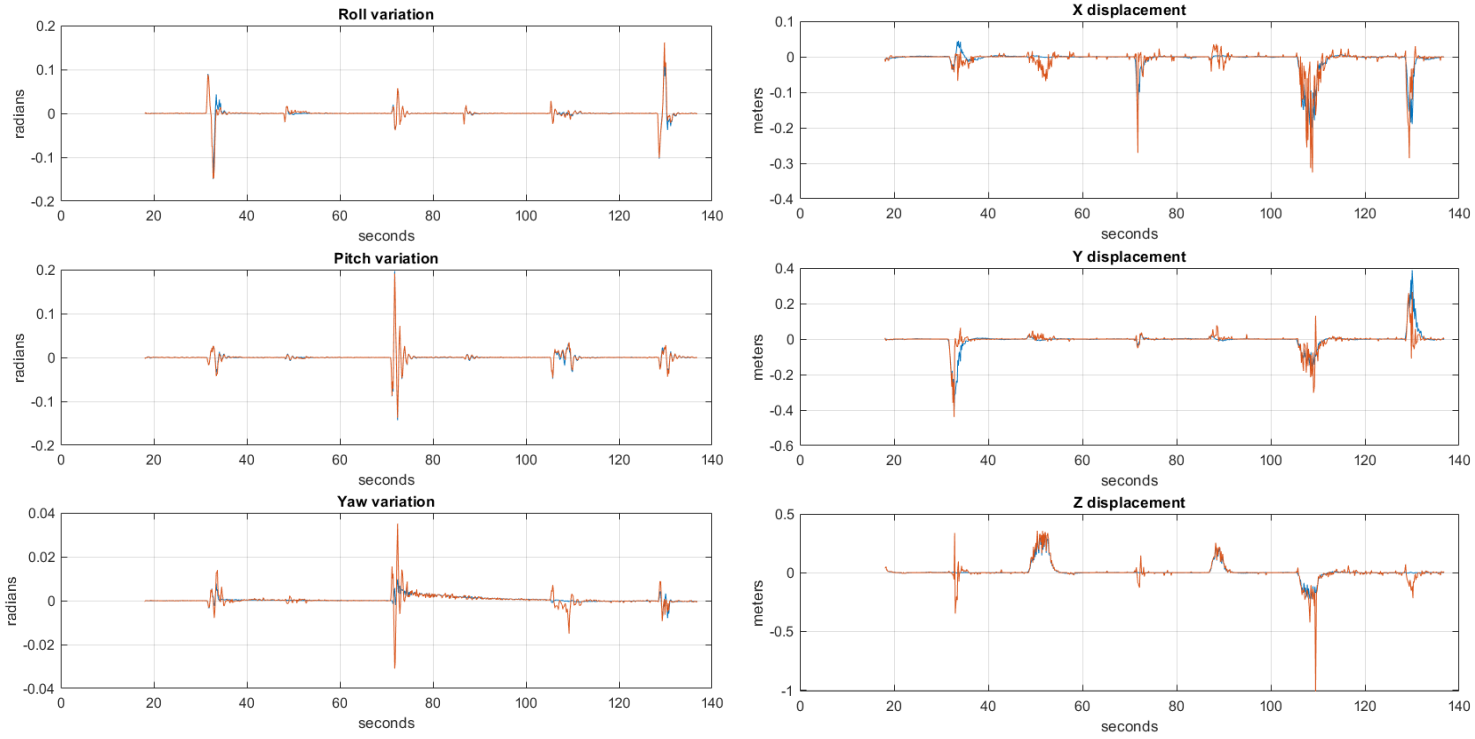

Ilustración 38. Resultados de la variación de ángulos y posición para la simulación con valores realimentados en el mundo con distintas alturas. En azul el valor real y, en rojo, el estimado.

Realizando una comparativa, en la gráfica de las variaciones destaca el error en el giro en Z, mientras que en el resto parece que no es muy apreciable. Si pasamos a la gráfica que suma todas las variaciones, se pueden apreciar de nuevo los ángulos residuales y, al final del giro en Z, la interacción entre los ángulos debido a un desacoplamiento que no es perfecto. En lo relativo al posicionamiento en XYZ, se ve una degradación de la estimación en Y, sin embargo, para el caso de X y Z este efecto apenas se aprecia, incluso pudiera parecer que la altura proporciona un valor mejor cuando se realimenta que cuando se usa el valor real.

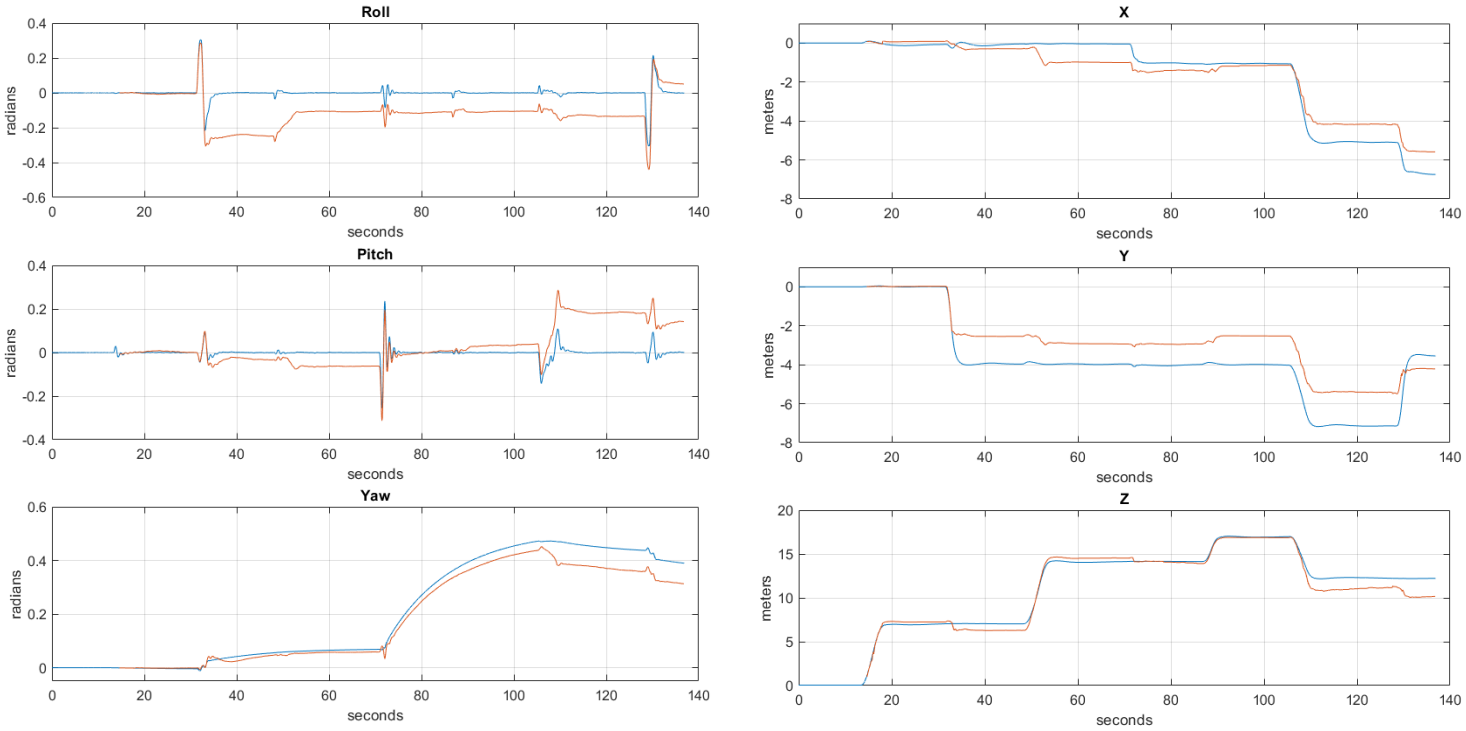

Ilustración 39. Resultados de posición y ángulos para la simulación con valores realimentados a distintas alturas. En azul el valor real y, en rojo, el estimado.

7.4 Conclusiones preliminares

Todos los resultados anteriores son los resultados de varias simulaciones llevadas a cabo, por lo que las conclusiones que se extraen a continuación tienen carácter general.

En primer lugar, no se aprecia apenas que haya un empeoramiento de las estimaciones en el caso en el que no se cumpla que todos los puntos tratados se encuentren en el mismo plano. Es decir, la nube de puntos detectados puede suponerse que se encuentran en el plano del suelo sin que haya un gran cambio en los resultados. Esto se refuerza en los resultados de la simulación para varias alturas, en la que a partir del segundo 100 se mueve por encima de un tejado y una farola tal y como se ve en la Ilustración 40. Se ve que en estos periodos se producen estimaciones no del todo exactas del desplazamiento, pero nada que no entre dentro de los errores del algoritmo que ya se produjeron incluso cuando estaban los puntos situados en el plano. Por ello parece tener robustez frente a cambios de altura.

En segundo lugar, cabe destacar que en algunos de estos casos, las malas estimaciones iniciales son compensadas posteriormente. Es decir, puesto que los desplazamientos acumulados tienen un offset, conforme se mueve el dron, estos *offsets* se van sumando y en algunos casos cancelando. Esto no es que sea algo positivo, porque no podemos saber cuándo está bien y cuando mal la medida, pero nos asegura que tendremos un mínimo conocimiento sobre la posición del UAV.

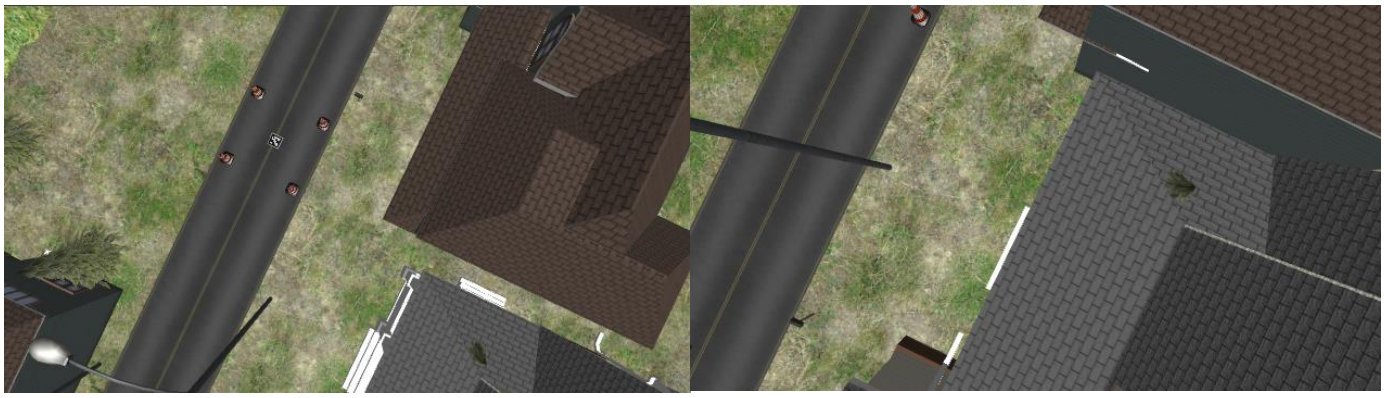

Ilustración 40. Ejemplo de dos imágenes durante la simulación en las que no se cumplen que los puntos de interés estén en un mismo plano.

Debido al carácter aditivo de este método, es decir, la posición actual se basa en la posición anterior más un desplazamiento, se dificulta la tarea de corregir errores de estimación ya que en ningún momento se corrigen las estimaciones pasadas sino que se dan directamente por buenas. Esto da lugar a errores residuales (errores que se mantienen constantes en el tiempo) que penalizan el comportamiento final del algoritmo sobre todo si ocurren en los ángulos ya que estos afectan a su vez al cambio de base de la traslación.

Es en la línea de este último comentario en la que se van a intentar realizar una serie de mejoras ya indicadas en la introducción del capítulo y que se van a simular directamente sobre el entorno con diversas alturas para así ver un comportamiento más parecido al de un caso real.

- Con la primera modificación se va a comprobar el efecto de un cambio en el algoritmo de creación de descriptores de SURF a ORB.
- La segunda, tendrá como objetivo fundamental eliminar los ángulos residuales y se basará en la suposición de que cuando el dron apenas varía sus ángulos, es porque está en posición estable horizontal.
- La tercera y última, intentará reducir el efecto acumulativo de los errores de estimación de las variaciones no comparando directamente cada fotograma con el anterior, sino con uno de referencia que irá cambiando según se aleje de la posición de referencia (en la que se tomó ese fotograma de referencia). De esta forma se intenta que los efectos sumativos se alejen en el tiempo.

7.5 Uso de algoritmo ORB

En un principio se indicó que se iba a utilizar SURF junto con el hessiano rápido, ahora bien, ¿hay alguna diferencia reseñable si se utiliza otro algoritmo de detección y descripción? En un principio se puede pensar que no ya que, al fin y al cabo, va a detectar puntos que, aunque tengan otras características, definirán de una forma parecida la imagen, por lo que las diferencias entre imágenes no deben variar mucho. Además, ambos algoritmos crean descriptores con invarianza frente a traslación y rotación. El *quid* de la cuestión está por tanto en, ¿qué diferencias hay entre algoritmos que realizan la misma tarea pero en una cantidad de tiempo diferente? Para ello se va a probar el algoritmo ORB, que es otro algoritmo usado comúnmente en visión.

El algoritmo ORB (*ORiented FAST and Rotated BRIEF*) [26] como las siglas indica, hace uso del algoritmo FAST para la detección de puntos y del algoritmo BRIEF y se caracteriza por su velocidad de ejecución.

Tras realizar diversas simulaciones con la función ORB de OpenCV, utilizando distintos parámetros tanto de número de puntos máximos detectados como la escala entre niveles de la pirámide, se ve que el comportamiento final depende en parte de cuántos puntos se quieran obtener. Usando los parámetros por defecto de ORB se consigue que en media se produzcan datos cada 0.089s frente a los 0.122s de SURF. El resultado de la simulación se muestra en Ilustración 41. Si en vez de los valores por defecto se opta por coger el doble de puntos (1000) el algoritmo para a tardar en torno a 0.14s aunque el resultado también es algo mejor (no demasiado) como se ve en Ilustración 42.

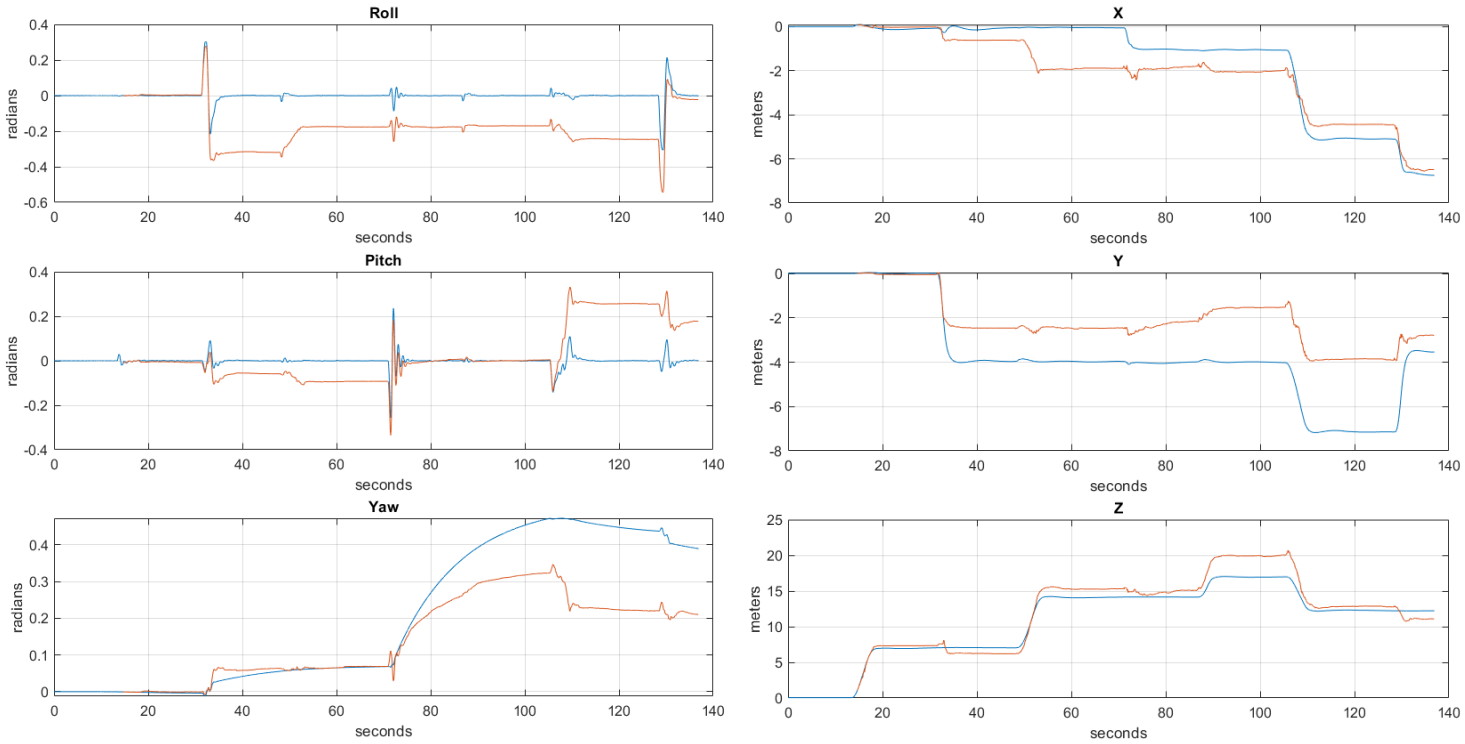

Ilustración 41. Simulación con ORB para parámetros por defecto (500 puntos y 1.2 de escala).

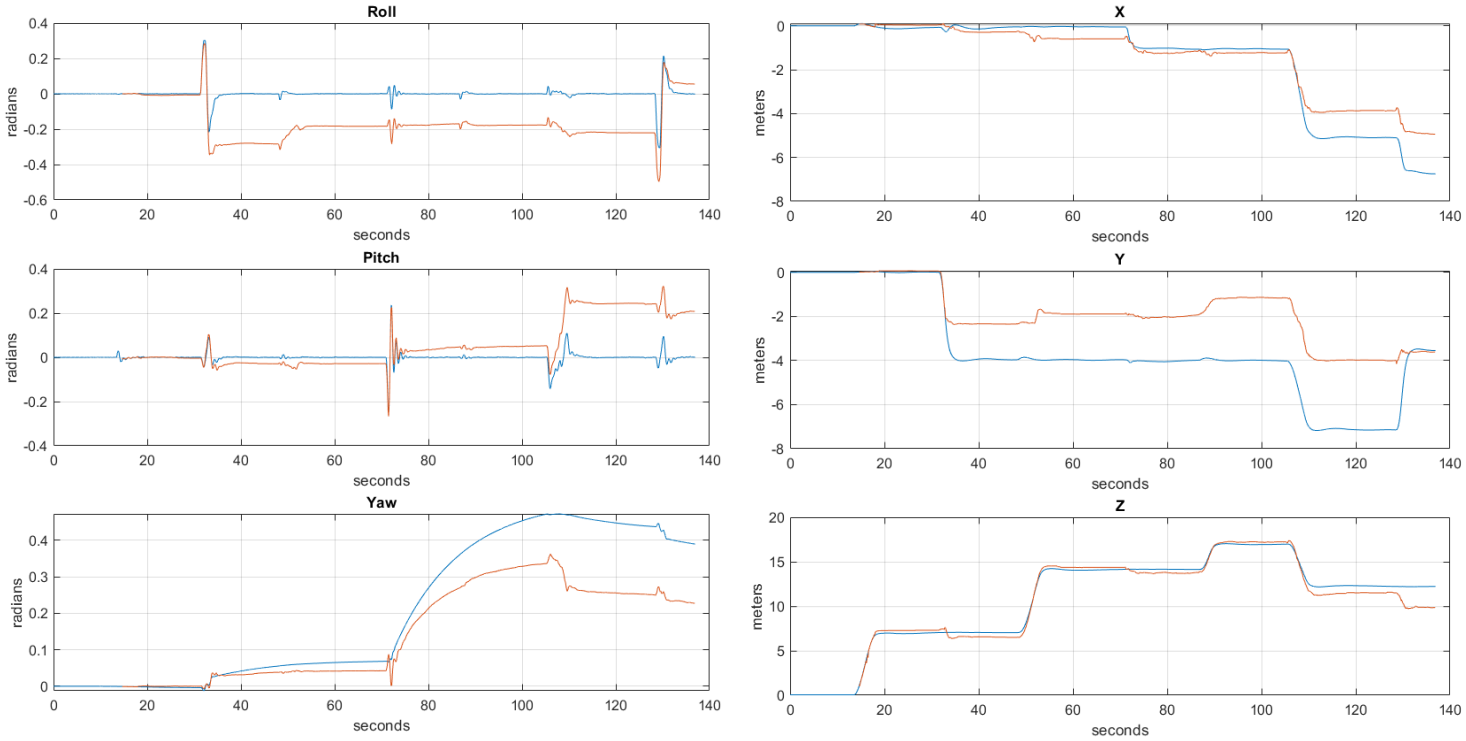

Ilustración 42. Simulación con ORB para 1000 puntos y 1.2 de factor de escala.

Sin embargo, parece que ORB presenta más problemas para detectar los giros en Z incluso cuando se le pide que tome más puntos, lo cual sorprende cuando leemos que este algoritmo es invariante frente a rotaciones. Puede ser que sea menos robusto frente a estas de lo que se pensó en un principio al leer artículos como [26] o [13]. Aun así, podría ser una alternativa a SURF si se necesitara menor tiempo de ejecución (a costa de precisión). Aunque otra opción también planteable sería la modificación de los parámetros de SURF para tener que realizar menos operaciones.

En definitiva, usar el algoritmo ORB por sí solo no soluciona los problemas como ya se indicó en un principio. Además, la mayor velocidad de ORB no aporta gran información ya que también realiza una peor estimación de los desplazamientos que es contraproducente.

7.6 Reinicio de ángulos en posición estable

Parece que uno de los principales problemas del algoritmo principal se debe a los ángulos residuales. Por ello, se propone que, puesto que el dron cuando se mueva va a tener unos giros bruscos y no constantes, si durante un determinado periodo de tiempo el dron tiene una variación en sus ángulos menor de una determinada cantidad, se supondrá que realmente está en su posición de reposo con 0 radianes en X e Y. Esto se propone porque las medidas por visión son capaces de detectar muy bien cuándo no hay movimiento. Se realiza la prueba de tal forma que si durante 15 periodos de cálculo seguidos el cambio en todos los ángulos es menor que 0.01 radianes, se supone que el dron está parado. Si pasamos a grados dicha cantidad para tener una mejor intuición de la desviación máxima que puede haber de un fotograma a otro se obtienen 0.57 grados. Es decir, suponiendo que la medida es suficientemente fiable, resulta bastante probable que si el dron está girando menos de 0.6º entre cada

fotograma (~0.15s) es porque el dron está quieto ya que en movimiento es complicado que mantenga unos ángulos en un rango de valores tan acotado. Esta suposición se haya en *mixed_with_reinit.launch* y da las siguientes gráficas:

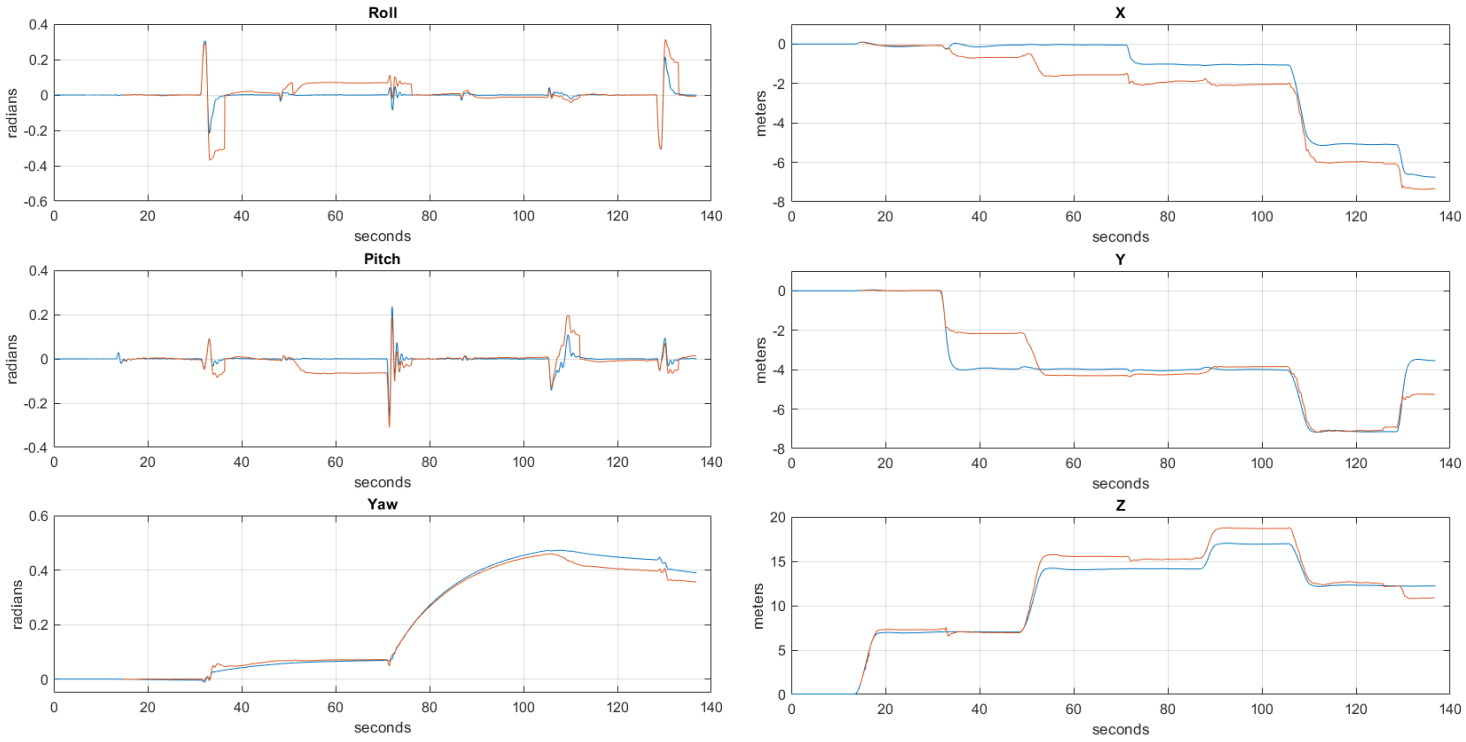

Ilustración 43. Simulación para el caso a varias alturas con reseteo de los ángulos. En azul el valor real y, en rojo, el estimado.

En la Ilustración 43 se ve cómo mejora la estimación de los ángulos obtenidos ya que, debido al reinicio, dejan de acumularse esos errores permanentes debido a sobrestimaciones o infraestimaciones de los ángulos rotados en cada momento. Sin embargo, esto no parece traducirse directamente en una mejora en la determinación de la posición. Aunque haya algunos tramos que mejoren, esto simplemente es debido a que los excesos y defectos que se van acumulando por la suma de las traslaciones anteriores se cancelan.

7.7 Homografía a partir de referencias variables

El algoritmo principal de posicionamiento desarrollado planteó el uso de lo homografía para calcular los desplazamientos de la cámara de un fotograma al fotograma siguiente, pero esto tenía un problema, se podían acumular los errores pasados aunque la estimación actual fuera buena. Por ello, se va a intentar reducir este efecto acumulador no solamente en los ángulos medidos sino también en la posición.

El nuevo planteamiento del problema se va a centrar en tomar una imagen de referencia con posición y ángulos conocidos (la inicial sería la del final del despegue) y comparar dicha imagen con los nuevos fotogramas que se vayan extrayendo hasta que los desplazamientos o ángulos calculados difieran más de una cantidad de unos límites establecidos. En dicho instante, se tomará la nueva imagen como nueva referencia y se realizará la misma tarea comentada antes con las imágenes futuras. De este modo solo habrá que sumar los efectos de movimiento entre cada referencia.

El planteamiento matemático es igual para el caso principal, solamente que en vez de ir sumando los desplazamientos continuamente a sí mismo, se sumarán a la posición referente a la referencia. De igual forma para los ángulos. Es por ello que no se va a entrar en más detalle ya que sería redundante. Lo que sí se van a indicar son los parámetros elegidos como límites de desplazamiento o de ángulos que indican un cambio de referencia.

7.7.1 Límites para el cambio de referencia.

Interesaría que los límites de movimiento estuvieran condicionados por la cantidad de puntos de la referencia que dejamos de ver. Sin embargo, esta no es una tarea que se pueda asociar a unos límites numéricos fijos como 2m o 0.5m, ya que no es lo mismo el conjunto de puntos que dejan de verse si te mueves 2m en X si el UAV está a 3m de altura a que si está a 15m de altura. Por ello, se van a obtener límites proporcionales a la altura.

Para obtener unos valores aproximados de estos límites, se va a suponer que la visión de la cámara está definida por un cono que tiene por vértice el estenopo y eje la dirección focal. La forma de este cono se aproxima de forma experimental a través de la toma de varias imágenes, obteniendo que el ángulo que forman el eje y la generatriz es de aproximadamente 23.4º. Con este ángulo y bajo la suposición de que todos los puntos están en la base ya podemos hallar la relación base/altura:

$$b = h \cdot \tan(23.4º) = 0.434 \cdot h$$

Ahora bien, ahora habría que definir esos límites relativos. Para intentar no perder demasiados puntos de una referencia a otra (lo cual afecta a la estimación), se va a poner como límite que el desplazamiento máximo sea la mitad de la base, lo que hace que, si el desplazamiento es mayor que 0.217·b se cambie de referencia. Pero claro, la base no es cuadrada, es decir, que esa base en el eje X será diferente de en el eje Y, el cual es más ancho. Usando la resolución del sensor se puede estimar cuál debe ser el desplazamiento máximo en Y bajo las mismas condiciones:

$$b' = 640/360 \cdot 0.217 \cdot h = 0.386 \cdot h$$

Finalmente, habría que limitar el desplazamiento en z, puesto que si se aleja demasiado de la referencia empieza a aparecer ruido en la estimación. Inicialmente se plantea este límite en un 30% de la altura, aunque será posteriormente ajustado si se ve un mal comportamiento.

Para limitar los ángulos, se ha optado por seleccionar este umbral de manera manual según las simulaciones previas y se ha decidido ponerlo en 0.1 radianes. Sin embargo, no se limita el valor del giro en Z, puesto que al ser esto solamente una rotación, no debería hacer que se dejaran de ver puntos (se usan en principio descriptores resistentes a rotación).

De manera adicional, se impondrá que, para evitar que algún ruido pueda llevar a un mal cambio de referencia, solamente se producirá este cambio si los desplazamientos han sido superiores al límite durante un total de 10 periodos de muestreo.

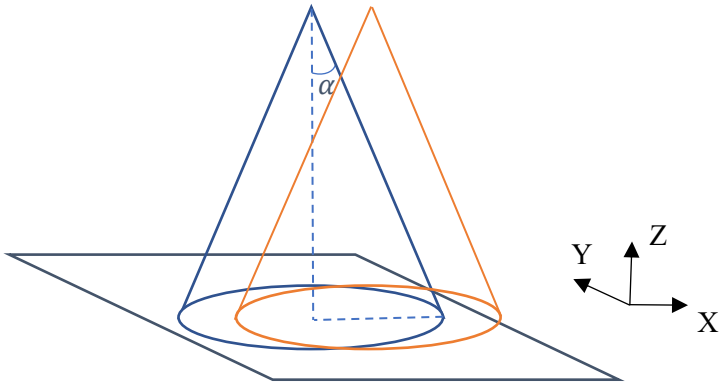

Ilustración 44. Esquema de máxima desviación antes de cambiar de referencia en X. En azul el cono de referencia y, en naranja, el cono límite. Se ve como el cono naranja sigue manteniendo un buen número de puntos en común para evitar tomar una referencia a partir de una mala estimación.

7.7.2 Simulaciones y reajuste de parámetros

Se va a comprobar el funcionamiento del método para los parámetros indicados antes y, a partir de los resultados, se buscarán aquellos posibles cambios que mejoren la estimación.

La primera estimación se muestra en la Ilustración 45 y se ve claro cómo hay un momento en el que la estimación pega en picotazo. Eso es debido a una falta de puntos de interés que lleva a una mala estimación.

Aquí se plantean dos posibles soluciones:

- Disminuir el umbral de aceptación de puntos del algoritmo SURF.
- Facilitar el cambio de referencia debido a desplazamientos en Z ya que los desplazamientos verticales de 1.5m por encima de los 10m no tienen ningún efecto más allá de empeorar (al estar más lejos) la estimación total. Es decir, el 30% establecido antes puede ser que sea excesivo.

Yendo por el primer camino, se empieza bajando el umbral a 500. Los resultados que se muestran en la Ilustración 46 son mejores que los obtenidos anteriormente y ya no aparece ese impulso. Ahora bien, ¿podría mejorarse el comportamiento si se baja el umbral hasta 100? La respuesta es que en la práctica, aparte de que se necesitaría un mayor tiempo de ejecución, la detección de demasiados puntos puede ser también negativa al detectarse puntos que no destacan tanto y que, por tanto, pueden confundirse y dar lugar a un mal emparejamiento tal y como se ve en la Ilustración 48.

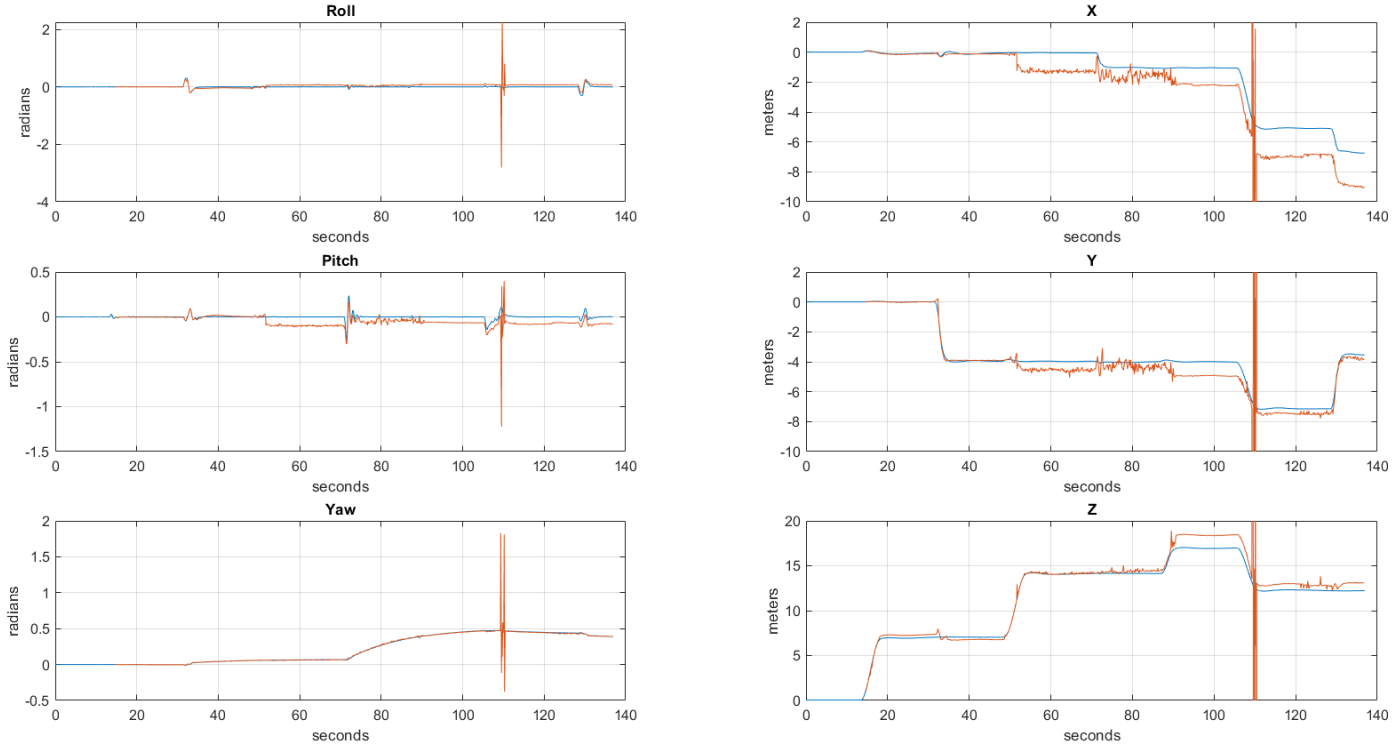

Ilustración 45. Resultado de la primera simulación de prueba (en el mundo a diferentes alturas) para la homografía con referencias variables.

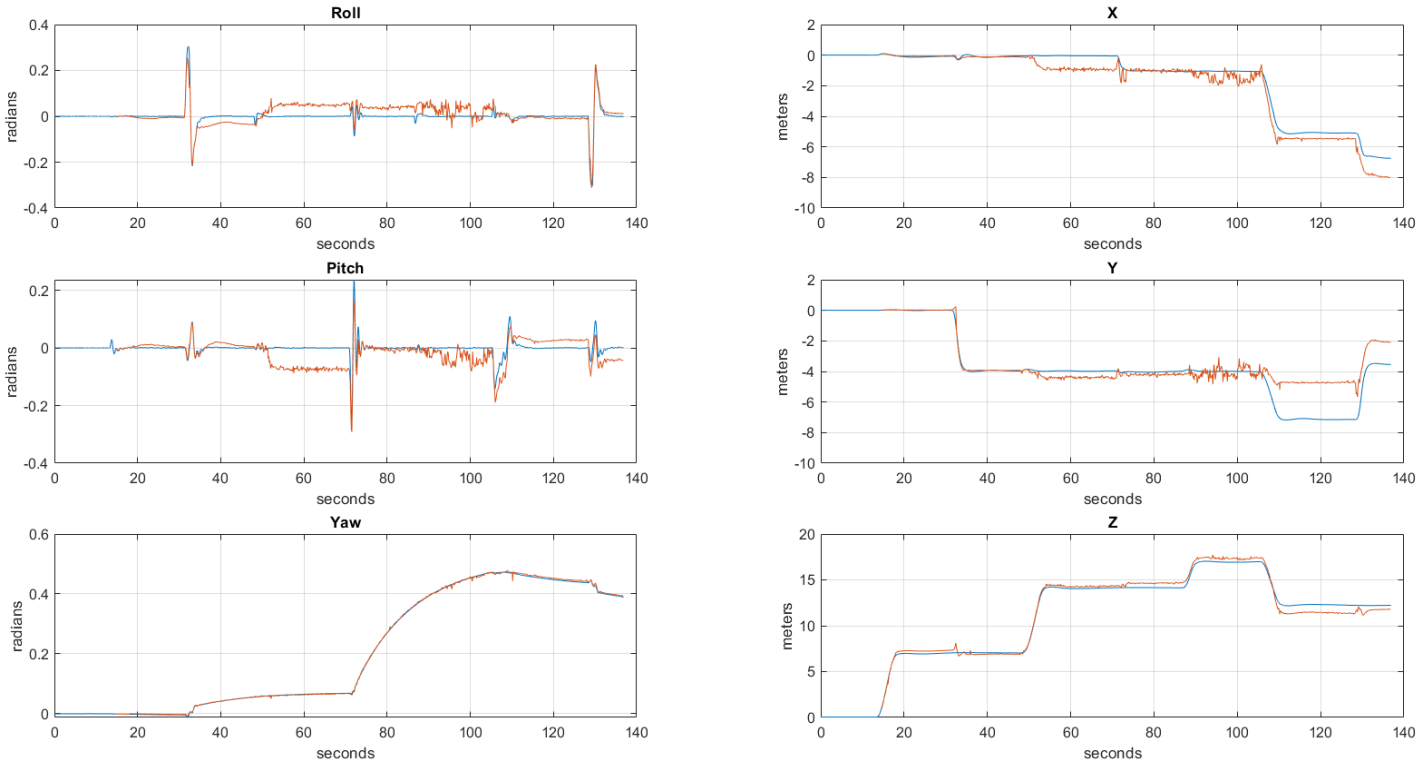

Ilustración 46. Simulación para un umbral de 500. En azul el valor real y, en rojo, el valor estimado.

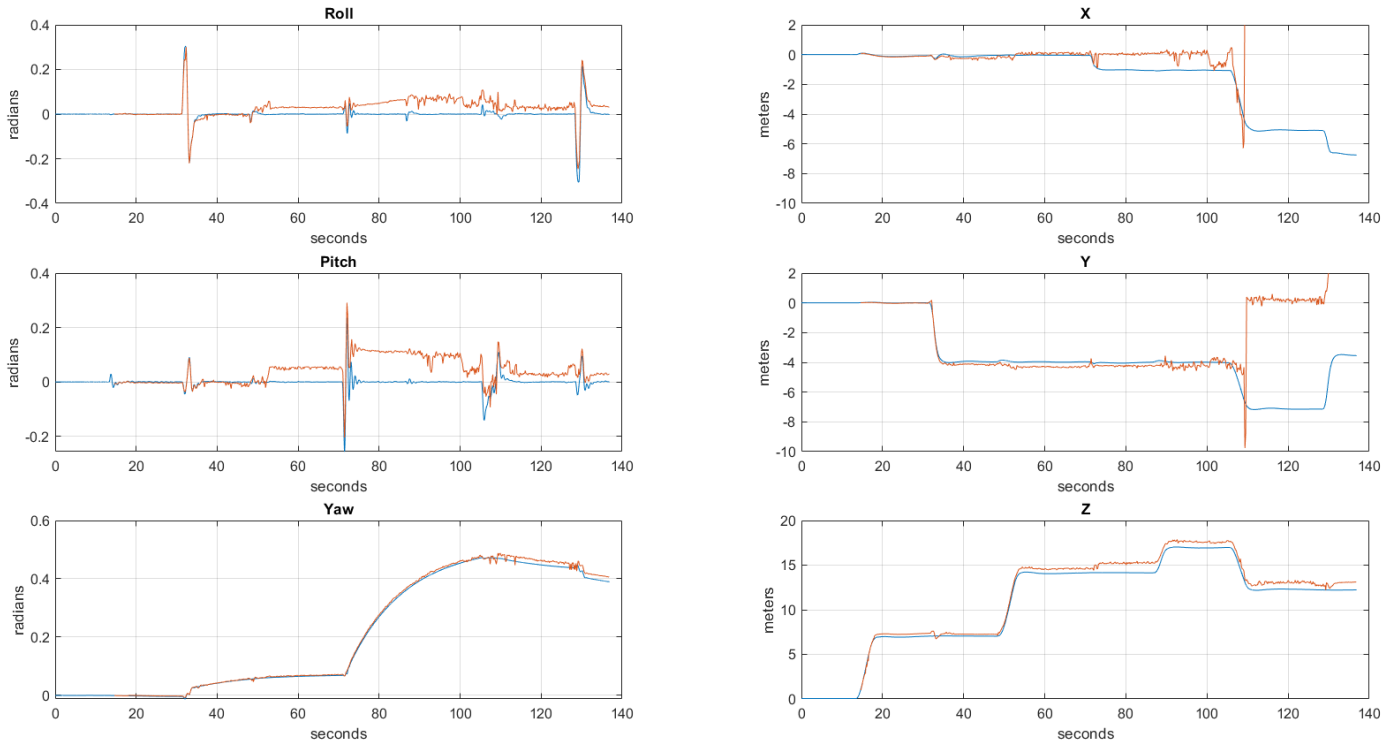

Ilustración 48. Simulación para un umbral de 100. En azul el valor real y, en rojo, el valor estimado.

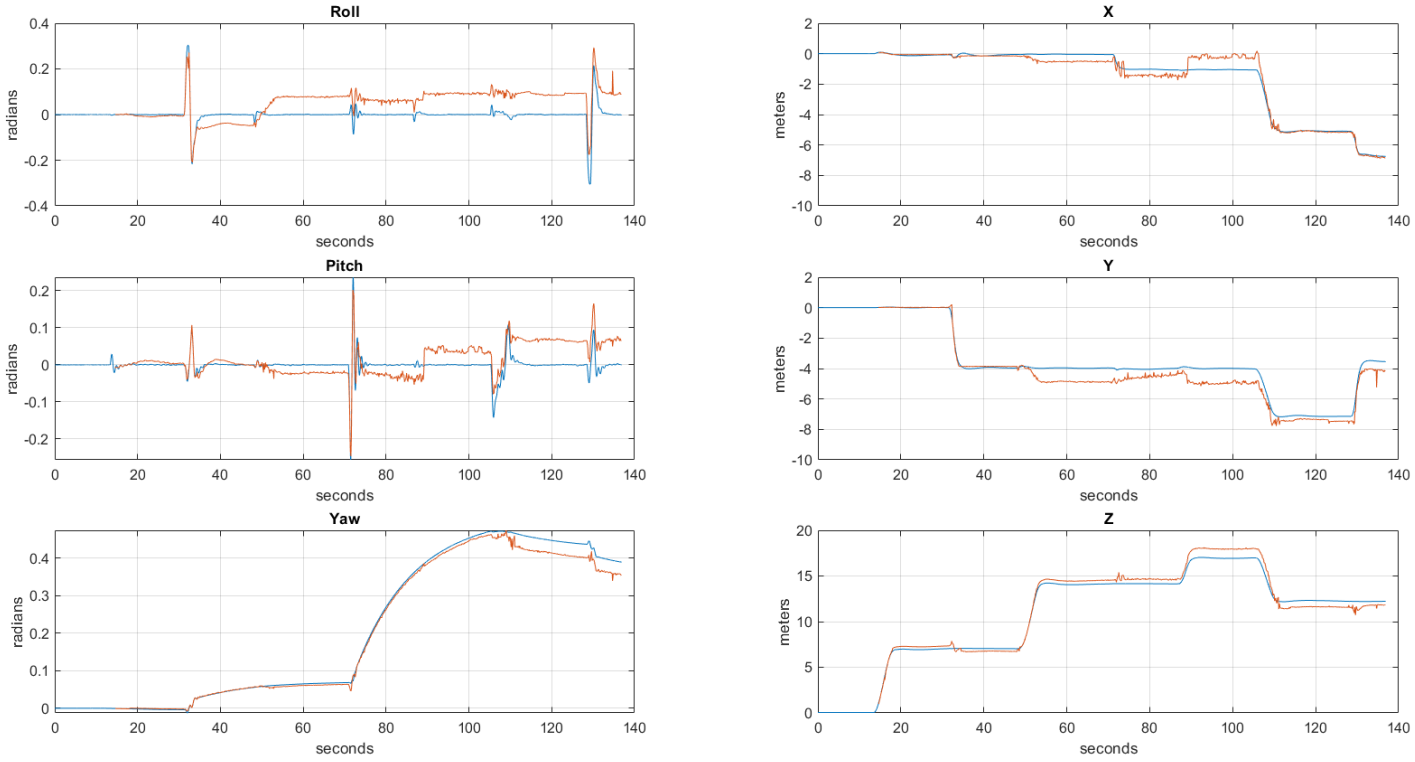

Ilustración 47. Resultado simulación para la variación de altura límite para la referencia de 1m. En azul el valor real y, en rojo, el estimado.

La otra opción sería imponer que el cambio de referencia se pueda dar también cuando el dron asciende o desciende un metro de altura, siendo así más restrictivos con respecto a las variaciones en esta dirección. Esto debería facilitar el cambio de referencia lo justo para evitar que se produzcan situaciones en las que los puntos en el nuevo fotograma sean muy diferentes de aquellos de la referencia. El resultado de posición obtenido es francamente bueno como se puede apreciar en la Ilustración 47. Si bien es verdad que presenta esos ángulos residuales que tan flaco favor hacen a la estimación. Este efecto puede tener consecuencias más a largo plazo, pero parece que a corto, no funciona tan mal.

Como tanto el uso de un umbral de 500 como la modificación de la variación límite de la altura dan buenos resultados, se ha probado a unir ambas modificaciones. Sin embargo, el resultado obtenido aunque es parecido, empeora ligeramente. Es por ello por lo que nos quedaríamos con la versión que limita a 1m la variación máxima de la altura antes de cambiar de referencia y usa un umbral para el gradiente rápido de 1000. El archivo a ejecutar sería *camera_abs_hom*.

8 FILTRO DE KALMAN EXTENDIDO PARA EL MÉTODO DE VISIÓN BÁSICO CON MARCADORES NATURALES

En el capítulo anterior se realizaron diferentes pruebas sobre algoritmos basados en marcadores naturales. Ahora se pasa a comprobar otro método diferente para intentar mejorar esa estimación de la posición basada en los desplazamientos entre fotogramas consecutivos. Este nuevo método se preguntará si la integración de las medidas procedentes de la IMU puede aportar información útil que mejore el comportamiento global de nuestro sistema de determinación de la posición y ángulos del UAV.

Para ello, recurriremos en principio a los valores del giroscopio y del acelerómetro, dejando fuera de la simulación el magnetómetro ya que, como no hay ruido electromagnético en la simulación, las medidas que este proporciona son demasiado precisas y no serían reproducibles en un entorno real que no estuviera muy controlado. Estas fuentes de información junto con la estimación proporcionada por visión (aquella basada en variaciones entre fotogramas consecutivos) serán fusionadas usando un filtro de Kalman extendido (EKF), obteniéndose así un valor de salida que debería aportar, en términos generales, unos valores mejores que al utilizar la visión solamente.

Un filtro de Kalman es una manera de implementar un filtro de Bayes, es decir, poder trabajar con las distribuciones de probabilidad de unas medidas de entrada con ruido blanco para así poder tener otra de salida. Sin embargo, el filtro de Kalman está diseñado para sistema lineales, lo cual no se corresponde con la situación que se plantea, es por eso por lo que se habla en todo momento de filtros de Kalman extendidos, que no son más que una extensión a sistemas no lineales. El filtro de Kalman extendido (*Extended Kalman Filter*) relaja las restricciones de linealidad, permitiendo que puedan usarse funciones no lineales. Sin embargo, el algoritmo por dentro lo que hace es linealizar dichas funciones en torno a los puntos de trabajo, por lo que el resultado final del algoritmo no será el valor de la distribución exacta o real, sino una aproximación de esta que se basa en este proceso de linealización. A pesar de esto, es una herramienta muy utilizada en robótica que proporciona buenos resultados incluso en casos en los que se incumplen las condiciones subyacentes como se indica en [25].

Finalmente, utilizando este algoritmo se simulará el mismo movimiento realizado en las simulaciones anteriores para el mundo con distintas alturas.

8.1 Determinación de la varianza de la medida de visión

En primer lugar, es necesario para el filtro extendido de Kalman proporcionar no solo el valor medido, sino también la varianza de este para así poder tener una distribución de probabilidad de entrada. Para la IMU no hay problema porque esos valores ya vienen enviados en el mismo mensaje, pero nos faltaría determinar la de nuestro algoritmo de visión.

Existen diferentes formas de calcularla. Una opción es hacer una varianza móvil a partir de las medidas obtenidas durante los n últimos periodos. Sin embargo, como nuestro algoritmo de visión ya es de por sí algo lento a la hora de ejecutarse (0.13 s/muestra), estos cálculos podrán atrasarlo más. Además, cuando el UAV se esté moviendo, los valores de varianza que se obtendrán serán bastante elevados porque, aunque tomemos una ventana pequeña, el desplazamiento en ese tiempo será bastante grande. En resumen, como el tiempo de ejecución es demasiado grande, se obtendrían varianzas grandes (lo que equivale a una mayor incertidumbre) y esa varianza excesivamente grande no aportará mucho al filtro.

Otra opción que no sería más que una estimación (no se buscaría obtener el valor real sino un valor parecido) sería determinar una varianza estática a partir de los errores. Bajo el supuesto de que las estimaciones de visión no tienen ningún offset, es decir, en media coinciden con el valor real (como se vio en la Ilustración 38), puede suponerse que cualquier desviación con respecto a la traslación real entra dentro de esta varianza de la estimación. Además, se supone también que el error de la estimación por visión tiene media nula. Es decir, el valor medio de las variaciones de posición y ángulos debe coincidir con el valor real al no haber offset y, como el error tendría media nula, la varianza de la medida de visión coincidiría con la varianza del error de las variaciones. De manera offline se realiza una simulación diferente con movimientos variados, obteniéndose así los valores mostrados en la Tabla 6.

X [m^2]	5e-5
Y [m^2]	1e-4
Z [m^2]	2e-6
Roll [rad^2]	4e-3
Pitch [rad^2]	1e-3
Yaw [rad^2]	4e-3

Tabla 6. Varianzas estimadas para la visión.

8.2 Parámetros para el EKF

El EKF será implementado a través del paquete *robot_localization* de ROS. Este paquete recibe mensajes de la IMU (la cual tiene incluida la varianza) y de la visión y los fusiona para obtener una estimación única. Una vez estimada la varianza de la visión, ya se podrían introducir dichos valores al nodo de EKF. Sin embargo, viene bien comentar los parámetros que se pueden ajustar.

En primer lugar, hay que indicar los elementos a fusionar de cada sensor. Por un lado, se fusionarán las variaciones de ángulos y posiciones aportadas por visión, que se dividirán entre el tiempo pasado para obtener velocidades (ya que el paquete funciona con posiciones o velocidades). Por el otro, la IMU proporciona varias medidas: magnetómetro, giroscopio y acelerómetro. El magnetómetro da un resultado prácticamente idéntico al real al no haber ruido electromagnético en la simulación, pero en la realidad no es así, por lo que para intentar asemejarnos más a la realidad no se va a tener en cuenta esta medida para el cálculo del EKF (aunque en el sistema real debería hacerse). El giroscopio será utilizado pero con la salvedad de aplicar las correcciones propias de la calibración indicada en la Tabla 3. Finalmente, para el acelerómetro se van a realizar un par de pruebas para comprobar si vale la pena o no introducirlo en nuestro caso ya que de por sí solo la estimación aportada es pésima.

Puesto que cada 0.13s de media se obtiene una medida por visión, se ha indicado una frecuencia de 7Hz para la publicación de datos por el nodo que realiza el EKF. Este nodo podrá almacenar los 8 últimos mensajes de visión y los 40 últimos de la IMU.

También habría que ajustar la matriz de incertidumbre inicial (como conocemos la posición inicial deberían ser valores bajos) y la matriz de ruido que se añade tras cada estimación (la cual se deja sin tocar en principio).

8.3 Resultados obtenidos

Inicialmente, se va a probar a fusionar lo mencionado anteriormente con el acelerómetro. Para ello se ejecuta *mixed_with_EKF6b*. Este resultado se muestra en la Ilustración 49, en donde se ve claro que parece haber una deriva en la dimensión Z incluso tras haber realizado la calibración que, recuérdese, consistía en calcular la media de medidas durante 3min estando el dron en el suelo. A pesar de intentar resolver este problema aumentando el *offset* de la medida en Z, el resultado no varía, por lo que parece que esa salida se debe a la fusión de la medida del acelerómetro con la imagen, no siendo este un resultado del todo deseable.

Una posible causa es que como los ángulos no coinciden con los reales, no se elimina bien la gravedad. Como el acelerómetro te da los valores en los ejes de la IMU, para cancelar la gravedad (que en reposo tiene un valor de +9.81 en Z en la simulación) es necesario cambiar de base IMU a base global y, ahí ya sí, eliminar la gravedad. Posteriormente se devuelve a la base IMU para pasarle ese mensaje a *robot_localization*. Si no se cancela bien la gravedad, el resultado será que quedará una gravedad residual en Z de carácter positivo ya que al descomponer +9.81 en los ejes globales, parte de ese 9.81 se divide entre los ejes X e Y debido a la imprecisión. Al no cancelar completamente la gravedad, se obtiene una deriva positiva que es la que probablemente se ve.

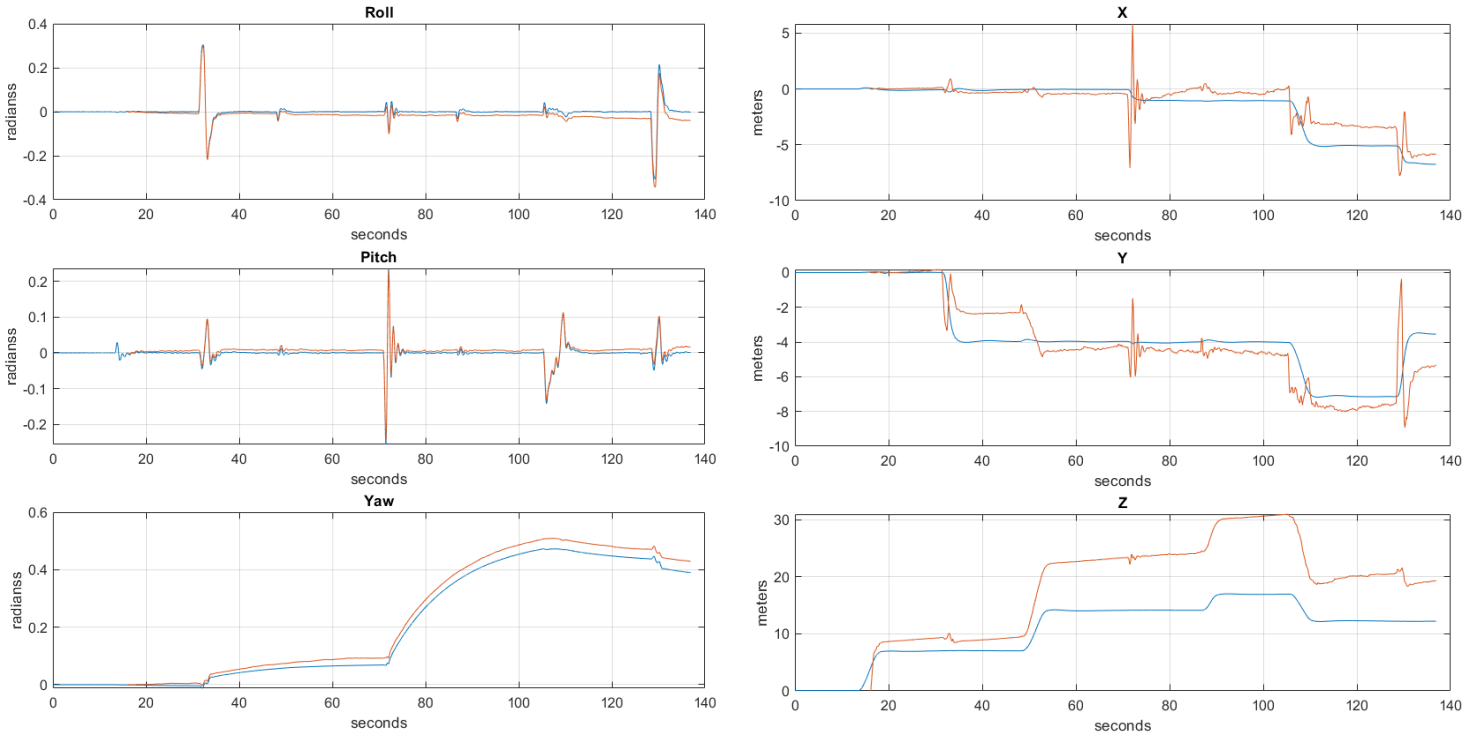

Ilustración 49. Simulación para un EKF con acelerómetro. En azul el valor real y, en rojo, el valor estimado.

Si no se fusionara el acelerómetro, habría que ejecutar el programa *mixed_with_EKF6.launch* con el que se obtendrá la Ilustración 50. Esta imagen al igual que la anterior muestra un comportamiento mucho mejor para los ángulos que el uso único de visión, eliminándose así parte de esos ángulos residuales.

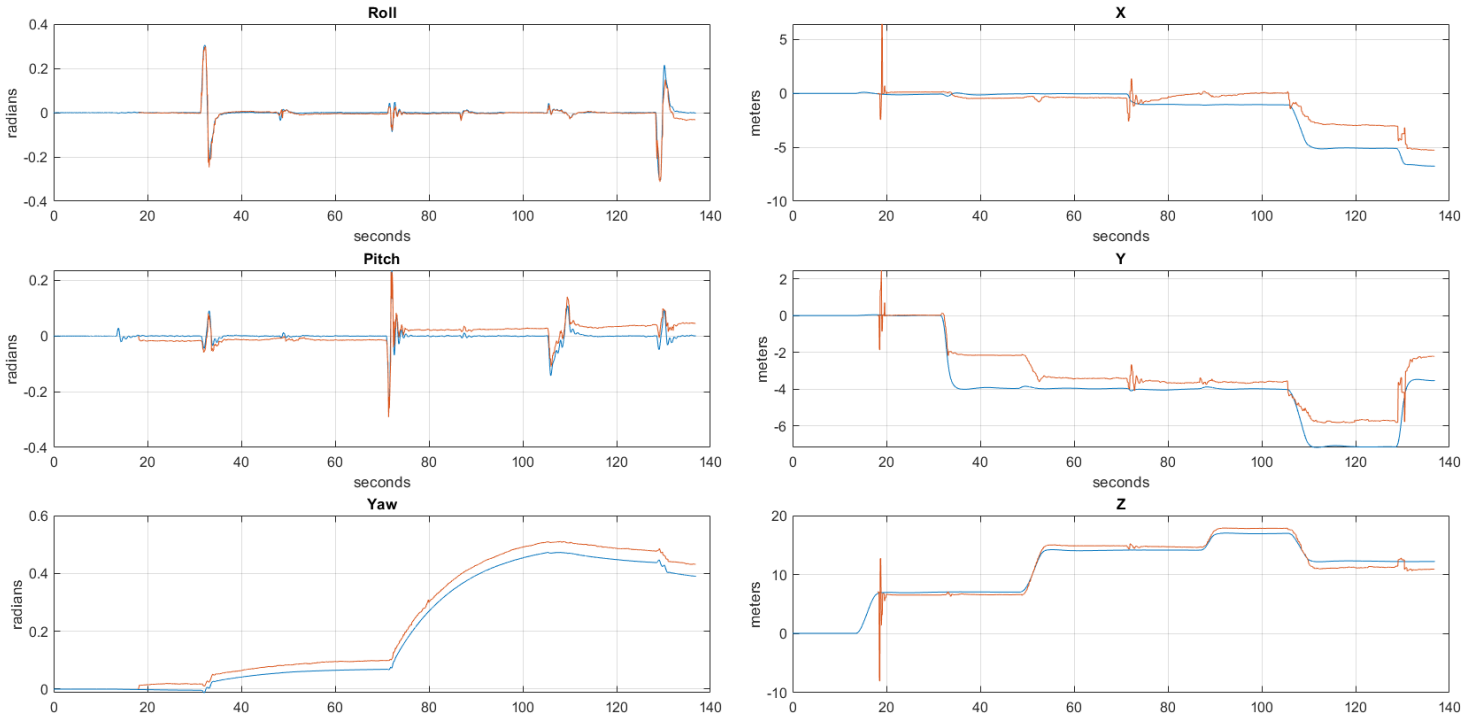

Ilustración 50. Simulación para el EKF sin acelerómetro. En azul el valor real y, en rojo, el valor estimado.

Con respecto a las posiciones de la fusión sin acelerómetro, se ve cómo ya no existe ese error en Z ni tampoco aparecen esas oscilaciones tan bruscas que aparecían antes. La causa de estas oscilaciones era probablemente la diferencia entre las estimaciones proporcionadas por acelerómetro y visión, que harían al sistema ir de un valor a otro hasta estabilizarse. Sin embargo, en la figura sin acelerómetro, hay un pico inicial. Esto es porque mientras que para el código con acelerómetro se ha incorporado al filtro de Kalman la parte del despegue (marcadores ArUCo) para evitar que se acumulase error del acelerómetro, en este caso, al no haber acelerómetro, el EKF ha empezado a recibir información justo al terminar el despegue. Ha pasado un rato sin recibir datos, por lo que ha ido acumulando la incertidumbre de la medida. Al pasar de no recibir nada y no tener información previa, a tener información, el algoritmo ha tardado unos segundos en llegar a unos valores estables. Puede preguntarse entonces, por qué no se ha metido en este caso la estimación de ArUCo en el EKF. Se ha optado por no meter la parte del ArUCo al filtro porque ya da buenos resultados de por sí y proporciona una muy buena estimación de la posición inicial prácticamente sin error, por lo que no se hacía necesaria ninguna modificación de estos valores.

Los resultados de posición obtenidos para el EKF sin acelerómetros no son perfectos, pero podrían utilizarse en el caso de que las distancias fueran menores o si no se desea un 100% de precisión sino una idea orientativa del movimiento. Con respecto a los ángulos el resultado obtenido es bastante bueno porque, como ya se comentó para el giroscopio, una vez calibrado proporciona a medio/corto plazo resultados bastantes semejantes a los reales.

9 CONCLUSIONES Y TRABAJOS FUTUROS

El objetivo de este proyecto, recordemos, era utilizar la información proveniente de la cámara inferior de dron ErleCopter para realizar estimaciones tanto de su posición como de los ángulos. En este proyecto se han estudiado diferentes métodos que buscan alcanzar tal objetivo y, aunque las simulaciones han sido realizadas con el ErleCopter, los algoritmos y metodologías aplicadas no son exclusivas de este, siendo posible su uso para otros drones.

Por un lado, los marcadores absolutos ArUCo se han utilizado para este problema proporcionando unos resultados bastante pobres de por sí que solamente mejoraron al complementar la información de la orientación con aquella procedente de la IMU. Aun así, se llega a la conclusión de que, en el momento en que se aleja demasiado de ellos, dejan de proporcionar información de calidad. Además, no solamente influye que no ae aleje mucho de ellos, sino que deben estar en todo momento dentro del rango de visión del UAV, lo cual limita bastante las posibilidades de movimiento del dron. Por ello, no es recomendable su uso para este tipo de casos a no ser que sea un entorno limitado y controlado como sería el caso del despegue del dron, el cual estaría diseñado para extraer el máximo partido posible a estos métodos de referencias absolutas.

Por otro lado, el enfoque principal del proyecto es el intento de realizar ese posicionamiento de manera relativa, sin necesidad de conocer objetos o marcadores externos (aunque sabiendo unas condiciones iniciales). Este enfoque ha sido tratado de diversas maneras intentando siempre mejorar los resultados obtenidos.

El método incremental basado en las variaciones de fotograma a fotograma presenta claros problemas de ángulos residuales y una incapacidad de corregir sus medidas, por lo que los errores cometidos pueden acumularse incluso sin llevar demasiado tiempo de simulación. Este problema es a simple vista difícil de solucionar puesto que es algo natural del algoritmo y que surge de no saber el valor exacto de la traslación en cada momento, por lo que es esperable esta clase de offset entre los valores reales y los estimados. Sin embargo, esos mismos errores que pueden hacer que se sobreestime una medida, pueden actuar posteriormente para subestimar la traslación siguiente y volver al valor correcto. Esto último es pura casualidad, pero ocurre a menudo en las simulaciones realizadas. Este método no presenta una gran fiabilidad, pero al menos da unos valores que no son desproporcionados.

Además, se ha probado a usar también algún otro algoritmo de detección y caracterización de puntos, obteniéndose unos peores resultados si bien la frecuencia de estimación por visión mejoraba. No parece que esta línea vaya a aportar mucho más al funcionamiento final.

Para intentar mejorar ese resultado obtenido (sobre todo los ángulos residuales) se ha propuesto que si el dron pasa más de 15 periodos con una variación de los ángulos menor a 0.5º, se supondrá que el dron se encuentra en realidad en posición horizontal y los ángulos que teníamos estimados no eran más que errores acumulados. Este algoritmo se basa en estimar que un dron en funcionamiento continuo, por mucho que esté en movimiento, será incapaz de mantener unas variaciones en los ángulos de giro en X e Y menores de 0.5º durante 15 muestras seguidas. Sin embargo, tampoco ha aportado muchas mejoras con respecto a la posición en XYZ.

Otra idea que se ha utilizado es intentar eliminar una de las fuentes de error de la idea original: la integración continua de desplazamientos. Para ello, se ha planteado un modelo a medio camino entre los marcadores relativos y los marcadores absolutos. Se ha utilizado un algoritmo que utiliza una imagen de referencia mientras que no se aleje demasiado de ese punto. Si se alejase demasiado, se guardaría la variación de posición y ángulos anteriores y se tomaría la nueva imagen como la nueva referencia. Esto presenta como ventaja que, aunque también estemos sumando desplazamientos, en este caso se realiza sobre referencias, por lo que se pospone en el tiempo esa acumulación de errores. Además, permite filtrar aquellas estimaciones que sean *outliers* y que puedan dar problemas ya que, aunque den un valor desproporcionado en un punto, si no continua la perturbación en el instante siguiente se volverá a una medida normal. Este algoritmo proporciona unos resultados que a simple vista mejoran aquellos del sugerido inicialmente. Es por ello que, en caso de aplicarse en la realidad, esta sería una opción a barajar.

Por último, también se ha probado a utilizar un filtro de Kalman extendido para fusionar las medidas provenientes de la IMU con aquellas estimaciones de visión realizadas con el programa original basado en marcadores naturales y variaciones entre fotogramas consecutivos. Para las simulaciones utilizadas se han fusionado el giroscopio solo y el giroscopio y acelerómetro a la vez. La fusión del giroscopio proporcionó mejores resultados que la que incluía el acelerómetro. En la realidad se recomendaría añadir también el magnetómetro, el cual se omitió al dar unos valores en la simulación que no serían reproducibles en la realidad. El uso de un EKF entre la visión y el giroscopio sería otra alternativa a implementar en caso en caso de realizarse experimentos.

En resumen, en caso de que se quisieran implementar estos algoritmos en un sistema real se recomendaría, en primer lugar, el despegue de UAV basándose en referencias absolutas. Una vez finalizado, se podría utilizar el método de visión por marcadores naturales con referencias variables según el desplazamiento o el método que incorpora el filtro de Kalman extendido ya que estos son los que mejores resultados han proporcionado.

En línea con este proyecto se proponen diferentes vías de continuación en el estudio del posicionamiento de un UAV mediante métodos de visión:

- Intentar realizar el EKF con la estimación proveniente del método híbrido en vez del incremental.
- Reproducir en el mundo real estas simulaciones y comparar resultados.
- Estudiar el efecto que puede tener la existencia de objetos en movimiento en la imagen y buscar formas de solventarlo.
- Fusionar estas mediciones de visión e IMU con fuentes de datos absolutos como una señal GPS.
- Intentar controlar la posición del dron basándose en la referencia proporcionada por alguno de los algoritmos anteriores.

10 REFERENCIAS

[1] V. M. Mondéjar Guerra, "Contribuciones a la estimación de pose de cámara," 2016, [Online]. Available: http://helvia.uco.es/handle/10396/13856.

[2] M. B. Alatise and G. P. Hancke, "Pose estimation of a mobile robot based on fusion of IMU data and vision data using an extended kalman filter," *Sensors (Switzerland)*, vol. 17, no. 10, Oct. 2017, doi: 10.3390/s17102164.

[3] G. Bleser and D. Stricker, "Advanced tracking through efficient image processing and visual-inertial sensor fusion," *Comput. Graph.*, vol. 33, no. 1, pp. 59–72, 2009, doi: 10.1016/j.cag.2008.11.004.

[4] T. Tumurbaatar and T. Kim, "Comparative study of relative-pose estimations from a monocular image sequence in computer vision and photogrammetry," *Sensors (Switzerland)*, vol. 19, no. 8, p. 1905, Apr. 2019, doi: 10.3390/s19081905.

[5] E. López *et al.*, "A multi-sensorial simultaneous localization and mapping (SLAM) system for low-cost micro aerial vehicles in GPS-denied environments," *Sensors (Switzerland)*, vol. 17, no. 4, Apr. 2017, doi: 10.3390/s17040802.

[6] L. Joseph, *Robot Operating System for Absolute Beginners*. Apress, 2018.

[7] A. Ahmadi Tazehkandi, *Computer Vision with OpenCV 3 and Qt5*, 1st ed. 2018.

[8] S. Brahmbhatt, *Practical OpenCV*. .

[9] E. Oyallon and J. Rabin, "An Analysis of the SURF Method," *Image Process. Line*, vol. 5, pp. 176–218, Jul. 2015, doi: 10.5201/ipol.2015.69.

[10] D. G. Low, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vis.*, pp. 91–110, 2004, [Online]. Available: https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf.

[11] D. Huttenlocher, "Computer vision," in *Computer Science Handbook, Second Edition*, 2004, pp. 43-1-43–23.

[12] M. S. Nixon and A. S. Aguado, "Low-level feature extraction (including edge detection)," in *Feature Extraction and Image Processing for Computer Vision*, Elsevier, 2020, pp. 141–222.

[13] P. Loncomilla, J. Ruiz-del-Solar, and L. Martínez, "Object recognition using local invariant features for robotic applications: A survey," *Pattern Recognit.*, vol. 60, pp. 499–514, Dec. 2016, doi: 10.1016/j.patcog.2016.05.021.

[14] I. Manuel Esteban Poot, "Desarrollo de un sistema de visión computacional para el vuelo autónomo de un UAV."

[15] E. Malis and M. Vargas, "Deeper understanding of the homography decomposition for vision-based control," *Sophia*, vol. 6303, no. 6303, p. 90, 2007, doi: 10.1126/science.318.5857.1691b.

[16] J. C. Wilcox, "Vector representation of finite rotations," *J. Guid. Control. Dyn.*, vol. 11, no. 4, p. 381, 1988, doi: 10.2514/3.56469.

[17] G. Nützi, S. Weiss, D. Scaramuzza, and R. Siegwart, "Fusion of IMU and vision for absolute scale estimation in monocular SLAM," *J. Intell. Robot. Syst. Theory Appl.*, vol. 61, no. 1–4, pp. 287–299, 2011, doi: 10.1007/s10846-010-9490-z.

[18] D. Eberly, "Euler angle formulas," *Geom. Tools, LLC, Tech. Rep.*, pp. 1–21, 2008, [Online]. Available: https://www.geometrictools.com/.

[19] G. G. Slabaugh, "Computing Euler angles from a rotation matrix," *denoted as TRTA Implement. from httpwww starfireresearch comservicesjava3dsamplecodeFlorinE ulers html*, vol. 6, no. 2000, pp. 1–6, 1999, [Online]. Available: http://gregslabaugh.name/publications/euler.pdf.

[20] J. L. Blanco Claraco, "A tutorial on SE(3) transformation parameterizations and on-manifold optimization," 2017. [Online]. Available: https://w3.ual.es/personal/jlblanco/.

[21] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognit.*, vol. 47, no. 6, pp. 2280–2292, Jun. 2014, doi: 10.1016/j.patcog.2014.01.005.

[22] R. Ragel De la Torre, "Entorno de simulación ROS/Gazebo," 2010, pp. 65–118.

[23] D. C. Popescu, M. O. Cernaianu, P. Ghenuche, and I. Dumitrache, *An assessment on the accuracy of high precision 3D positioning using planar fiducial markers*. 2017.

[24] D. C. Popescu, M. O. Cernaianu, and I. Dumitrache, "Automatic rough alignment for key components in laser driven experiments using fiducial markers," in *Journal of Physics: Conference Series*, Sep. 2018, vol. 1079, no. 1, doi: 10.1088/1742-6596/1079/1/012013.

[25] D. Thrun, Sebastian; Burgard, Wolfram; Fox, "Probabilistic robotics," 2005. [Online]. Available: http://portal.acm.org/citation.cfm?doid=504729.504754.

[26] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 2564–2571, 2011, doi: 10.1109/ICCV.2011.6126544.

[26] Docs.opencv.org. 2020. Opencv: Basic Concepts Of The Homography Explained With Code. [online] Available at: <https://docs.opencv.org/master/d9/dab/tutorial_homography.html> [Accessed 4 July 2020].

[27] GitHub. 2020. Methyldragon/Ros-Sensor-Fusion-Tutorial. [online] Available at: <https://github.com/methylDragon/ros-sensor-fusion-tutorial> [Accessed 4 July 2020].

[28] Docs.ros.org. 2020. *Robot_Localization Wiki — Robot_Localization 2.6.8 Documentation*. [online] Available at: <http://docs.ros.org/melodic/api/robot_localization/html/index.html> [Accessed 4 July 2020].

[29] Ros.org. 2020. REP 103 -- Standard Units Of Measure And Coordinate Conventions (ROS.Org). [online] Available at: <https://www.ros.org/reps/rep-0103.html> [Accessed 4 July 2020].

[30] Ros.org. 2020. REP 105 -- Coordinate Frames for Mobile Platforms (ROS.Org). [online] Available at: <https://www.ros.org/reps/rep-0105.html> [Accessed 4 July 2020].

11 ANEXOS

A) Instrucciones para la instalación del entorno para la simulación del Erle-Copter

1. Introducción

Para la realización de este trabajo se van a realizar simulaciones con el objetivo de ver el funcionamiento aproximado que tendrían cada uno de los algoritmos implementados. Para realizar dichas simulaciones será necesario instalar una serie de paquetes que venían explicados en la página web de la propia empresa ErleRobotics. Sin embargo, dicha empresa cerró en julio de 2019 por lo que tanto la página web como los foros han desaparecido. Por ello, se ha optado por recurrir a otros documentos que indican cómo realizar esa tarea.

2. Instalación de programas y entorno de trabajo

Tras realizar bastantes búsquedas en internet se ha llegado a un trabajo fin de grado [1] en el que se explica cada paso a seguir. Además, dentro de las referencias del mismo se encuentra un repositorio de Github de la propia empresa en el que también se explica parte del proceso de instalación [2]. Sin embargo, puesto que la información de esas páginas no es completa, se recomienda recurrir a un archivo de internet como [3] en el que tienen una copia de la página web antes de que cerrara, pudiendo así acceder a los tutoriales de instalación de los programas, del entorno y a las pruebas de funcionamiento. Aun así, pueden surgir problemas durante el proceso de instalación, por lo que se recomienda mirar [1] y [2] en caso de que [3] dé problemas. Por si este enlace desapareciera también, se expone el proceso a continuación:

Antes que nada, se recomienda utilizar la versión de Ubuntu 14.04 de 64 bits ya que algunos de los paquetes posteriores como ROS Indigo pueden no ser compatibles con versiones posteriores de Ubuntu. Por ello, se pide que se descarguen las versiones especificadas de cada programa.

En primer lugar, se instalan los paquetes base, las dependencias para MAVProxy y MAVProxy. El -y de algunos de los comandos indica que se dirá que sí a todas las preguntas de confirmación (por ejemplo, para descargar un archivo) que surjan. Hay que tener en cuenta que en el código inferior si hay un guion al final de una línea y pasa a la línea siguiente es como si no hubiera ningún espacio y las palabras que une el guion van unidas y con el guion:

```
sudo apt-get update
sudo apt-get install gawk make git curl cmake -y
sudo apt-get install g++ python-pip python-matplotlib python-serial python-wxgtk2.8 python-scipy python-opencv python-numpy python-pyparsing ccache realpath libopencv-dev -y
sudo pip install future
sudo apt-get install libxml2-dev libxslt1-dev -y
sudo pip2 install pymavlink catkin_pkg --upgrade
sudo pip install MAVProxy==1.5.2
```

A continuación, se descarga ArUco que es una librería de realidad aumentada desarrollada por la universidad de Córdoba. Para ello hay que entrar en la página web [4] y descargar la versión 1.3.0 (apartado *OldVersions*). Se instala de la siguiente forma:

```
cd ~/Descargas // Esta dirección depende de donde se haya guardado el paquete
tar -xvzf aruco-1.3.0.tgz
cd aruco-1.3.0/
mkdir build && cd build
cmake ..
make
sudo make install
```

Posteriormente, se instala una rama específica de Ardupilot:

```
mkdir -p ~/simulation; cd ~/simulation
git clone https://github.com/erlerobot/ardupilot -b gazebo
```

Antes de instalar ROS, vendrían bien instalar los siguientes paquetes:

```
sudo apt-get install python-rosdep
sudo apt-get install autoconf
```

El siguiente paso sería instalar ROS Indigo que, como se dijo antes, es compatible con la versión de Ubuntu recomendada:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key 0xB01FA116
sudo apt-get update
sudo apt-get install ros-indigo-ros-base -y
sudo rosdep init
rosdep update
```

```
      echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
      source ~/.bashrc
      sudo apt-get install python-rosinstall ros-indigo-octomap-msgs
ros-indigo-joy ros-indigo-geodesy ros-indigo-octomap-ros ros-indigo-
mavlink   ros-indigo-control-toolbox ros-indigo-transmission-interface
ros-indigo-joint-limits-interface -y
```

Ahora, creamos el espacio de trabajo:

```
      mkdir -p ~/simulation/ros_catkin_ws/src
      cd ~/simulation/ros_catkin_ws/src
      catkin_init_workspace
      cd ~/simulation/ros_catkin_ws
      catkin_make
      source devel/setup.bash
      cd src/
```

Descargamos todos los paquetes en la capeta actual (src):

```
  git clone https://github.com/erlerobot/ardupilot_sitl_gazebo_plugin
  git clone https://github.com/tu-darmstadt-ros-pkg/hector_gazebo/
  git    clone    https://github.com/erlerobot/rotors_simulator    -b
sonar_plugin
  git clone https://github.com/PX4/mav_comm.git
  git clone https://github.com/ethz-asl/glog_catkin.git
  git clone https://github.com/catkin/catkin_simple.git
  git clone https://github.com/erlerobot/mavros.git
  git  clone  https://github.com/ros-simulation/gazebo_ros_pkgs.git  -b
indigo-devel
  #Add Python and C++ examples
  git clone https://github.com/erlerobot/gazebo_cpp_examples
  git clone https://github.com/erlerobot/gazebo_python_examples
```

En caso de que se vayan a utilizar otras librerías de ros, podrían clonarse aquí. En mi caso, se añade el paquete *robot_localization*:

```
  git  clone  https://github.com/cra-ros-pkg/robot_localization.git  -b
indigo-devel
```

Instalamos Gazebo:

```
  sudo sh -c 'echo "deb
http://packages.osrfoundation.org/gazebo/ubuntu-stable $(lsb_release -
cs) main" > /etc/apt/sources.list.d/gazebo-stable.list'
  wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key
add -
  sudo apt-get update
  sudo apt-get remove .*gazebo.* '.*sdformat.*' '.*ignition-math.*' &&
sudo apt-get update && sudo apt-get install gazebo7 libgazebo7-dev
drcsim7 -y
```

Finalmente, compilamos el espacio de trabajo. El único detalle a tener en cuenta es que si se utiliza el comando *catkin_make* puede dar problemas con el paquete *glog_catkin* por lo que, para solucionarlo, habría que realizar el siguiente cambio en las instrucciones:

```
cd ~/simulation/ros_catkin_ws
catkin_make --pkg mav_msgs mavros_msgs gazebo_msgs
source devel/setup.bash
catkin_make -j 4
```

```
cd ~/simulation/ros_catkin_ws
catkin init
catkin build
source devel/setup.bash
```

* Si diera error porque ya existe la carpeta devel o alguna otra, borrar todas las carpetas de ~/simulation/ros_catkin_ws MENOS la carpeta src y probar de nuevo.

Y, para descargar algunos modelos de Gazebo para la simulación:

```
mkdir -p ~/.gazebo/models
git clone https://github.com/erlerobot/erle_gazebo_models
mv erle_gazebo_models/* ~/.gazebo/models
```

Adicionalmente, se usará la versión 2.0.18 de APM Planner 2. Para ello, se descargará el fichero apm_planner_2.0.18_ubuntu_trusty64.deb del apartado *archive* de [5] y, en la carpeta en la que se descargue, ejecutamos la siguiente línea:

```
sudo dpkg -i apm_planner*.deb
```

En caso de que falte alguna dependencia:

```
sudo apt-get -f install
sudo dpkg -i apm_planner*.deb
```

Pudiendo comprobar su funcionamiento con:

```
apmplanner2
```

3. Puesta en marcha

Para comprobar el correcto funcionamiento de los programas anteriormente instalados, se prueba a realizar los tutoriales relativos al Erle-Copter que se encuentran en [3], teniendo en cuenta que para este trabajo no utilizaremos ni en el sonar ni en el láser.

Para empezar la simulación es necesario utilizar dos terminales:

- El primero ejecutará MAVProxy que se quedará esperando a que le conteste el segundo terminal. Se arranca así:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
cd ~/simulation/ardupilot/ArduCopter
../Tools/autotest/sim_vehicle.sh -j 4 -f Gazebo
```

- Se ejecutan las instrucciones del segundo y se espera a que por este terminal salga que está al 100% de batería. Entonces, se ejecuta la siguiente línea (solamente la primera vez):

```
Param load /[path_to_your_home_directory]/simulation/ardupilot/
Tools/Frame_params/Erle-Copter.param
```

- En el segundo terminal se ejecuta lo siguiente:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
roslaunch ardupilot_sitl_gazebo_plugin erlecopter_spawn.launch
```

- Una vez que haya cargado Gazebo, nos vamos al primer terminal (MAVProxy) y probamos a moverlo:

```
mode GUIDED
arm throttle
takeoff 5

# Se espera a que llegue y se puede probar a darle una posición
relativa a la actual

position -1 -1 -2

# El eje Z del dron apunta hacia el suelo, por lo que -2 indica que
suba 2

Mode LAND
```

- Si queremos ver lo que está grabando la cámara de la parte inferior del dron bastaría con abrir otro terminal e introducir:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
rosrun image_view image_view image:=/erlecopter/bottom/image_raw
```

- Si quisiéramos abrir la frontal sería:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
rosrun image_view image_view image:=/erlecopter/front/image_front_raw
```

- Si se quisiera cambiar de estación de tierra de MAVProxy a APM Planner 2, habría que abrir un terminal más:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
roslaunch mavlink_bridge.launch
```

Hay que tener en cuenta que, aunque en los vídeos tutoriales [6] el programa APMPlanner2 se conecta automáticamente con el dron, la primera vez que lo hagas puede ser que no esté conectado. Para solucionarlo, hay que darse cuenta de que, al estar haciendo un SITL, APMPlanner2 no se va a conectar por un puerto serial físico sino por uno UDP tal y como indica el documento mavlink_bidge.launch. Dicho puerto UDP puede variar desde el 14550 al 14559. Para saber cuál de esos es, bastaría con usar el comando output una vez por el terminal que está corriendo MAVProxy. Ahora, habría que irse a APMPlanner 2, activar la configuración avanzada (menú superior) y, una vez hecho esto, se pulsa en comunicación (de nuevo en el menú superior) en el apartado de puerto UDP. En la ventana que aparezca hay que poner el valor de antes y darle a conectar. En mi caso particular esto pasó para 14551.

El resto de pruebas que se refieren a la creación de archivos con órdenes para el dron y la obtención de información de este, pueden verse en [3].

4. Cambiar entorno de simulación

Si quisiéramos cambiar el lugar en el que se simula el erlecopter, tendremos que modificar los archivos *.world* y *.launch* de la simulación ubicados en *src/ardupilot_sitl_gazebo_plugin/ ardupilot_sitl_gazebo_plugin* dentros en las carpetas *worlds* y *launch* respectivamente.

En la carpeta *launch*, podemos modificar el archivo erlecopter_spawn.launch y así modificar la posición en la que aparece al inicio. En mi caso, se ha cambiado el valor de *yaw* a 0 radianes para que, al inicio, los ejes del UAV sean coincidentes con los ejes mundo y no tener que sumar un offset a la hora de realizar gráficas.

Si se quisiera cambiar la apariencia del lugar donde se realiza la simulación, habrá que retocar el fichero *empty.world* (dentro de *empty_world*) ya que este es el mundo al que llama por defecto. Para aumentar el número de elementos que podemos incluir en la simulación, podemos descargar todos los modelos de objetos disponibles de Gazebo de la siguiente forma:

```
git clone https://github.com/osrf/gazebo_models.git
mv gazebo_models/* ~/.gazebo/models
```

Personalmente, he optado por copiar todos los elementos del mundo *outdoor-village* y pegarlos en nuestro fichero *empty.world*. De esta forma estaremos simulando el dron en un mundo cuya apariencia es más realista (y visualmente atractiva). Para añadir un nuevo elemento, sólo hay que incluir la siguiente estructura por cada elemento junto al resto de modelos:

```
<include>
  <name>nombrequedamosalobjeto</name>
  <uri>model://nombredelmodelo</uri>
  <pose>x y z   roll pitch yaw</pose>
</include>
```

Si quisiéramos añadir un modelo house_2 en la posición [-12, 8, 0] con un giro en Z de 90º y fuera la segunda casa que añadimos, podríamos incluir algo así:

```
<include>
  <name>house_2_2</name>
  <uri>model://house_2</uri>
  <pose>-12 8 0   0 0 1.5707</pose>
</include>
```

Ojo, a la hora de hacer esto al igual que al añadir un nuevo modelo al mundo, tenemos que tener cuidado de no borrar líneas referidas a las condiciones atmosféricas (a no ser que queramos cambiarlas). Igualmente, como se acaba de indicar, podrían modificarse los parámetros físicos para ver cómo se comportaría frente a diferentes condiciones ambiente.

El fichero del mundo utilizado en este trabajo se incluye en el repositorio creado a cuenta de este proyecto.

5. Referencias

[1]: Guijarro Martínez, Miguel Ángel & Espinosa Zapata, Felipe & Martínez Rey, Miguel. (2018) *Modelado y simulación de dron Erle-Copter con ROS/Gazebo* (trabajo fin de grado). Universidad de Alcalá de Henares, Madrid, España.

[2]: Buyval, Alexandr & et al. (11 junio 2015) *RotorS_simulator.* Recuperado de: https://github.com/erlerobot/rotors_simulator

[3]: Erlerobotics. (26 abril 2018) *Erlerobotics documents.* Recuperado de: https://web.archive.org/web/20180426132658/http://docs.erlerobotics.com/

[4]: Muños-Salinas, Rafael. (25 marzo 2017) *Descargar versiones ArUco.* Recuperado de: https://sourceforge.net/projects/aruco/files/

[5]: ArduPilot. (25 julio 2015) *Descargar versiones ArduPilot.* Recuperado de: https://firmware.ardupilot.org/Tools/APMPlanner/

[6]: Acutronic Robotics. (18 enero 2016) *Erle-Copter.* Recuperado de: https://www.youtube.com/watch?v=2HN-W8qLb6g&list=PL39WpgKDjDfVh5_3aVwd2RYH6f_t-ByKv

[7]: Fang, Troye (4 noviembre 2018) *Connecting to gazebo model database (answer).* Recuperado de: https://stackoverflow.com/questions/37105913/connecting-to-gazebo-model-database

B) Instrucciones de instalación de OpenCV 3.1.0 y cv_bridge para conexión con ROS

1. Instalación OpenCV 3.1.0

Se procede a explicar los pasos a seguir para tener OpenCV 3.1.0 operativo. Esto se realiza porque, aunque al instalarse ROS Indigo se instala de manera paralela la versión 2.4 de OpenCV, no se incluyen unos módulos que incorporan algunas funciones que usaremos posteriormente como, por ejemplo, el creador de descriptores SURF.

- Se instalan unos paquetes para el correcto funcionamiento del sistema (a lo mejor ya se tenían de antes):

```
  sudo apt-get install build-essential
  sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev
libavformat-dev libswscale-dev
  sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev
libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev
// Este ultimo paquete es opcional
```

- A continuación, descargamos los archivos correspondientes a la versión/rama 3.1.0 en una carpeta que, en mi caso, ya he creado y se llama OCV:

```
  cd ~/OCV
  git clone -b 3.1.0 --single-branch
https://github.com/opencv/opencv/tree/3.1.0
  git clone -b 3.1.0 --single-branch
https://github.com/opencv/opencv_contrib/tree/3.1.0
```

* Si no funcionase, probar con https://github.com/opencv/opencv y con https://github.com/opencv/opencv_contrib.

- Creamos un directorio para hacer *build* y, a continuación, hacemos *cmake*:

```
  cd opencv
  mkdir build
  cd build

  cmake -D CMAKE_BUILD_TYPE=RELEASE \ -D CMAKE_INSTALL_PREFIX=/usr/local
\ -D INSTALL_C_EXAMPLES=OFF \ -D INSTALL_PYTHON_EXAMPLES=ON \ -D
OPENCV_EXTRA_MODULES_PATH=~/OCV/opencv_contrib/modules \ -D
BUILD_EXAMPLES=ON ..
  # Todo esto es una única línea. Puede dar error debido a las -D. Si
esto ocurriera, mantén las -D delante de CMAKE… y elimina las demás.
  make -j4
  sudo make install
  sudo ldconfig
```

¡Ya tenemos instalado OpenCV 3.1.0!

2. Compilación de archivos que recurren a OpenCV con cmake/make

A la hora de compilar, el archivo CMakeLists.txt tendrá que tener una apariencia como la siguiente:

```
# cmake needs this line
cmake_minimum_required(VERSION 2.8)

# Define project name
project(Nombre_proyecto)

# Find OpenCV, you may need to set OpenCV_DIR variable
# to the absolute path to the directory containing OpenCVConfig.cmake
file
# via the command line or GUI
find_package(OpenCV REQUIRED)

# If the package has been found, several variables will
# be set, you can find the full list with descriptions
# in the OpenCVConfig.cmake file.
# Print some message showing some of them
message(STATUS "OpenCV library status:")
message(STATUS "    config: ${OpenCV_DIR}")
message(STATUS "    version: ${OpenCV_VERSION}")
message(STATUS "    libraries: ${OpenCV_LIBS}")
message(STATUS "    include path: ${OpenCV_INCLUDE_DIRS}")

include_directories(${OpenCV_INCLUDE_DIRS})

# Declare the executable target built from your sources
add_executable(Nombreejecutable1 Nombrecpp1.cpp)
add_executable(Nombreejecutabl21 Nombrecpp2.cpp)

# Link your application with OpenCV libraries
target_link_libraries(Nombreejecutable1 PRIVATE ${OpenCV_LIBS})
target_link_libraries(Nombreejecutable2 PRIVATE ${OpenCV_LIBS})
```

Aquí se utilizan exclusivamente las librerías adicionales de OpenCV, pudiendo añadirse otras dependencias necesarias según se requiera. Dicho archivo CMakeLists.txt se encontrará en la misma carpeta que el cpp a compilar (aunque también pueden meterse los archivos en una carpeta si se cambia alguna parte de CMakeLists.txt). Para compilar bastaría con ejecutar por terminal:

```
cd direccion_carpeta_con_archivos
cmake ..
make
```

Y, para ejecutarlo (suponiendo que tiene dos argumentos de entrada):

```
./Nombreejecutable1 arg1 arg2
```

3. Conectar ROS y OpenCV

Como estamos usando OpenCV 3.1, no podremos usar el bridge que viene instalado con ROS Indigo por defecto. Por ello, hay que instalar y compilar el nuevo archivo *bridge_cv*:

```
  cd ~/Descargas
  git clone -b indigo https://github.com/ros-perception/
vision_opencv.git
  cd vision_opencv
  mkdir -p ~/catkin_cv/src
  cd ~/catkin_cv/src
  catkin_init_workspace
  cd ~/Descargas/vision_opencv
  cp -rf cv_bridge ~/catkin_cv/src
  cd ~/catkin_cv/src

  mv cv_bridge cv_bridge_new
  cd cv_bridge_new
  gedit CmakeLists.txt #Sustituimos en el nombre del proyecto cv_bridge
por cv_bridge_new
  gedit package.xml #Sustituimos cv_bridge por cv_bridge_new igualmente
  cd ..
  cd ..
  catkin_make
```

- Añadimos este cv_bridge a nuestra fuente para que el sistema busque esa configuración:

```
  source /home/nombreusuario/catkin_cv/devel/setup.bash
```

- Si queremos que esta fuente esté por defecto, habrá que cambiar el ~/.bashrc:

```
  cd ~
  gedit .bashrc
```

Al final del archivo que se abra, deberemos añadir la línea de código anterior source/home/nombreusuario/catkin_cv/devel/setup.bash .Esto podría hacerse igualmente para el archivo setup.bash de las simulaciones en Gazebo, evitando así tener que escribir la fuente siempre. Además, como pasaremos a usar el nuevo cv_bridge_new, habrá que modificar los archivos CMakeFile.txt y package.xml sustituyendo cv_bridge por cv_bridge_new y tendremos que borrar el archivo cv_bridge:

```
  rm     -rf     /home/nombreusuario/catkin_cv/devel/lib/python2.7/dist-
packages/cv_bridge
```

- Ahora ya podría funcionar... pero podría dar algún fallo al no detectar algunas librerías como libopencv_highgui.so.3.1 en mi caso. Para resolverlo:

```
  sudo find / -name "librería_que_no_encuentra*" #copiamos la dirección
que salga pero sin incluir el archivo. Ejemplo:

Salida: /usr/local/lib/libopencv_core.so.3.2.

Copiamos: /usr/local/lib/

  su    #Para acceder al modo administrador (a lo mejor te pide tu
contraseña)
  cd /etc/ld.so.conf.d
  gedit opencv.conf #pegamos en el documento únicamente la dirección que
obtuvimos como resultado anterior
  sudo ldconfig -v
```

¡ Ya no debería dar problemas con las bibliotecas y podremos enlazar ROS y OpenCV !

4. Referencias

[1]: Huamán, Ana. (18 diciembre 2015) *Installation in Linux*. Recuperado de : https://docs.opencv.org/3.1.0/d7/d9f/tutorial_linux_install.html

[2]: Wang, Marc. *How to install OpenCV 3.1 on Ubuntu 14.04 64bits.* Recuperado de: https://gist.github.com/MarcWang/0547f87cf777b6576275

[3]: Chen, Tao. (22 septiembre 2016) *OpenCV-3.1-with-Python-2.7-and-ROS-Indigo.* Recuperado de: https://github.com/taochenshh/OpenCV-3.1-with-Python-2.7-and-ROS-Indigo

[4]: Whitaker, Kirstie. (20 enero 2016) *Setting up your .bashrc file.* Recuperado de: https://github.com/KirstieJane/NSPN_MRIProcessing/wiki/Setting-up-your-.bashrc-file

[5]: Hongchen, Gao. (4 marzo 2017) *OpenCV runtime error: "libopencv_core.so.3.2: cannot open shared object file: No such file or directory" in Fedora 24.* Recuperado de: https://github.com/cggos/dip_cvqt/issues/1

C) Índice de programas y archivos ROS del repositorio

A continuación se muestra una lista con los programas subidos al repositorio y una corta descripción.

Atención 1: La mayoría de programas guardan datos en un fichero con una dirección determinada, por lo que si se van a usar se recomienda cambiar dicha dirección o borrarla directamente.

Atención 2: Si no se comenta nada se utiliza SURF por defecto y no ORB.

Atención 3: Aunque para el despegue se haya tomado la medida de la orientación del magnetómetro para tener en la simulación una condición inicial lo mejor posible, también podría usarse el resultado de la integración del giroscopio. Para ello habría que ejecutar IMU_publisher, Imu_integration_estimation_publisher y suscribirse al *topic* /imu_estimation.

- camera_abs_hom.cpp: Usa inicialmente el marcador ArUCo con ID 23 para estimar la posición inicial y, cuando supera los 7 metros, cambia al algoritmo de comparación de fotrogramas con respecto a fotograma de referencia. Apartado 7.7.
- camera_handmade_marker.cpp: Realiza la estimación de posición con un marcador ArUCo de ID 23, tomando como matriz de cambio de base la calculada a mano. Apartado 6.3.
- camera_IMUinv_magnetometer_marker.cpp: Realiza la estimación de posición con un marcador ArUCo de ID 23, tomando como matriz de cambio de base la proporcionada por el magnetómetro. Apartado 6.5.
- camera_IMUinv _marker.cpp: Realiza la estimación de posición con un marcador ArUCo de ID 23, tomando como matriz de cambio de base la proporcionada por la integración del giroscopio. Apartado 6.4
- camera_mixed.cpp: Inicialmente se basa en el marcador ArUCo de ID 23 para el despegue hasta llegar a 7m. Entonces, realiza la estimación de posición a través de la homografía entre fotogramas continuos usando como matriz de giro y factor de escala los valores estimados.
- camera_mixed_with_ORB.cpp: Igual que el anterior pero utilizando el algoritmo ORB con los parámetros por defecto en vez de SURF con un umbral de 1000 como se usaba en camera_mixed.cpp. Apartado 7.5.
- camera_mixed_reinit.cpp: Igual que camera_mixed.cpp, pero con la condición de reinicio ya comentada en el apartado 7.6.
- camera_mixed_vars.cpp: Igual que camera_mixed.cpp, pero te guarda los desplazamientos y variaciones de ángulos en el archivo en vez de la posición y ángulos totales.
- camera_mixed_with_EKF6: camera_mixed.cpp pero publicando en un *topic* las estimaciones de la homografía como velocidades para usarlas en el filtro de Kalman. Apartado 8.3.
- camera_mixed_with_EKF6b: Igual que el anterior, pero fusiona también las estimaciones del ArUCo. Esto es útil para que, si se usa el acelerómetro, no acumule mucho error al principio, aunque perdemos precisión durante el despegue.
- camera_perfect.cpp: Realiza la estimación de posición a través de la homografía entre fotogramas continuos usando como matriz de giro y factor de escala el valor real (*ground_truth*).
- camera_perfect_vars.cpp: Igual que el anterior, pero guarda solo las variaciones en el archivo de texto.
- camera_pure_marker.cpp: Realiza la estimación de posición a partir de un marcador único cualquiera del diccionario utilizado que debe estar en la base despegue.

- camera_pure_several_markers.cpp: Realiza la estimación de posición utilizada a lo largo de 6.2. Deben comentarse partes del código según el número de marcadores a utilizar. Si se usan los 9 marcadores en las posiciones indicadas no se comentaría nada.
- CMakeLists.txt: Archivo que almacena los nombres de los ficheros a compilar y las dependencias del paquete que estamos haciendo.
- EKF_receiver.cpp: Se subscribe al *topic* de salida del EKF y guarda en un fichero los mensajes.
- Euler.hpp: Fichero de cabecera que incluye funciones relacionadas con la transformación de cuaterniones, ángulos de Euler y matrices de rotación. Apartado
- get_images.cpp: Guarda las imágenes tomadas por la cámara del dron.
- ground_truth.cpp: Se subscribe al *topic* del valor real de posición y ángulos del dron y guarda la información en un archivo de salida.
- handmade_marker.launch: Ejecuta camera_handmade y ground_truth.
- Image_viewer.cpp: Muestra por pantalla en directo las imágenes que está grabando el dron.
- IMU_EKF.launch: Ejecuta un EKF con los valores del acelerómetro y giroscopio calibrados según params_IMU.yaml.
- imu_integration_estimation_publisher:.cpp Integra los resultados del giroscopio y el acelerómetro ya calibrados y los publica.
- imu_listener.cpp: Se subscribe al *topic* de la IMU y estima posición y ángulos sin calibrar.
- IMU_publisher.cpp: Se subscribe al *topic* de la IMU y aplica por un lado la calibración y, por otro, la eliminación de la gravedad en función de la estimación anterior de la orientación ofrecida por el EKF.
- IMUinv_integration_marker.launch: Ejecuta camera_IMUinv_marker, ground_truth y IMU_publisher y Imu_integration_estimation_publisher para el apartado 6.4.
- IMUinv_magnetometer_marker.launch: Ejecuta camera_IMUinv_magnetometer_marker y ground_truth. Apartado 6.5.
- mixed.launch: Ejecuta camera_mixed y ground_truth.
- mixed_ORB: Ejecuta camera_mixed_ORB y ground_truth.
- mixed_with_EKF6.launch: Ejecuta camera_mixed_with_EKF6, IMU_publisher, el nodo EKF con los parámetros de params_mixed_withEKF6.yaml y EKF_receiver.
- mixed_with_EKF6b.launch: Ejecuta camera_mixed_with_EKF6b, IMU_publisher, el nodo EKF con los parámetros de params_mixed_withEKF6b.yaml y EKF_receiver.
- mixed_with_reinit.launch: Ejecuta camera_mixed_with_reinit y ground_truth.
- params_IMU.yaml: Contiene los parámetros del EKF de IMU_EKF.launch.
- params_mixed_with_EKF6.yaml: Contiene los parámetros del EKF de mixed_with_EKF6.launch.
- params_mixed_with_EKF6b.yaml: Contiene los parámetros del EKF de mixed_with_EKF6b.launch.
- perfect.launch: Ejecuta camera_perfect y ground_truth.
- pure_marker: Ejecuta camera_pure_marker y ground_truth.
- pure_several_markers.launch: Ejecuta camera_pure_several_markers, Image_viewer y ground_truth.
- Vars.launch: Ejecuta camera_perfect_vars y ground_truth.

D) Ejemplo de programas realizados con comentarios

En este apartado se va a exponer a modo de ejemplo algunos de los tipos de programas que se han realizado y que están disponibles en el repositorio indicado en el capítulo 2 (https://github.com/JJavierga/TFG_posicionamiento_por_vision.git). En concreto, se va a elegir el archivo .launch más completo de todos, aquel que usa el EKF para fusionar con el giroscopio.

1. Archivo mixed_with_EKF6.launch

Este archivo es el que se encarga de indicar a ROS los nodos que tiene que crear y los ejecutables que contiene cada uno de ellos. En este caso arranca un nodo de estimación de posición basado en visión, uno que se encarga de modificar los valores de la IMU de acuerdo con la calibración obtenida, otro nodo que realiza el EKF según los parámetros especificados en *params_mixed_with_EKF6.yaml* y un último que escribe en un fichero los datos de salida del nodo EKF.

```
  <launch>
    <node pkg="useful_info" type="camera_mixed_with_EKF6" name="pub_camera_estimation"/>
    <node pkg="useful_info" type="IMU_publisher" name="imu_adaptor"/>
    <node pkg="robot_localization" type="ekf_localization_node" name="ekf_se"
clear_params="true">
      <rosparam command="load" file="$(find useful_info)/params_mixed_with_EKF6.yaml" />
    </node>
    <node pkg="useful_info" type="EKF_receiver" name="EKF_adaptor"/>
  </launch>
```

2. Archivo mixed_with_EKF6.yaml

Es el que guarda los parámetros que se van a utilizar a la hora de elegir los parámetros que definirán el comportamiento de nuestro nodo de EKF.

```
  frequency: 7 # Frecuencia de fusión de las medidas
  sensor_timeout: 30 # Segundos que debe pasar un sensor sin mandar información antes de
que se considere que está roto

  two_d_mode: false # No es un problema de movimiento plano


  transform_time_offset: 0.0
  transform_timeout: 0.0

  print_diagnostics: false
  debug: false # Si quieres depurar, poner a true (aunque hace el código más lento)
  debug_out_file: /home/javier/pruebas/MasPruebas/Data/file.txt # Salida de depuración
  publish_tf: true
  publish_acceleration: false

  # Las 4 líneas siguientes sirven para renombrar los sistemas definidos por REP-105

  map_frame: map              # Por defecto a "map"
  odom_frame: odom            # Por defecto a "odom"
  base_link_frame: erlecopter/imu_link  # Por defecto a "base_link"
  world_frame: odom           # Por defecto a odom_frame
```

```
  # Indicamos primera fuente de información, que nos proporciona velocidades angular y
lineales a través del topic Camera_estimation y puede fusionar los últimos 8 mensajes
recibidos.

  twist0: /Camera_estimation
  twist0_config:  [false,  false,  false, # x, y, z
                   false,  false,  false, # roll, pitch, yaw
                   true,   true,   true,  # vx, vy, vz
                   true,   true,   true,  # vroll, vpitch, vyaw
                   false,  false,  false] # ax, ay, az
  twist0_differential: false
  twist0_relative: false

  twist0_queue_size: 8


  # Indicamos segunda fuente de información, que nos proporciona velocidad angular a
través del topic imu_modified (valores de la IMU calibrados) y puede fusionar los últimos
40 mensajes recibidos. La IMU es recomendable usarla en modo diferencial. Si se quisiera
incluir la aceleración, solamente haría falta poner la última fila a true ya que al haberse
calibrado también se ha eliminado la gravedad.


  imu0: /imu_modified
  imu0_config:  [false, false, false,
                 false, false, false,
                 false, false, false,
                 true,  true,  true,
                 false, false, false]
  imu0_differential: true
  imu0_relative: false
  imu0_queue_size: 40
  imu0_remove_gravitational_acceleration: false
  # La eliminación de gravedad automática solamente funciona si se proporciona en el
mensaje unos valores de los ángulos que permitan descomponerla, en nuestro caso estos
valores procedentes del magnetómetro no se usan.

  # Los valores ordenados son: x,y,z,roll,pitch,yaw,vx,vy,vz,vroll,vpitch,vyaw,ax,ay,az
  # Matriz de covarianza del ruido añadido al error total tras cada paso de predicción:

  process_noise_covariance:  [0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                              0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0.6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0.3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0.3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0.6, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0, 0.25, 0, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0, 0, 0.25, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0, 0, 0, 0.4, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0, 0, 0, 0,
                              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.2, 0, 0, 0,
                              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0, 0,
                              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0,
                              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1]


  # Matriz de covarianza de la estimación del error del estado inicial:


  initial_estimate_covariance: [1e-6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 1e-6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 1e-6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 1e-6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 1e-6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 1e-6, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 1e-6, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 1e-6, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 1e-6, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 1e-6, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1e-6, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1e-6, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1e-6, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1e-6, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1e-6]
```

3. Archivo camera_mixed_with_EKF6.cpp

Este archivo se encarga de realizar las estimaciones de visión. Para ello, tiene un periodo inicial de despegue durante el que se basa en un marcador ArUCo y, al llegar a los 7m, empieza el uso de marcadores naturales.

Durante el despegue, se usa *Aruco_call*, la cual tiene como objetivo buscar en las imágenes que va recibiendo de la cámara un marcador ArUCo con ID 23 que proporcionará información de la posición, mientras que la orientación se toma directamente del magnetómetro de la IMU (*topic* de la IMU original /erlecopter/imu). Esto último se hace para asegurar un buen resultado de la estimación inicial para cuando pase a estimar la posición utilizando marcadores naturales.

Tras el despegue, se usa *Homography_call*, que realiza los pasos ya comentados en el capítulo 3 para la estimación de posición a partir de marcadores naturales. Los valores de los ángulos y de la posición anteriores del UAV los obtiene de la salida del filtro de Kalman (*topic* /odometry/filtered) ya que estos son los valores estimados en total uniendo visión y giroscopio. Una vez calculadas las variaciones estimadas por visión, manda al nodo EKF dichos valores por el topic /Camera_estimation.

```
#include <ros/ros.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>

#include <image_transport/image_transport.h>
#include "geometry_msgs/PoseStamped.h"
#include <geometry_msgs/TwistWithCovarianceStamped.h>
#include "nav_msgs/Odometry.h"
#include "geometry_msgs/PoseStamped.h"
#include "sensor_msgs/Imu.h"
#include <cv_bridge/cv_bridge.h>

#include <opencv2/highgui.hpp>
#include <opencv2/imgcodecs.hpp>
#include "opencv2/core.hpp"
#include "opencv2/imgproc.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/calib3d.hpp"
#include "opencv2/xfeatures2d.hpp"
#include <opencv2/aruco.hpp>

#include "Euler.hpp"

Mat Prev_rot_matrix=Mat::eye(3,3,CV_64FC1);//Para calculo rotación más parecida
Mat R_axis_w2uav=Mat::eye(3,3,CV_64FC1); // Rotacion de los ejes desde el mundo al UAV
EulerAngles Euler_curr;
Mat Position=Mat::zeros(3,1,CV_64FC1);
Mat Disp=Mat::zeros(3,1,CV_64FC1);


EulerAngles angles={0,0,0};
double Initial_height=0;

int seq=0;
Mat img_object;
int flag_h7=0;//Bandera de fin de despegue

ros::Time Prev_time;

std::ofstream file;

EulerAngles transf={0,CV_PI,CV_PI/2};
Mat Rot=ComposeRotation(transf);//Matriz de cambio de base UAV-camara
```

```
  ros::Publisher Pub_estimation;
  // Publicador global ya que no se publicara desde el main sino desde una callback

  boost::array<double, 36ul> Covariance_default=
  {/*x*/5e-5, 0, 0, 0, 0, 0,
   0, /*y*/1e-4, 0, 0, 0, 0,
   0, 0, /*z*/2e-6, 0, 0, 0,
   0, 0, 0, /*roll*/4e-3, 0, 0,
   0, 0, 0, 0, /*pitch*/1e-3, 0,
   0, 0, 0, 0, 0, /*yaw*/4e-3}; // Matriz de covarianza de vision

  Mat CameraMatrix = (Mat_<double>(3,3) << 374.67,0,320.5,0,374.67,180.5,0,0,1);
  //Matriz intrínseca de la cámara

  void Homography_call(Mat img_scene,ros::Time stamp);// Función de homografía
  void Aruco_call(Mat img_scene,ros::Time stamp);// Función ArUCo


  void imuCallback(const sensor_msgs::Imu::ConstPtr& msg)
  {

       if(flag_h7==0)

              Euler_curr=Quat2Euler(msg->orientation); // Pasa cuaternion a angulos
              // Mientras que no llegas a 7m, lees la orientación del magnetómetro (para
              así tomar como valor inicial el más preciso posible.

  }


  void ScaleCallback(const nav_msgs::Odometry::ConstPtr& msg) // Toma estimación anterior
saliente del EKF para estimar el siguiente instante tras el despegue (7m)
  {

       if(flag_h7==1)
       {

              //Desplazamientos relativo al final del despegue
              Disp.at<double>(0)=msg->pose.pose.position.x;
              Disp.at<double>(1)=msg->pose.pose.position.y;
              Disp.at<double>(2)=msg->pose.pose.position.z;


              // Rotacion que convierte los ejes mundo a UAV
              R_axis_w2uav=ComposeRotation(Quat2Euler(msg->pose.pose.orientation));


              //Posicion global con respecto al punto inicial (ejes mundo)
              Position.at<double>(0)=Disp.at<double>(0);
              Position.at<double>(1)=Disp.at<double>(1);
              Position.at<double>(2)=Initial_height+Disp.at<double>(2);

       }
  }


  void imageCallback(const sensor_msgs::ImageConstPtr& msg) //Cuando llega una imagen
  {
    if(seq==0)
    {

       seq=1;
       img_object = cv_bridge::toCvShare(msg, "mono8")->image;
       Prev_time=msg->header.stamp;
    }
    else
    {

       try
       {

       // Lectura de imágenes
       Mat img_scene = cv_bridge::toCvShare(msg, "mono8")->image;

       if( !img_object.data || !img_scene.data )
       { std::cout<< " --(!) Error reading images " << std::endl;}
```

```
        // Distinción entre despegue y funcionamiento normal.
        if(flag_h7==1)
        {Homography_call(img_scene,msg->header.stamp);}
        else
        {Aruco_call(img_scene,msg->header.stamp); }


        img_scene.copyTo(img_object);
        cv::waitKey(30);
        }

        catch (cv_bridge::Exception& e)
        {ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str());}
    }
  }


  int main(int argc, char **argv)
  {
    // Comandos de inicio y configuración del nodo de ROS
    ros::init(argc, argv, "image2angles");
    ros::NodeHandle nh;
    image_transport::ImageTransport it(nh); // Comando para convertir imágenes a Mat
    image_transport::Subscriber sub_img = it.subscribe("/erlecopter/bottom/image_raw", 1,
imageCallback); // Subscripcin a imagenes de la camara inferior
    ros::Subscriber    sub_h    =    nh.subscribe("/erlecopter/imu",    1,    imuCallback);
//Subscripcion a valores de la IMU (sin calibrar y que usaremos el magnetómetro como base)
    ros::Subscriber  sub_h2  =  nh.subscribe("/odometry/filtered",  1,  ScaleCallback);
//Subscripcion a salida del nodo que realiza el EKF

Pub_estimation=nh.advertise<geometry_msgs::TwistWithCovarianceStamped>("Camera_estimatio
n",1); // Especificacion del topic de publicacion
    ros::spin(); // Se repite de manera indefinida
  }


  void Homography_call(Mat img_scene, ros::Time stamp)
  {

// Selección del algoritmo SURF con un umbral de 1000

Ptr<Feature2D> fd=cv::xfeatures2d::SURF::create(1000); // Threshold=1000


//Detección de puntos y creación de descriptores

Mat descriptors_object, descriptors_scene;
std::vector<KeyPoint> keypoints_object, keypoints_scene;
fd->detectAndCompute( img_object, Mat(), keypoints_object, descriptors_object );
fd->detectAndCompute( img_scene, Mat(), keypoints_scene, descriptors_scene );


// Emparejamiento de descriptores con algoritmo FLANN

std::vector< DMatch > matches;
FlannBasedMatcher matcher;
if(keypoints_object.size()>0 && keypoints_scene.size()>0)
{ //Inicio if al detectarse algún punto

matcher.match( descriptors_object, descriptors_scene, matches );


// Nos quedamos con los mejores emparejamientos

double max_dist = 0; double min_dist = 100;

// Calculamos distancias
for( int i = 0; i < descriptors_object.rows; i++ )
{

  double dist = matches[i].distance;
  if( dist < min_dist ) min_dist = dist;
  if( dist > max_dist ) max_dist = dist;

}
```

```
//Cogemos solo aquellos con una distancia menos de un tercio de la amplitud
double threshold=(max_dist-min_dist)/3+min_dist;
std::vector< DMatch > good_matches;
for( int i = 0; i < descriptors_object.rows; i++ )
{

  if( matches[i].distance <= threshold )
  { good_matches.push_back( matches[i]); }

}



// Reordena los puntos en vectores para la img anterior y la actual
std::vector<Point2f> obj;
std::vector<Point2f> scene;


for( size_t i = 0; i < good_matches.size(); i++ )
{
    obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
    scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
}


Mat H;


// Calcula homografía por RANSAC

if(!good_matches.empty())
   H = findHomography( obj, scene, RANSAC );


EulerAngles Prev_angles=angles;
Mat tras=Mat::zeros(3,1,CV_64FC1);


// Extracción de desplazamientos y rotaciones

if (! H.empty())
{

  std::vector<Mat> Rs_decomp, ts_decomp, normals_decomp;
  double Min_var=10000,Var_cost=0;
  int SOL=0;


  // Descomposición de la homografía
  int solutions=decomposeHomographyMat(H,CameraMatrix,Rs_decomp,ts_decomp,
  normals_decomp);



  // Quédate con la solución que aplique
  for (int i = 0; i < solutions; i++)
  {

        if(normals_decomp[i].at<double>(2)>0)
        {

                Var_cost=FrameVar(Rs_decomp[i], Prev_rot_matrix);
                // Calcula la variación en el giro entre el giro anterior y el actual
                //(definida en Euler.hpp)
                if(Var_cost<Min_var)
                {

                        Min_var=Var_cost;
                        SOL=i;

                }

        }
  }
```

```
  // Calcula el desplazamiento y los ángulos girados en la base mundo.
  tras=-R_axis_w2uav*Rot*ts_decomp[SOL]*(Position.at<double>(2));
  angles=fast_Euler_angles(R_axis_w2uav*Rot*Rs_decomp[SOL].t()*Rot);

  Prev_rot_matrix=Rs_decomp[SOL];

}
else //Si no detectas suficientes puntos, supon que no gira ni se mueve
{  Prev_rot_matrix=Mat::eye(3,3,CV_64FC1);  }


// Calcula tiempo pasado entre imágenes

double interval=(stamp-Prev_time).toSec();


// Crea y publica el mensaje con las variaciones

geometry_msgs::TwistWithCovarianceStamped pub_msg;


pub_msg.header.seq=seq;
pub_msg.header.stamp=stamp;
pub_msg.header.frame_id="odom";

pub_msg.twist.twist.linear.x=tras.at<double>(0)/interval;
pub_msg.twist.twist.linear.y=tras.at<double>(1)/interval;
pub_msg.twist.twist.linear.z=tras.at<double>(2)/interval;

pub_msg.twist.twist.angular.x=(angles.roll-Prev_angles.roll)/interval;
pub_msg.twist.twist.angular.y=(angles.pitch-Prev_angles.pitch)/interval;
pub_msg.twist.twist.angular.z=(angles.yaw-Prev_angles.yaw)/interval;

pub_msg.twist.covariance=Covariance_default;


Pub_estimation.publish(pub_msg);


Prev_time=stamp;


// Guarda en archivo

file.open("/home/javier/pruebas/MasPruebas/Data/Mixed_EKF_hom.txt",  std::ios_base::app);
file << stamp << ", " <<  angles.roll << "," << angles.pitch << "," << angles.yaw << ","
<<    Position.at<double>(0)    <<    ","    <<    Position.at<double>(1)    <<    ","    <<
Position.at<double>(2) << std::endl;

file.close();


}//Final del if de detectar algún punto

}//Final de la funcion


void Aruco_call(Mat img_scene, ros::Time stamp)
{

// Toma rotación de la IMU

EulerAngles Angles=Euler_curr;
Mat rot_mat_imu=ComposeRotation(Angles);


// Toma como diccionario el de 6x6

cv::aruco::Dictionary dictionary =
cv::aruco::getPredefinedDictionary(cv::aruco::DICT_6X6_250);

std::vector<int> ids;
std::vector<std::vector<cv::Point2f> > corners;
```

```
// Busca marcadores

cv::aruco::detectMarkers(img_scene, dictionary, corners, ids);


int i, flag23=0;
for(i=0;i<ids.size();i++)
{

  if(ids[i]==23) // Si detecta el de ID 23...
  {

        flag23=1;
        break;

  }

}



if (flag23==1) // ...Entonces...
{

  // Estima posición del marcador con respecto a la cámara
  std::vector<cv::Vec3d> rvecs, tvecs;
  cv::aruco::estimatePoseSingleMarkers(corners, 0.45215, CameraMatrix, cv::Mat(), rvecs,
tvecs);


  // Cambia el resultado a cámara con respecto al marcador
  Mat t_aux=(Mat_<double>(3,1) << tvecs[i][0], tvecs[i][1], tvecs[i][2]);
  Position= -1*rot_mat_imu*Rot.t()*t_aux;
  Position.at<double>(0)+=-0.15; // Offset posicion en X

  // Almacena el resultado en el archivo
  file.open("/home/javier/pruebas/MasPruebas/Data/Mixed_EKF_aruco.txt",
std::ios_base::app);
  file << stamp << ", " <<  Angles.roll << "," << Angles.pitch << "," << Angles.yaw <<
","   <<   Position.at<double>(0)   <<   ","   <<     Position.at<double>(1)   <<   ","   <<
Position.at<double>(2) << std::endl;
  file.close();


  if(Position.at<double>(2)>7) // Si supera los 7m de altura
  {

        flag_h7=1; //Termina el despegue
        Initial_height=Position.at<double>(2);


  }


}

}
```

4. Archivo IMU_publisher.cpp

Este archivo se encarga de realizar la calibración de los valores aportados por la IMU, es decir, se subscribe a los valores de la IMU, los corrige, y devuelve el resultado por el *topic* /imu_modified. Puesto que elimina también la gravedad, necesita subscribirse a la salida del EKF (que es el valor estimado de posición y orientación) para saber los ángulos del UAV y así cambiar de base.

```
#include "sensor_msgs/Imu.h"
#include "nav_msgs/Odometry.h"

#include <fstream>

#include <opencv2/highgui.hpp>
#include <opencv2/imgcodecs.hpp>
#include "opencv2/core.hpp"
#include "opencv2/imgproc.hpp"

#include "Euler.hpp"


EulerAngles Ang_prev={0, 0, 0};
ros::Publisher IMU_mod_pub;

using namespace cv;

void OrientationCallback(const nav_msgs::Odometry::ConstPtr& msg)
//Toma ángulos de salida del EKF
{
     Ang_prev=Quat2Euler(msg->pose.pose.orientation); // Toma orientacion estimada como
     cuaternion y la pasa a angulos de Euler
}

void IMUCallback(const sensor_msgs::Imu::ConstPtr& msg)
{

sensor_msgs::Imu pub_msg;

//Crea vector de aceleraciones
Mat Acc_wrt_prev = (Mat_<double>(3,1) << msg->linear_acceleration.x,
msg->linear_acceleration.y, msg->linear_acceleration.z);

Mat Rot = ComposeRotation(Ang_prev); // Crea matriz de rotación con los ángulos de
salida del EKF

Mat Acc_wrt_origin = Rot*Acc_wrt_prev; // Pasa aceleración a base global

Acc_wrt_origin.at<double>(2) = Acc_wrt_origin.at<double>(2)-9.81; //Elimina gravedad

Acc_wrt_prev = Rot.t()*Acc_wrt_origin; //Devuelve aceleración a base IMU

// Publica los valores de la IMU calibrados y sin gravedad

pub_msg = *msg;

pub_msg.angular_velocity.x+=0.00054;
pub_msg.angular_velocity.y+=-0.00585;
pub_msg.angular_velocity.z+=0.00083;

pub_msg.linear_acceleration.x += Acc_wrt_prev.at<double>(0)+0.18784;
pub_msg.linear_acceleration.y += Acc_wrt_prev.at<double>(1)+0.22967;
pub_msg.linear_acceleration.z += Acc_wrt_prev.at<double>(2)-0.11516;


IMU_mod_pub.publish(pub_msg);


}

int main(int argc, char **argv)
{
 // Configuración de nodos y topics de ROS
  ros::init(argc, argv, "imu_listener");
  ros::NodeHandle n;
  ros::Subscriber subIMU = n.subscribe("/erlecopter/imu", 1, IMUCallback);
  ros::Subscriber subEKF = n.subscribe("/odometry/filtered", 1, OrientationCallback);
  IMU_mod_pub=n.advertise<sensor_msgs::Imu>("imu_modified",2);
  ros::spin();

  return 0;
}
```

5. Archivo EKF_receiver.cpp

Guarda en un archivo de texto los valores de posición y ángulos estimados por el nodo que aplica el EKF. Por ello, tiene que subcribirse a su *topic* /odometry/filtered.

```
#include <ros/ros.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include "nav_msgs/Odometry.h"
#include "Euler.hpp"

std::ofstream file;

void Callback(const nav_msgs::Odometry::ConstPtr& msg)
{

      EulerAngles a = Quat2Euler(msg->pose.pose.orientation);
      geometry_msgs::Point p = msg->pose.pose.position;

      file.open("/home/javier/pruebas/MasPruebas/Data/EKF_data.txt",
      std::ios_base::app);
      file << msg->header.stamp << "," << a.roll << "," << a.pitch << "," << a.yaw <<
      "," << p.x << "," << p.y << "," << p.z << std::endl;
      file.close();

}


int main(int argc, char **argv)
{
  ros::init(argc, argv, "EKF_receiver");
  ros::NodeHandle nh;
  ros::Subscriber sub_h = nh.subscribe("/odometry/filtered", 10, Callback);
  ros::spin();
}
```