Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías Industriales

Control servo-visual de una plataforma giroestabilizada

Autor: José Ignacio Murillo Álvarez

Tutor: Daniel Rodríguez Ramírez

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

Sevilla, 2021

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Control servo-visual de una plataforma giroestabilizada

Autor:
José Ignacio Murillo Álvarez

Tutor:
Daniel Rodríguez Ramírez
Profesor titular

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo Fin de Grado: Control servo-visual de una plataforma giroestabilizada

Autor: José Ignacio Murillo Álvarez

Tutor: Daniel Rodríguez Ramírez

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

En primer lugar, agradecer a toda mi familia el esfuerzo que supone educar, formar y promover el estudio en una persona desde que es un niño hasta que se convierte en un adulto. Gracias a mis amigos, por permitirme desconectar de las duras horas de estudio como ellos mejor saben. A Rocío, por apoyarme y darme fuerzas durante todo el desarrollo del TFG, que no ha sido fácil para ninguno. A Juan, por el apoyo técnico en materia audiovisual. También agradecer a todos los profesores de la Escuela que me hayan hecho un poco mejor ingeniero en cada una de mis etapas, especialmente a Daniel, el tutor de este trabajo. Finalmente, agradecer a Hernán por toda la ayuda con el proyecto, así como por poder haber aprendido muchísimo de él durante todo el tiempo que hemos trabajado juntos.

José Ignacio Murillo Álvarez

Sevilla, 2021

Resumen

El control servo-visual consiste en utilizar una referencia de imagen obtenida a través de un elemento fotográfico, como una cámara de vídeo, para mover un servomecanismo de la forma deseada en función a la información obtenida. Este tipo de control es usado en plataformas giroestabilizadas (gimbal) para, por ejemplo, realizar el seguimiento de un objetivo dado.

La integración de este tipo de plataformas es muy común en los UAV (Vehículos Aéreos No Tripulados), ya que compensan en movimiento del vehículo para ofrecer una imagen estabilizada.

En el presente trabajo, se aborda la implementación de un controlador servo-visual en una plataforma giroestabilizada de tres grados de libertad. El algoritmo pretende centrar en el plano imagen un objeto detectado, tratando de obtener a su vez una estimación de la posición real del objetivo con respecto a la plataforma.

El problema se compone de tres grandes frentes:

El primero, conseguir el correcto control de la plataforma y su convergencia hacia las orientaciones deseadas. Para diseñar un controlador para la plataforma, hemos usado la herramienta Matlab-Simulink de cara a realizar simulaciones que nos permitan validarlo.

El segundo, reconocer un objetivo a seguir mediante un flujo de vídeo, conseguir una referencia en base a esta detección y proporcionársela al controlador diseñado anteriormente. En este apartado, se incluye la estimación de la posición en 3D del objetivo, la cual obtendremos mediante métodos basados en fotogrametría.

El tercero, integrar los diferentes elementos del sistema y permitir la comunicación entre estos, para lo cual hemos usado la herramienta ROS (Robotic Operating System).

Finalmente, hemos realizado diferentes pruebas en el sistema real para validar la correcta funcionalidad del algoritmo de control servo-visual diseñado.

Abstract

The visual-servo control consist in extract information from a image, which is obtain by a sensor, like a videocamera, to move a servomechanism in the desired way according to what is observed. This type of control is used in gyro-stabilized platforms (gimbal) in order to, for example, target tracking.

The integration of gimbals is very common in UAV (Unmanned Aerial Vehicle), because it compensate the movement of the vehicle to give a stabilized image.

In this work, we deal with the implementation of a visual-servo controller in a three degrees of freedom gimbal. The algorithm pretens to center a detected object in the center of the image plane, getting also a estimation of the real target position relative to the gimbal.

The problem consists of three parts:

First, achieve the correct control of the gimbal and its convergency to the desired attitude. With the purpose of design a controller, we have used the software tool Matlab-Simulink to simulate and validate it.

Second, perceive a target to tracking from a video stream, obtain a desired attitude based on this detection and provide it to the controller previously designed. In this part, its include the estimation in three dimensions of the object position, which we obtain by photogrammetry methods.

Third, to integrate the elements that form the system and allow the communication between them. We have used the software ROS (Robotic Operating System) to solve this problema.

Finaly, we have done several test in the real system to validate the correct performance of the visual-servo control algorithm designed.

Índice

Agradecimientos ix
Resumen xi
Abstract xiii
Índice xiv
Índice de Tablas xvi
Índice de Figuras xvii
Notación xix
1 Introducción 1
1.1 Objetivos 1
1.2 Metodología 1
1.3 Sumario 2
2 Modelo de la plataforma 3
2.1 Herramientas matemáticas 3
2.1.1 Ángulos de Euler 3
2.1.2 Cuaternios 5
2.2 Modelo geométrico 8
2.2.1 Modelo con Ángulos de Euler 8
2.2.2 Modelo con Cuaternios 9
2.2.3 Ventajas de trabajar con Cuaternios 9
2.3 Descripción del sistema real 12
3 Modelo de la cámara 15
3.1 Cámara Pinhole 15
3.2 Cámara con sensor CCD o CMOS 16
3.3 Descripción del Sistema real 18
4 Control de la plataforma 19
4.1 Diseño del controlador 19
4.2 Simulación del controlador 21
5 Estimación de la posición del objetivo 25
5.1 Visión por computadora 25
5.1.1 OpenCV 26
5.1.2 AprilTag 26
5.2 Triangulación 27
5.3 Calibración 28
5.4 Obtención de la posición del objetivo. Primer método. 30
5.5 Obtención de la posición del objetivo. Segundo método. 34
5.5.1 Filtro de Kalman Adaptativo 35
6 Integración del sistema 38
6.1 Comunicación con la plataforma. MAVLink 38
6.2 Robot Operating System (ROS) 39

6.3 Sistema completo. Nodos, tópicos, mensajes y funcionamiento 41
6.3.1 Nodo ros_gremsy 42
6.3.2 Nodo control_gremsy 42
6.3.3 Nodo Procesamiento 43
6.3.4 Nodo Grabación 44
6.3.5 Nodo Sincronización 45
6.3.5 Nodo Parámetros 45
6.3.6 Nodo usb_cam 45
6.3.7 Esquema general 46
7 Resultados 47
7.1 Prueba 1. Objetivo siempre en el rango de visión 47
7.2 Prueba 2. El objetivo escapa del rango de vision 54
7.3 Prueba 3. UAV en movimiento 56
7.2 Prueba 4. UAV en movimiento. Control en posición 57
8 Conclusión y trabajos futuros 61
Referencias 62
Anexos 65
Anexo A: Triangulación 65
Anexo B: Nodo Control_gremsy 69
Anexo C: Nodo Procesamiento 78
Anexo D: Nodo Grabación 84
Anexo E: Nodo Sincronización 86
Anexo F: Nodo Parámetros 87
Anexo G: Nodo Control_gremsy. Control en posición 89

ÍNDICE DE TABLAS

TABLA 2–1. CARACTERÍSTICAS MECÁNICAS Y ELÉCTRICAS DE PIXY U 12
TABLA 2–2. CARACTERÍSTICAS DE TRABAJO DE PIXY U 13
TABLA 3–1. ESPECIFICACIONES CÁMARA SONY RX0 II 18
TABLA 5–1. POSICIÓN RELATIVA DE LA CÁMARA EN LA TOMA DE IMÁGENES 32
TABLA 5–2. RESULTADOS DE LA TRIANGULACIÓN (METROS) 33
TABLA 7–1. PARÁMETROS DE CONTROL. PRUEBA 1 47

ÍNDICE DE FIGURAS

FIGURA 2-1. GIRO POSITIVO DE ÁNGULO $\boldsymbol{\theta}$ DE UN VECTOR $\boldsymbol{vo}$ [6] 3
FIGURA 2-2. ROTACIONES CON ÁNGULOS DE EULER [41] 4
FIGURA 2-3. ÁNGULOS DE EULER EN UN AVIÓN [42] 4
FIGURA 2-4. INSCRIPCIÓN EN BROOM BRIDGE, DUBLÍN, EN REFERENCIA A LA FÓRMULA DE LOS CUATERNIOS ESCRITA POR HAMILTON 5
FIGURA 2-5. ROTACIÓN DE UN CUERPO EJE-ÁNGULO [44] 7
FIGURA 2-6. ESQUEMA DEL MODELO GEOMÉTRICO DE LA PLATAFORMA CON ÁNGULOS DE EULER 8
FIGURA 2-7. ESQUEMA DEL MODELO GEOMÉTRICO DE LA PLATAFORMA CON CUATERNIOS 9
FIGURA 2-8. APOLLO IMU [11] 10
FIGURA 2-9. PIXY U 12
FIGURA 3-1. MODELO DE LA CÁMARA PINHOLE [13] 15
FIGURA 3-2. MODELO DE CÁMARA OSCURA O ESTENOPEICA [13] 16
FIGURA 3-3. MODELADO DE DISTORSIÓN DE LENTE EN COMPONENTE RADIAL Y TANGENCIAL [13] 17
FIGURA 3-4. CÁMARA SONY RX0 II [17] 18
FIGURA 4-1. REPRESENTACIÓN DEL CUATERNIO $\boldsymbol{qt}$ 20
FIGURA 4-2. ESQUEMA EN SIMULINK DE LA SIMULACIÓN DEL CONTROLADOR 21
FIGURA 4-3. OBTENCIÓN DEL CUATERNIO DESEADO EN LA SIMULACIÓN 22
FIGURA 4-4. ESQUEMA PARA EL CÁLCULO DE LAS VELOCIDADES ANGULARES DESEADAS 22
FIGURA 4-5. REPRESENTACIÓN DE $\boldsymbol{qt}$, $\boldsymbol{qe}$, $\boldsymbol{q}$ EN LA SIMULACIÓN 23
FIGURA 4-6. REPRESENTACIÓN DEL VECTOR DESEADO, VECTOR ACTUAL Y EL PRODUCTO VECTORIAL 24
FIGURA 4-7. REPRESENTACIÓN DEL ÁNGULO DE ERROR Y LA VELOCIDAD ANGULAR DESEADA 24
FIGURA 5-1. ETAPAS DEL PROCESO DE ANÁLISIS DE IMÁGENES [47] 25
FIGURA 5-2. EJEMPLO DE RECONOCIMIENTO DE APRILTAGS EN UNA IMAGEN 27
FIGURA 5-3. TRIANGULACIÓN CON DOS IMÁGENES [16] 28
FIGURA 5-4. PATRÓN USADO PARA LA CALIBRACIÓN. TABLERO DE AJEDREZ 10X7 29
FIGURA 5-5. EJEMPLO DE LAS IMÁGENES TOMADAS PARA LA CALIBRACIÓN 30
FIGURA 5-6. EJE DE COORDENADAS DE LA CÁMARA Y DE LA IMAGEN 31
FIGURA 5-7. IMÁGENES TOMADAS PARA LA TRIANGULACIÓN. RESPECTIVAMENTE, IMAGEN 1,2 Y 3 32
FIGURA 5-8. IMÁGENES USADAS PARA EL CÁLCULO DE LA TRIANGULACIÓN 33
FIGURA 5-9. REPRESENTACIÓN EN EL ESPACIO DEL RESULTADO DE LA TRIANGULACIÓN 34
FIGURA 5-10. ITERACIÓN EN FILTRO DE KALMAN [50] 36
FIGURA 6-1. ESTRUCTURA DE UN MENSAJE MAVLINK [32] 38
FIGURA 6-2. COMUNICACIÓN ENTRE DOS NODOS, POR MENSAJES A TRAVÉS DE UN TÓPICO [51] 40
FIGURA 6-3. DIFERENCIA ENTRE COMUNICACIÓN POR TOPIC Y SERVICIOS [36] 41
FIGURA 6-4. ESQUEMA DEL PROGRAMA DE TRACKING 41
FIGURA 6-5. RECTÁNGULO USADO REPRESENTADO EN EL PLANO IMAGEN 43
FIGURA 6-6. INTERFAZ GRÁFICA PARA INTRODUCIR LOS PARÁMETROS DESEADOS 45
FIGURA 6-7. RQT_GRAPH DURANTE EL FUNCIONAMIENTO DEL PROGRAMA 46
FIGURA 6-8. NODOS Y TÓPICOS DEL PROGRAMA COMPLETO 46
FIGURA 7-1. REPRESENTACIÓN GRÁFICA DE CUATERNIOS. PRUEBA 1 48
FIGURA 7-2. REPRESENTACIÓN GRÁFICA DEL VECTOR DE APUNTAMIENTO. PRUEBA 1 48
FIGURA 7-3. REPRESENTACIÓN DE LAS VELOCIDADES ANGULARES. PRUEBA 1 49
FIGURA 7-4. REPRESENTACIÓN DE LA ORIENTACIÓN EXPRESADA EN ÁNGULOS DE EULER. PRUEBA 1 49
FIGURA 7-5. REPRESENTACIÓN GRÁFICA DEL VECTOR DE POSICIÓN DEL OBJETIVO ESTIMADO. PRUEBA 1. 50
FIGURA 7-6. REPRESENTACIÓN GRÁFICA DEL VECTOR DE POSICIÓN. AMPLIADO. PRUEBA 1 51
FIGURA 7-7. REPRESENTACIÓN DE LA DISTANCIA EN PÍXELES AL CENTRO DEL OBJETO EN EL PLANO IMAGEN. PRUEBA 1 51
FIGURA 7-8. TRAZA DE LA MATRIZ P. PRUEBA 1 52
FIGURA 7-9. TRAZA MATRIZ P. AMPLIADO. PRUEBA 1 52

FIGURA 7-10. TRAZA MATRIZ K. PRUEBA 1. 53
FIGURA 7-11. COMPONENTES DE LA MATRIZ A. PRUEBA 1 53
FIGURA 7-12. IMÁGENES DE LA PRUEBA 54
FIGURA 7-13. REPRESENTACIÓN GRÁFICA DE CUATERNIOS. PRUEBA 2 55
FIGURA 7-14. REPRESENTACIÓN GRÁFICA DEL VECTOR DE POSICIÓN. AMPLIADO. PRUEBA 1 55
FIGURA 7-15. REPRESENTACIÓN GRÁFICA DE CUATERNIOS. PRUEBA 3 56
FIGURA 7-16. REPRESENTACIÓN GRÁFICA DE ORIENTACIÓN EN ÁNGULOS DE EULER. PRUEBA 3 57
FIGURA 7-17. REPRESENTACIÓN DE CUATERNIOS. PRUEBA 4 58
FIGURA 7-18. REPRESENTACIÓN DEL VECTOR DEL OBJETIVO RELATIVO A LA CÁMARA. PRUEBA 4 59
FIGURA 7-19. REPRESENTACIÓN DE LA DISTANCIA EN PÍXELES AL CENTRO DEL OBJETO EN EL PLANO IMAGEN. PRUEBA 4 59

Notación

A^*	Conjugado
sin	Función seno
cos	Función coseno
ln	Función logaritmo natural
e	Número e
$\vec{A}$	Vector
A^T	Vector o matriz traspuesta
$\mathbb{R}$	Números reales
$\|\|A\|\|$	Norma 2
tanh	Tangente hiperbólica
$\times$	Producto vectorial

1 INTRODUCCIÓN

En los últimos años, el uso de los UAV (Unmaned Aerial Vehicle), también conocidos como drones, ha tenido un notable aumento, mostrándose muy útiles en diversas tareas. Entre ellas, la agricultura de precisión [1], en la cual se busca la toma de imágenes georeferenciadas para su posterior procesamiento y usarlas como herramienta para optimizar los cultivos. Por su naturaleza, los UAV son idóneos para este tipo de trabajos. Otra de las actividades más desarrolladas por estos vehículos aéreos es el seguimiento de objetivos móviles [2], donde se utilizan las ventajas de estos robots para seguir y detectar objetivos diferentes desde el aire. También es digno de mención el uso de los UAV para servicios de reparto [3].

Para desempeñar todas estas tareas de la forma óptima y sin inconvenientes, es necesario disponer de diferentes dispositivos que surtan al UAV de información obtenida de su entorno. Uno de los sensores más usados son las cámaras de vídeo, las cuales ayudan a los drones a conocer cómo se estructura su alrededor. Viendo las actividades que hemos presentado en el párrafo anterior, nos podemos hacer una idea de la importancia de que un UAV lleve una cámara, ya sea para obtener imágenes o para obtener un flujo de vídeo en directo que nos permita reconocer o detectar un objetivo que deseamos seguir.

Debido a las perturbaciones que un dron soporta durante el vuelo, movimientos bruscos o vientos, la cámara acoplada al sistema debe estar integrada en una plataforma giroestabilizada (gimbal).

Una plataforma giroestabilizada, o gimbal, es un dispositivo electromecánico cuyo objetivo es el mantener estable una cámara ante perturbaciones debidas a, por ejemplo, fricciones en los motores, aerodinámicas no balanceadas, fuerzas de torsión y otras. Las cámaras estabilizadas mediante gimbal se usan para diversos escenarios, tanto civiles como militares: fotografía aérea, vigilancia, seguimiento de misiles y navegación autónoma. Si la cámara no se estabiliza correctamente, es imposible capturar las imágenes de forma nítida en estas aplicaciones, lo cual es clave en su exitosa puesta en marcha [4].

1.1 Objetivos

El problema a resolver en este trabajo será el de un algoritmo de tracking mediante el uso de una plataforma gimbal de tres grados de libertad, en la cual estará acoplada una cámara de video.

El algoritmo de control debe fijar el objetivo en el centro del plano de la imagen, de forma que facilite un seguimiento correcto de este, así como permita realizar diferentes procesos sobre la imagen deseada.

En primer lugar, debemos acometer la detección del objetivo mediante el uso del flujo de vídeo obtenido mediante la cámara. Una vez reconocido, solo conoceremos su posición en el plano imagen, por lo que estimaremos la posición que tiene respecto a la plataforma, de forma que nos facilite el control de esta.

Una vez tengamos esta posición, actuaremos sobre la plataforma giroestabilizada para conseguir centrar el objetivo en la imagen.

1.2 Metodología

Para abordar los objetivos descritos anteriormente, necesitaremos diversas herramientas.

En primer lugar, para resolver los problemas geométricos y cinemáticos propios del control del gimbal, utilizaremos herramientas como los Ángulos de Euler, y, especialmente, los Cuaternios.

La detección del objetivo la llevaremos a cabo mediante la librería AprilTag, que nos permite usar algoritmos de detección sin tener que programarlos por nuestra propia mano, lo cual se escapa del alcance de este trabajo.

Acometeremos la obtención de la posición del objetivo con técnicas de fotogrametría, como la triangulación.

La comunicación entre los diferentes sistemas la implementaremos mediante ROS (Robot Operating System), un sistema operativo que ofrece un protocolo de comunicaciones y sincronización entre las diferentes partes de

un sistema robótico.

Para validar el cumplimiento de los objetivos, llevaremos a cabo algunas simulaciones mediante Matlab-Simulink y, fundamentalmente, realizaremos experimentos con la plataforma gimbal real que vamos a controlar.

1.3 Sumario

En el Capítulo 2, trataremos de obtener un modelo geométrico de la plataforma que nos permita representar la orientación de esta, así como una herramienta para poder desarrollar el algoritmo de control, meta del trabajo. También presentaremos el modelo real de la plataforma que vamos a usar.

A continuación, en el Capítulo 3, definiremos el modelo geométrico de la cámara que vamos a usar. Análogamente a lo que se hizo en el Capítulo 2, presentaremos el modelo real de este dispositivo.

Continuaremos con el Capítulo 4, en el cual se presentará el algoritmo de control para la plataforma gimbal.

Luego, en el Capítulo 5, seguiremos con la obtención de la posición del objetivo con respecto a la plataforma mediante técnicas de fotogrametría.

Posteriormente, en el Capítulo 6, definiremos las comunicaciones y trataremos la integración de los diferentes dispositivos que hemos definido, la cámara, la plataforma y la CPU encargada de ejecutar los programas.

Una vez hemos definido el sistema completo, mostraremos y analizaremos los resultados de los experimentos y pruebas llevados a cabo en el sistema real, todo ello en el Capítulo 7.

Para terminar, en el Capítulo 8, finalizaremos con las conclusiones y los posibles trabajos futuros que derivan de este trabajo.

2 MODELO DE LA PLATAFORMA

Para comenzar, es necesario desarrollar y definir correctamente el modelo geométrico de la plataforma que vamos a usar en este trabajo, y que nos será necesario de cara a diseñar el algoritmo de control que tenemos como objetivo.

2.1 Herramientas matemáticas

Vamos a presentar dos de las herramientas más usadas en este tipo de sistemas para representar giros y orientaciones. Los Ángulos de Euler y los Cuaternios.

2.1.1 Ángulos de Euler

El Teorema de Rotación de Euler afirma que la rotación arbitraria de un sólido puede ser descrita por tres parámetros. Estos parámetros son los Ángulos de Euler [5].

Estas rotaciones pueden ser escritas en forma de matrices de rotación. Una matriz de rotación representa el giro de un ángulo θ contrario a las agujas del reloj de un vector v dado [6].

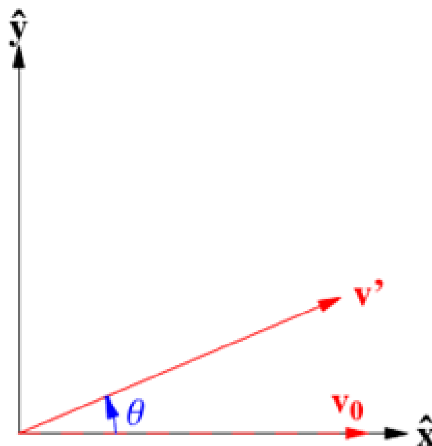

Figura 2-1. Giro positivo de ángulo θ de un vector v_0 [6]

La matriz de rotación que representa este giro es:

$$R_\theta = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{2.1}$$

Las matrices correspondientes a giros en tres dimensiones corresponden a las siguientes:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \tag{2.2}$$

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \tag{2.3}$$

$$R_z(\gamma) = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Los Ángulos de Euler podemos dibujarlos de esta forma:

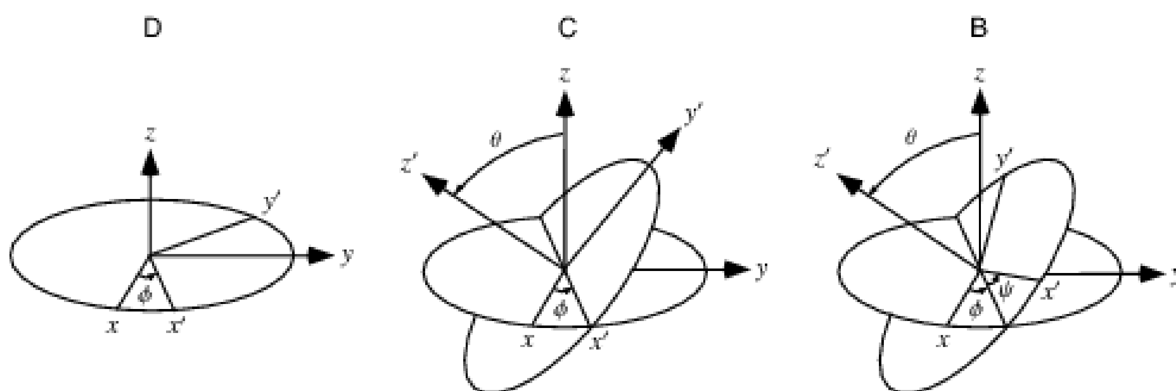

Figura 2-2. Rotaciones con Ángulos de Euler [41]

Para entenderlos mejor, veamos un ejemplo de cómo se usarían estos parámetros para definir las rotaciones que tienen lugar en un avión:

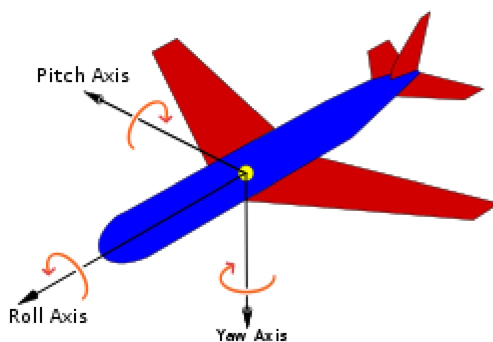

Figura 2-3. Ángulos de Euler en un avión [42]

Usando matrices de rotación, podemos establecer las rotaciones de la figura como:

$$D = R_z(\phi)$$
$$C = R_x(\theta) \tag{2.5}$$
$$B = R_y(\psi)$$

Donde ϕ, θ, ψ son los Ángulos de Euler (respectivamente, yaw, roll, pitch).

La matriz que expresa la rotación total del sólido, siguiendo el orden de rotaciones de la figura anterior es:

$$A = B * C * D \tag{2.6}$$

2.1.2 Cuaternios

En 1843, William Rowan Hamilton, creaba una versión en tres dimensiones de los números complejos, siendo llamados números hipercomplejos, escribiendo una famosa ecuación en una piedra del Broom Bridge, Dublín, en la que se basaría el algebra de los cuaternios [7].

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.7}$$

Figura 2-4. Inscripción en Broom Bridge, Dublín, en referencia a la fórmula de los cuaternios escrita por Hamilton

Los cuaternios se pueden denotar como:

$$q = q_o + \mathbf{q} = q_o + iq_1 + jq_2 + kq_3 \tag{2.8}$$

Donde i, j, k forman la base ortonormal estándar en $\mathbb{R}^3$.

En este caso, q_o es llamado la parte escalar del cuaternio, mientras que $\mathbf{q}$ es llamado la parte vectorial del cuaternio. Los escalares q_o, q_1, q_2, q_3 son llamados los componentes del cuaternio.

2.1.2.1 Conjugado del cuaternio

El conjugado del cuaternio:

$$q = q_o + \mathbf{q} = q_o + iq_1 + jq_2 + kq_3 \tag{2.9}$$

Es denotado como q^*, y obtenido como:

$$q = q_o - \mathbf{q} = q_o - iq_1 - jq_2 - kq_3 \tag{2.10}$$

Este conjugado cumple:

$$(pq)^* = q^* p^* \tag{2.11}$$

$$(p^* q)^* = q^* p \tag{2.12}$$

2.1.2.2 Norma del cuaternio

La norma de un cuaternio q se denota como el escalar *N(q):*

$$N(q) = \sqrt{q^* q} \tag{2.13}$$

O, en su defecto:

$$N^2(q) = q^* q \tag{2.14}$$

2.1.2.3 Cuaternio unitario

Un cuaternio unitario q tiene una norma igual a uno:

$$|q| = |q^*| = 1 \quad y \quad N^2(q) = q^* q = 1 \tag{2.15}$$

El resultado de los productos entre cuaternios unitarios son también cuaternios unitarios.

2.1.2.4 Cuaternio inverso

Por definición de una inversión tenemos que $q^{-1} q = q q^{-1} = 1$. Si premultiplicamos ambos lados de la segunda forma de la ecuación (2.9):

$$q^* q q^{-1} = N^2(q) q^{-1} = q^* \tag{2.16}$$

De aquí temenos que, como $q^* q = 1$:

$$q^{-1} = \frac{q^*}{N^2(q)} \tag{2.17}$$

Si q es un cuaternio unitario, entonces:

$$q^{-1} = q^* \tag{2.18}$$

2.1.2.5 Suma de cuaternios

La suma de dos cuaternios, $q = (q_o, \mathbf{q})$ y $p = (p_o, \mathbf{p})$, se define como [8]:

$$q + p = (q_o + p_o, \mathbf{q} + \mathbf{p}) \tag{2.19}$$

2.1.2.6 Producto de cuaternios

El producto de dos cuaternios, $q = (q_o, \mathbf{q})$ y $p = (p_o, \mathbf{p})$, esta definido como:

$$q \circ p = (q_o p_o - \mathbf{q} \cdot \mathbf{p}, q_o \mathbf{p} + p_o \mathbf{q} + \mathbf{q} + \mathbf{q} \times \mathbf{p}) \tag{2.20}$$

Nótese que el producto de dos cuaternios no es conmutativo.

2.1.2.7 Operaciones con vectores

Un vector $x = (x_1, x_2, x_3)$ puede ser expresado como un cuaternio con parte real igual a 0, es decir:

$$x = (0, x_1, x_2, x_3) \tag{2.21}$$

La siguiente operación realizada con un cuaternio unitario q representa una rotación del vector x:

$$x' = q \circ x \circ q^{-1} = q \circ x \circ q^* \tag{2.22}$$

El resultado es un cuaternio x' que representa un vector de la misma longitud que x.

2.1.2.8 Representación ángulo-eje

Considerando la rotación de la figura como un vector $\vec{e} = [\theta_x \ \theta_y \ \theta_z]^T$ con magnitud $\theta = \|\vec{\theta}\|$ en radianes, actuando en un eje representado como un vector unitario $\vec{u} = \vec{e}/\|\vec{\theta}\|$, la representación ángulo-eje de esta rotación se denota como $\vec{e} = \theta\vec{u}$.

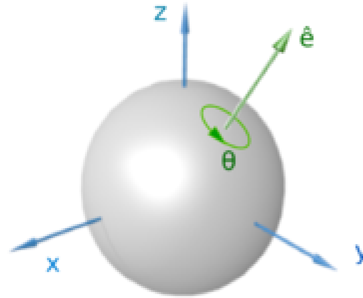

Figura 2-5. Rotación de un cuerpo eje-ángulo [44]

La formula de Euler-Rodrigues es el mapeo exponencial de la representación ángulo-eje para una rotación [9]. Se define como:

$$q = e^{\frac{1}{2}\theta\vec{u}} = \cos(\theta/2) + \vec{u}\sin(\theta/2) \tag{2.23}$$

Como $\|q\| = 1$, se trata de un cuaternio unitario.

De forma inversa, podemos obtener la representación ángulo-eje a partir de un cuaternio, usando el mapeo logarítmico:

$$\vec{\theta} = 2\ln q \tag{2.24}$$

$$\ln q = \begin{cases} [0\ 0\ 0]^T, si \ \|\vec{\mathbf{q}}\| = 0 \\ \dfrac{\vec{\mathbf{q}}}{\|\vec{\mathbf{q}}\|}\operatorname{acos}(q_o), si \ \|\vec{\mathbf{q}}\| \neq 0 \end{cases} \tag{2.25}$$

Recordamos que, $q = (q_o, \mathbf{q})$.

2.2 Modelo geométrico

Basándonos en las herramientas expuestas en apartado 2.1, vamos a definir el modelo geométrico de la plataforma.

2.2.1 Modelo con Ángulos de Euler

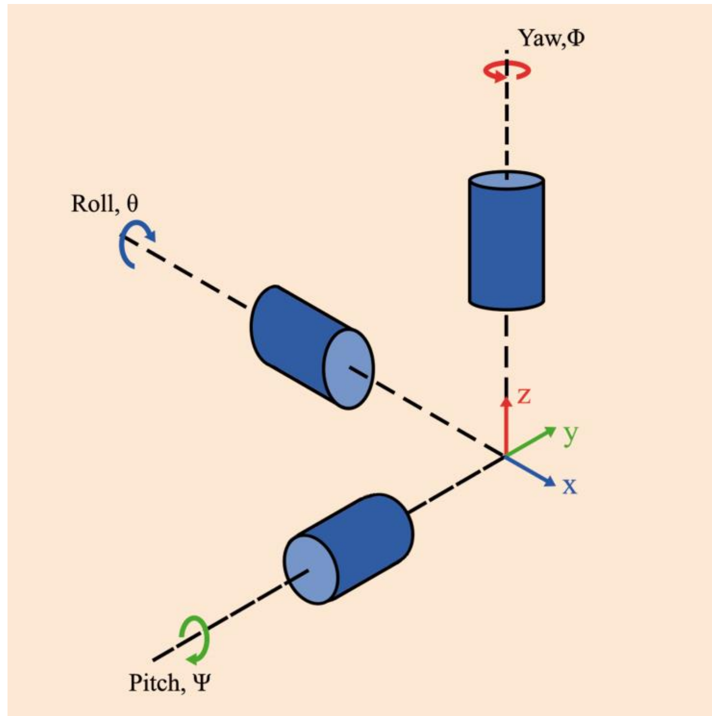

Figura 2-6. Esquema del modelo geométrico de la plataforma con Ángulos de Euler

En la Figura 2-6, podemos observer cómo sería el modelo geométrico de la plataforma usando los Ángulos de Euler, formando un sistema dextrógiro.

2.2.2 Modelo con Cuaternios

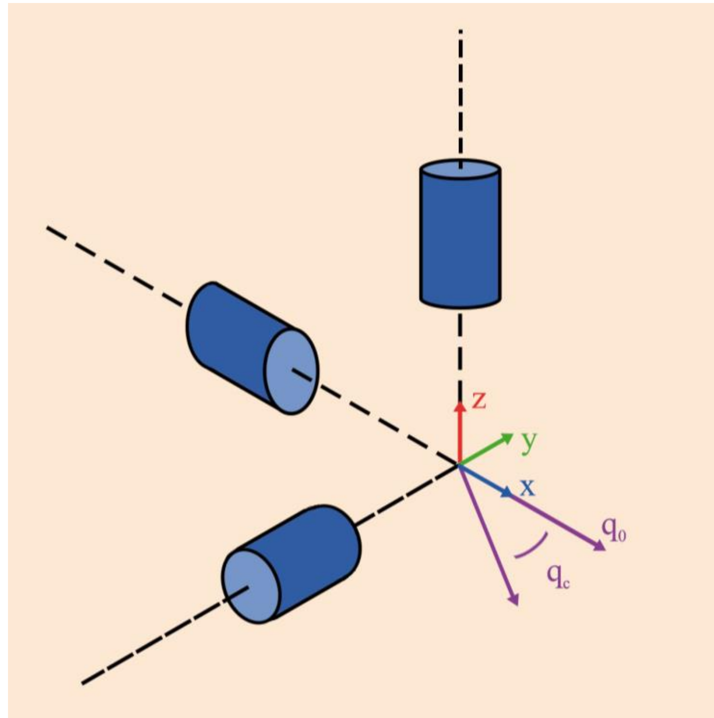

Figura 2-7. Esquema del modelo geométrico de la plataforma con Cuaternios

En la figura 2-7, podemos observar cómo se representaría el mismo sistema con cuaternios. El cuaternio q_0 representaría la orientación inicial, es decir, un giro nulo.

El cuaternio q_c representaría la orientación de la plataforma con respecto a la orientación elegida como origen. Ese cuaternio expresa el giro entre el vector de apuntamiento inicial y el vector de apuntamiento actual.

Por definición, $q_0 = (1,0,0,0)$.

2.2.3 Ventajas de trabajar con Cuaternios

En este trabajo, se va a trabajar principalmente usando el modelo de la plataforma basado en cuaternios. Las ventajas por las que se ha elegido este sistema son [10]:

- Los cuaternios son más compactos que las matrices de rotación. (Las matrices, por ejemplo la ecuación 2.4, se componen de 9 elementos, mientras que los cuaternios se componen de 4 elementos, visto en la ecuación 2.8.
- Las matrices de rotación introducen más operaciones aritméticas, por lo que el tiempo de computación es mayor.
- Extraer el ángulo y el eje de rotación es más simple usando cuaternios.
- La normalización de una matriz de rotación es más costosa computacionalmente hablando que la normalización de un cuaternio.
- Se evitan singularidades matemáticas que pueden conllevar un “Gimbal Lock”.

El Gimbal Lock ocurre cuando uno de los ejes de rotación se alinea con otro, lo que causa la pérdida de un grado de libertad. Es muy conocido el suceso en el que se vio involucrada la misión Apollo 11 [11], la encargada de llevar a la raza humana a la Luna por primera vez. La IMU (Unidad de medición inercial), estaba formada por una plataforma gimbal de tres grados de libertad. Este dispositivo proveía a los astronautas de la

orientación en la que estaba situada la nave en cualquier momento. Teóricamente, el sistema que iba a proteger a la IMU de entrar en Gimbal Lock era simplemente girar 180 grados cuando el ángulo de pitch estuviera cercano a 85 grados. Sin embargo, el ordenador de a bordo advirtió de un Gimbal Lock cuando se encontraban a 75 grados, y detuvo completamente la IMU cuando estaban en 85 grados. Entonces, la nave tenía que ser manualmente apartada de la posición de Gimbal Lock.

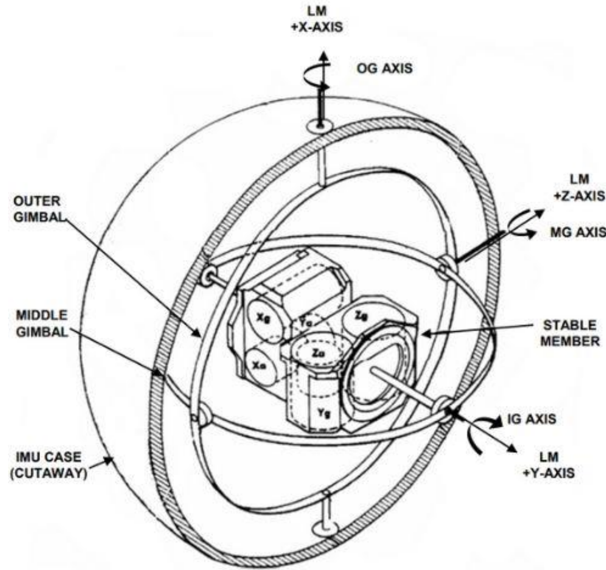

Figura 2-8. Apollo IMU [11]

Podemos ver en el siguiente caso, la diferencia entre las matrices de rotación y los cuaternios al entrar en la singularidad mencionada.

Si usamos una secuencia de rotación XYZ, rotamos en el eje X 0 grados, $\theta = 0$; rotamos en el eje Y 90 grados, $\psi = 90$; rotamos en el eje Z ϕ grados, $\phi = \phi$. Obtenemos una matriz de rotación como esta:

$$R_{XYZ} = \begin{bmatrix} 0 & 0 & 1 \\ -\sin(\phi) & \cos(\phi) & 0 \\ \cos(\phi) & \sin(\phi) & 0 \end{bmatrix} \quad (2.26)$$

Considerando ahora una rotación en el eje X de θ grados, $\theta = -\theta$; rotación en el eje Y de 90 grados de nuevo, $\psi = 90$; rotación en el eje Z de 0 grados, $\phi = 0$.

$$R_{XYZ} = \begin{bmatrix} 0 & 0 & 1 \\ -\sin(\theta) & \cos(\theta) & 0 \\ \cos(\theta) & \sin(\theta) & 0 \end{bmatrix} \quad (2.27)$$

Nos queda una matriz de rotación igual a la de la ecuación 2.25. Esto ocurre porque la rotación de 90 grados sobre el eje Y causa que el eje Z y el X se alineen, por lo que cualquier rotación en el eje Z y en el eje X trae los mismos resultados, lo que causa la singularidad.

Ahora veamos el resultado que arrojan los cuaternios en este caso.

Una secuencia de rotación XYZ en cuaternios es dada por [10]:

$$Q_{XYZ} = Q_X * Q_Y * Q_Z \quad (2.28)$$

Aplicando lo que se muestra en la ecuación 2.23 para cada ángulo de giro, obtenemos ese cuaternio, que podemos particularizar para los dos casos anteriores.

En el caso 1, $\theta = 0, \psi = 90, \phi = \phi$:

$$q_o{}^{XYZ} = \frac{1}{\sqrt{2}}\cos\left(\frac{\phi}{2}\right)$$

$$q_1{}^{XYZ} = -\frac{1}{\sqrt{2}}\sin\left(\frac{\phi}{2}\right) \tag{2.29}$$

$$q_2{}^{XYZ} = \frac{1}{\sqrt{2}}\cos\left(\frac{\phi}{2}\right)$$

$$q_3{}^{XYZ} = \frac{1}{\sqrt{2}}\sin\left(\frac{\phi}{2}\right)$$

En el caso 2, $\theta = -\theta, \psi = 90, \phi = 0$:

$$q_o{}^{XYZ} = \frac{1}{\sqrt{2}}\cos\left(\frac{\theta}{2}\right)$$

$$q_1{}^{XYZ} = \frac{1}{\sqrt{2}}\sin\left(\frac{\theta}{2}\right) \tag{2.30}$$

$$q_2{}^{XYZ} = \frac{1}{\sqrt{2}}\cos\left(\frac{\theta}{2}\right)$$

$$q_3{}^{XYZ} = -\frac{1}{\sqrt{2}}\sin\left(\frac{\theta}{2}\right)$$

Observamos que las respuestas en ambos casos son diferentes, por lo que se evita la singularidad.

2.3 Descripción del sistema real

La plataforma gimbal que se va a usar en este trabajo es la Pixy U de Gremsy [12]. Es un gimbal que soporta diferentes tipos de cámaras, compacto y ligero, idóneo para el tipo de actividades que se llevan a cabo con UAV, como la que se desarrolla en nuestro proyecto.

Tabla 2–1. Características mecánicas y eléctricas de Pixy U

Corriente de trabajo	Corriente estática: 400mA (@12V) Corriente dinámica: 800mA (@12V) Corriente locked motor: Max 4.0A (@12V)
Temperatura de operación	0º C ~ 50º C
Peso	0.465 kg
Dimensión	112(W) x 145(D) x 200(H)

Figura 2-9. Pixy U

Tabla 2–2. Características de trabajo de Pixy U

Rango de vibración angular	± 0.02°
Carga máxima	0.465 kg
Máxima velocidad de rotación controlada	Eje yaw: 100º/s Eje pitch: 100º/s Eje roll: 100º/s
Rango mecánico	Eje yaw: +330° a -330° Eje pitch: +135° a -45° Eje roll: +45° a -90°
Rango controlado de rotación	Eje yaw: +330° a -330° Eje pitch: +135° a -45° Eje roll: ± 45°

3 MODELO DE LA CÁMARA

Continuando con el capítulo anterior, tras establecer un modelo geométrico de la plataforma con distintas herramientas definidas, en este capítulo vamos a introducir los diferentes modelos matemáticos con los que vamos a trabajar en la introducción en el sistema de la cámara de vídeo, que será la encargada de proporcionar esa información necesaria para que el algoritmo de control actúe de la forma requerida en cada momento.

3.1 Cámara Pinhole

El modelo de cámara pinhole consiste en un centro óptico C en el cual convergen los rayos de la proyección, y un plano de imagen R en el cual la imagen es proyectada. El plano de imagen está ubicado a una distancia focal f del centro óptico y perpendicular al eje óptico [13].

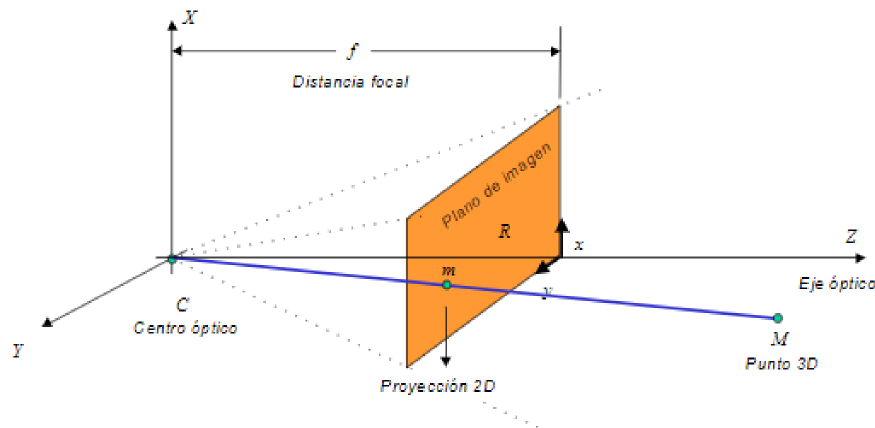

Figura 3-1. Modelo de la cámara pinhole [13]

El punto M en 3D se proyecta en la imagen como el punto m, que es definido por la intersección de la recta que une C y M con el plano de imagen R.

La ecuación que define la relación entre estos dos puntos en coordenadas homogéneas es:

$$Z\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.1}$$

Donde X, Y, Z son las coordenadas en el espacio del punto M; x, y son las coordenadas en el plano imagen del punto m y f es la distancia focal.

Otra forma muy común de representar este modelo de cámara es mediante el modelo geométrico de proyección de cámara oscura o estenopeica, en el cual el centro óptico está situado entre el punto M y el plano de imagen R.

Este modelo está basado en la cámara oscura, un instrumeto óptico que consistía en una caja cerrada y un pequeño agujero en el que entraban los rayos de luz que proyectaban la imagen en la pared opuesta [14].

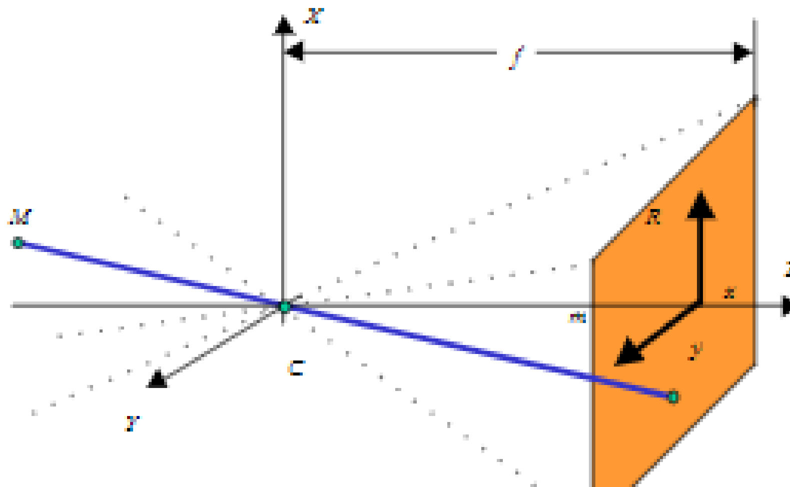

Figura 3-2. Modelo de cámara oscura o estenopeica [13]

En este modelo, es necesario cambiar en la ecuación matricial f por $-f$:

$$Z\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} -f & 0 & 0 & 0 \\ 0 & -f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad (3.2)$$

3.2 Cámara con sensor CCD o CMOS

Las cámaras reales actuales funcionan con elementos semiconductores fotosensibles que forman una matriz. En el caso del CCD, las cargas de las celdas de la matriz se convierten en un voltaje que entrega una señal analógica, que posteriormente es digitalizada por la cámara. En el sensor CMOS, cada celda es independiente, y la transformación analógico-digital tiene lugar en cada una de las celdas mediante un transistor [15].

A la hora de hablar de la formación geométrica de la imagen, tenemos que considerar cuatro aspectos que van a afectar a la misma cuando usamos una cámara de vídeo real [13]:

- *Cambio de escala:* Para transformar las coordenadas que tenemos en una imagen, que están expresadas en píxeles (u, v), a coordenadas en milímetros (x, y), necesitamos un factor de escala. Además, debido a que los píxeles no son cuadrados, sino que son rectangulares, tendremos uno diferente para cada eje de la imagen. Estos factores se expresan como α_x y α_y, expresados en píxel/mm.
- *Traslación del origen:* Tomamos las variables u_o y v_o para definir el punto de origen de la imagen en el nuevo sistema de coordenadas.
- *Rotación de los ejes:* Los ejes x, y y los ejes u, v no tienen la misma orientación. Existen dos métodos en la modelación de la cámara. Uno que supone que el ángulo entre los ejes es cero y que el ajuste debe hacerse en los ejes X, Y, Z y otro que considera un ángulo θ entre los ejes.
- *Factor de torcimiento:* Los ejes u, v pueden no ser ortogonales debido a que los píxeles en los arreglos CCD o CMOS no son completamente rectangulares. En este caso, se introduce un factor de torcimiento (skew) s. Normalmente, este parámetro suele ser cero en la mayoría de las cámaras.

Finalmente, considerando que la orientación de los ejes u, v y x, y es la misma, la transformación de coordenadas está definida por:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_x & s & u_o \\ 0 & \alpha_y & v_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3.3}$$

Donde la matriz definida por los parámetros que hemos listado anteriormente es conocida como la matriz de calibración de la cámara.

También vamos a introducir los parámetros externos de la cámara, que no son más que la transformación entre el eje de coordenadas global y el eje de coordenadas de la cámara [16]:

$$\begin{bmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.4}$$

Donde R es una matriz de rotación 3x3 y t es un vector de translación 3x1.

Finalmente, considerando las ecuaciones (3.2), (3.3) y (3.4), definimos la matriz de proyección desde el espacio en 3D hasta la imagen:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.5}$$

$$P = K[R|t]$$

Otro de los errores que introducen las cámaras comerciales a la proyección de la imagen es la distorsión provocada por la lente. Esto hace que las líneas rectas en 3D ya no sean vistas como líneas rectas en la proyección, sino como líneas curvas.

La distorsión es modelada generalmente como una componente radial y otra tangencial. La distorsión radial asume que un punto ideal (x, y) se proyecta en la imagen sobre la línea radial que une el punto con el centro de la imagen, afectando la distorsión mediante un cambio en la magnitud del radio. La distorsión tangencial asume que el cambio que existe tiene lugar en el ángulo de esa línea radial.

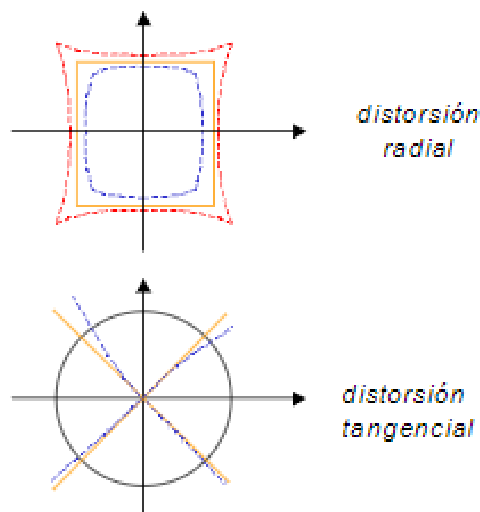

Figura 3-3. Modelado de distorsión de lente en componente radial y tangencial [13]

3.3 Descripción del Sistema real

La cámara que vamos a usar en este proyecto es la Sony RX0 II [17].

Tabla 3–1. Especificaciones cámara Sony RX0 II

Tipo de sensor	Sensor CMOS Exmor RS® tipo 1.0 (13,2 x 8,8 mm)
Número de píxeles efectivos	Aprox. 15,3 megapíxeles
Tipo de lente	Lente ZEISS® Tessar T*
Distancia focal	f = 7,9 mm
Temperatura de funcionamiento	0º C. - +40 º C
Tamaño	59 x 40,5 x 35 mm
Peso	Aprox. 132 g (batería y microSD incluidas) / aprox. 117 g (solo el cuerpo)
Apertura	F4.0
Ángulo de visión	84º

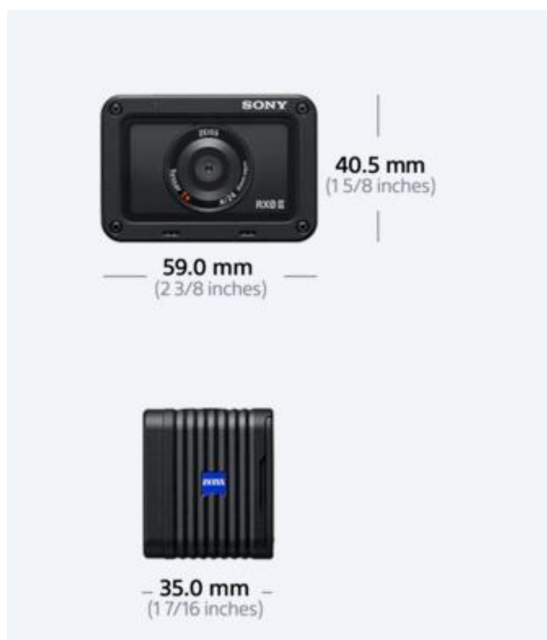

Figura 3-4. Cámara Sony RX0 II [17]

4 CONTROL DE LA PLATAFORMA

El primer problema al que vamos a enfrentarnos en este trabajo es el de diseñar un algoritmo de control para la plataforma gimbal, en el que tendremos en cuenta que la información de la que dispondremos es la que la cámara nos proporciona mediante el flujo de vídeo.

4.1 Diseño del controlador

Para este tipo de problemas hay diversas soluciones. Nosotros vamos a escoger una en la que supondremos que la posición del objetivo a seguir está definida (este problema será estimado en el Capítulo 5), ya que, además de que saber la posición del objetivo nos facilita el control de la plataforma, también es útil en general para el seguimiento de objetivos desde un UAV.

En [9], se diseña un algoritmo basado en cuaternios para controlar la orientación de un quadrotor, un dron con cuatro rotores. Nosotros vamos a adaptarlo a nuestra plataforma. Definimos los siguientes parámetros:

$\vec{n}_x$ = Vector normalizado de origen

$\vec{n}_u$ = Vector normalizado de apuntamiento deseado

q_d = Cuaternio que define la rotación más corta entre los vectores

Recordando la ecuación (2.23), q_d es definido como:

$$q_d = e^{\frac{1}{2}\theta_d \vec{u}_d} = \cos(\theta_d/2) + \vec{u}_d \sin(\theta_d/2) \quad (4.1)$$

Donde θ_d y $\overline{u}_d$ son, respectivamente, el ángulo y el eje de la rotación más corta.

El producto vectorial entre ambos vectores es definido como:

$$\vec{n}_u \times \vec{n}_x = \vec{u}_d \sin(\theta_d) \quad (4.2)$$

Mientras que el producto escalar es:

$$\vec{n}_u \cdot \vec{n}_x = cos(\theta_d) \quad (4.3)$$

Nos ayudaremos de algunas igualdades trigonométricas como:

$$\cos(\theta_d/2) = \pm\sqrt{\frac{1 + cos(\theta_d)}{2}} \; ; \; \sin(\theta_d/2) = \pm\sqrt{\frac{1 - cos(\theta_d)}{2}} \quad (4.4)$$

Entonces, el cuaternio que alinea el vector $\vec{n}_x$ con el vector $\vec{n}_u$ se define como:

$$q_t = \pm\sqrt{\frac{1 + \vec{n}_u \cdot \vec{n}_x}{2}} + \frac{\vec{n}_u \times \vec{n}_x}{\|\vec{n}_u \times \vec{n}_x\|}\sqrt{\frac{1 - \vec{n}_u \cdot \vec{n}_x}{2}} \quad (4.5)$$

Se puede añadir una rotación adicional sobre el eje X (rotación en roll):

$$q_d = q_t \circ q_z \tag{4.6}$$

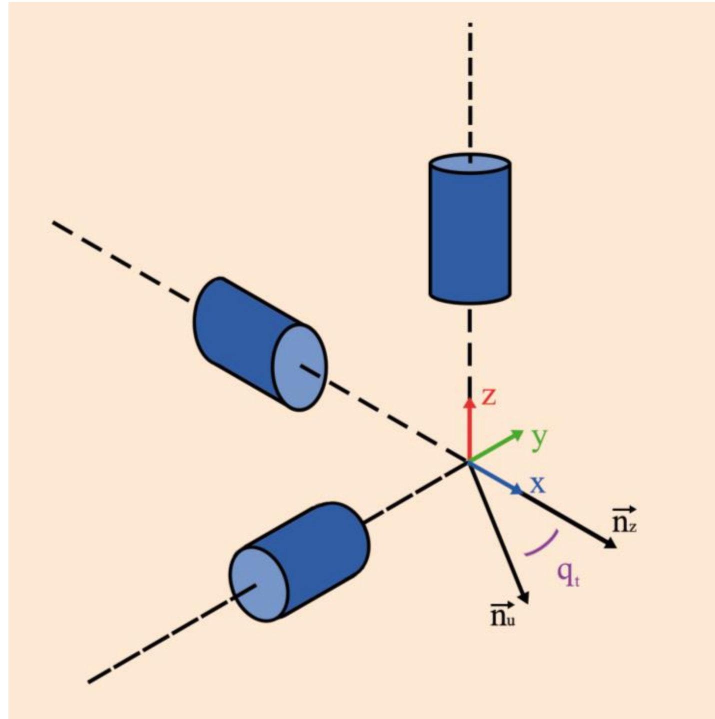

Figura 4-1. Representación del cuaternio q_t

Si denotamos q como el cuaternio que define la orientación de la plataforma, podemos considerar el cuaternio de error como:

$$q_e = {q_d}^* \circ q \tag{4.7}$$

Los ángulos de rotación de error se obtienen con las ecuaciones (4.24) y (4.25):

$$\theta_e = 2\ln\left(q_e\right) \tag{4.8}$$

El objetivo del controlador es que $q \rightarrow q_d$, lo que supone que $q_e \rightarrow (1,0,0,0)$, debido a que q y q_d son cuaternios unitarios, y su inversa es igual que su conjugado, y $q^{-1}q = (1,0,0,0) = 1 + 0i + 0j + 0k = 1$.

La velocidad angular deseada para conseguir la rotación deseada la obtendremos mediante el siguiente controlador:

$$w_e = -\gamma * \tanh\left(\frac{\left\|K_p \ln(q_e) + K_d * w\right\|}{\gamma}\right) * \frac{K_p \ln\left(q_e\right) + K_d * w}{\left\|K_p \ln\left(q_e\right) + K_d * w\right\|} \tag{4.9}$$

Siendo K_p y K_d constantes proporcionales y derivativas, γ un parámetro de saturación de la actuación y w la velocidad angular actual de la plataforma.

4.2 Simulación del controlador

Para validar el funcionamiento del controlador vamos a realizar una simulación en el entorno de Matlab-Simulink [18].

Hemos dividido el entorno de la simulación en tres partes:

- *Parámetros de entrada (Azul):* Aquí introducimos los vectores $\vec{n}_u$ y $\vec{n}_x$, así como las constantes de control K_p, K_d y γ.
- *Esquema de control (Verde):* Es donde calculamos el cuaternio de referencia, así como realizamos el cálculo de las velocidades angulares deseadas.
- *Obtención de las gráficas (Rojo):* Las variables se envían al Workspace de Matlab para procesar las gráficas deseadas. También se obtienen algunos resultados de interés, como el vector actual, que se obtiene mediante el giro del vector inicial según el cuaternio actual de la plataforma, o el producto vectorial del vector actual y el vector inicial.

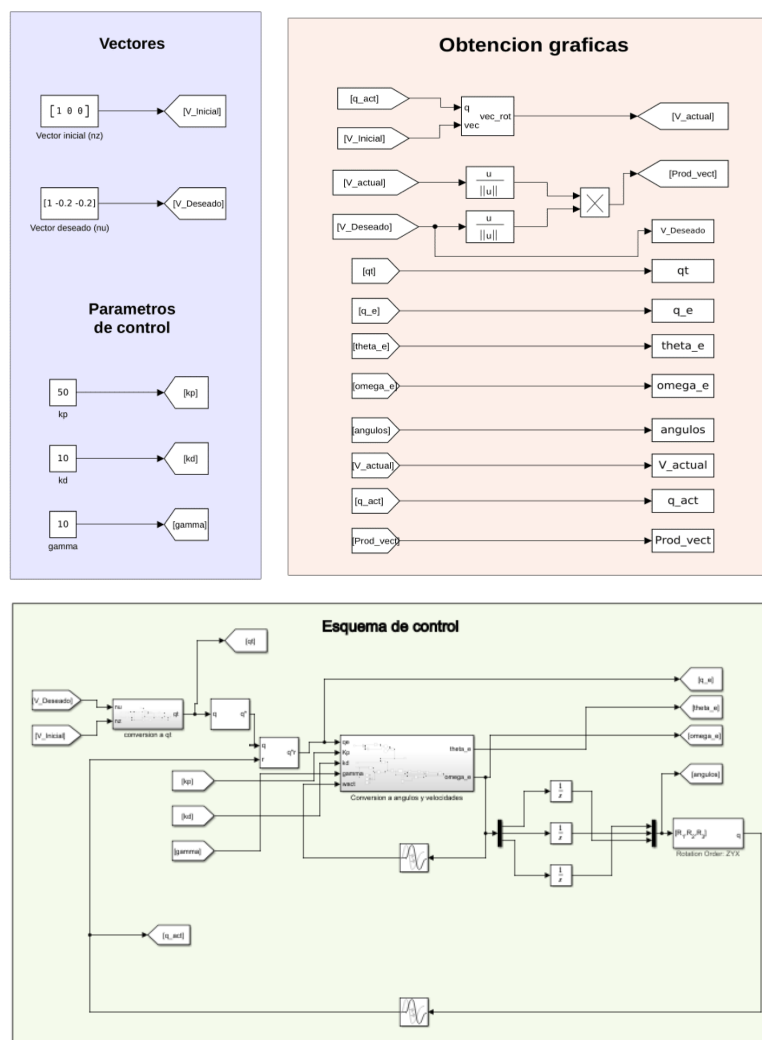

Figura 4-2. Esquema en Simulink de la simulación del controlador

Dentro del esquema de control, hemos dividido en varios subsistemas los diferentes cálculos.

Primero, tenemos la obtención de q_t:

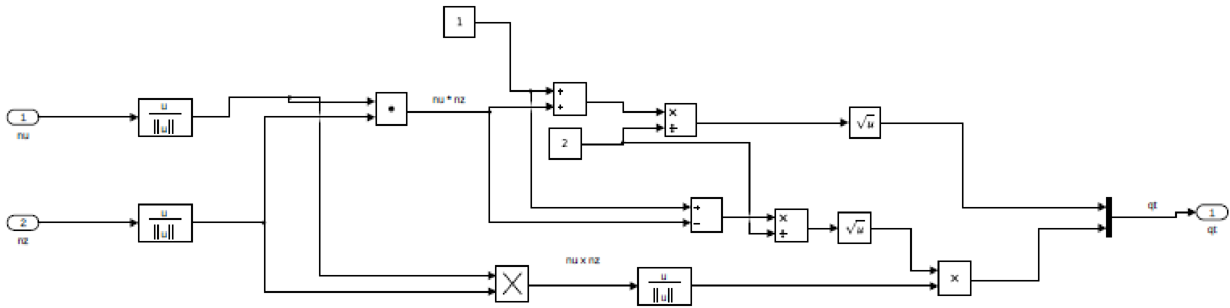

Figura 4-3. Obtención del cuaternio deseado en la simulación

En este subsistema, entran los vectores $\vec{n}_u$ y $\vec{n}_x$, con los que se calcula el cuaternio deseado, el cual se calcula su conjugado y se compara con el cuaternio actual de la plataforma, para proporcionar la señal del cuaternio de error, q_e.

Luego, este cuaternio de error se introduce en el subsistema del cálculo de las velocidades angulares:

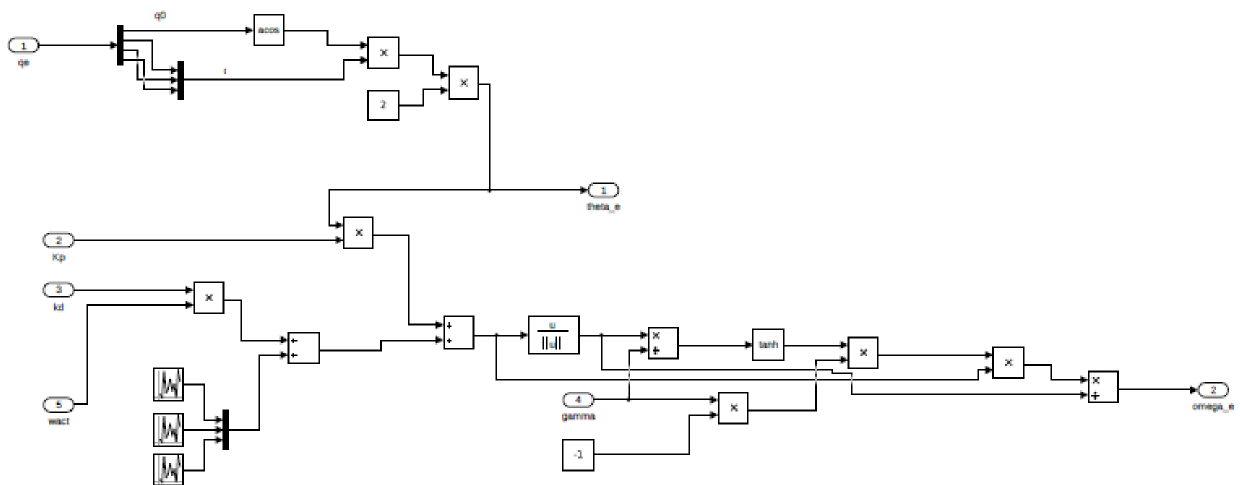

Figura 4-4. Esquema para el cálculo de las velocidades angulares deseadas

Para obtener en este esquema la velocidad actual, hemos utilizado la salida de este esquema, a la que le hemos añadido ruido uniformemente distribuido de valor ± 0.01 rad/s.

Además, para mejorar la fidelidad con la realidad de la simulación, hemos implementado un retraso de 0.2 segundos tanto a la velocidad actual que usamos en el controlador como al cuaternio de orientación actual.

Es digno de mención que, en muchos trabajos de este tipo en los que se controla una plataforma gimbal, el control se realiza con los voltajes de los motores de las articulaciones. Sin embargo, en este trabajo solo se obtienen unas velocidades deseadas. Esto es así porque el fabricante de la plataforma gimbal tiene una interfaz mediante la cual se puede controlar el dispositivo según la velocidad angular en cada eje que necesitemos. Incidiremos en este tema en el Capítulo 6, en el que nos ocuparemos de la integración de todos los componentes y dispositivos, tanto software como hardware, para formar el sistema completo, objeto de este proyecto.

La simulación que realizaremos tendrá los siguientes parámetros:

$\vec{n}_x = [1\ 0\ 0]$

$\vec{n}_u = [1 - 0.2 - 0.2]$

$K_p = 30$

$K_d = 0.15$

$\gamma = 12$

Los parámetros de control se han sintonizado mediante un método heurístico. Hemos aumentado la ganancia proporcional hasta que el sistema sobreoscila. Entonces, hemos usado el término derivativo para mejorar el transitorio.

A continuación, analizaremos los resultados:

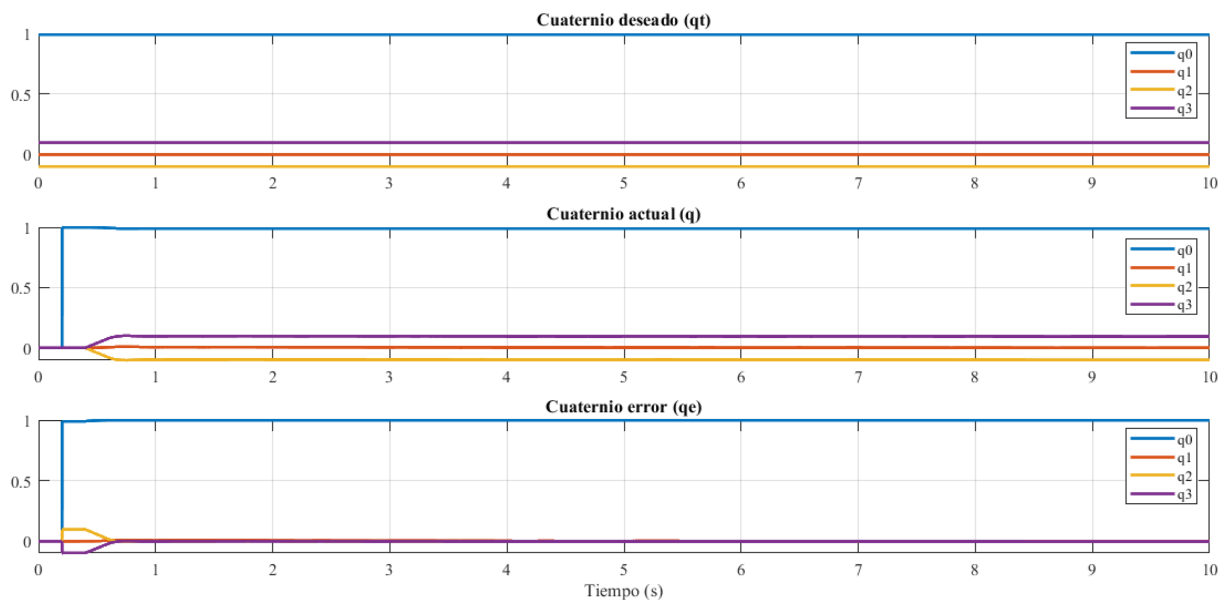

Figura 4-5. Representación de q_t, q_e, q en la simulación

En la Figura 4-4, podemos ver cómo el cuaternio de la plataforma converge hacia el cuaternio deseado, haciendo que el cuaternio de error se haga (1,0,0,0), lo que significa que se alcanza correctamente esa referencia.

Para ver si la orientación es, en efecto, correcta, vamos a analizar cómo evoluciona el vector de apuntamiento de la plataforma:

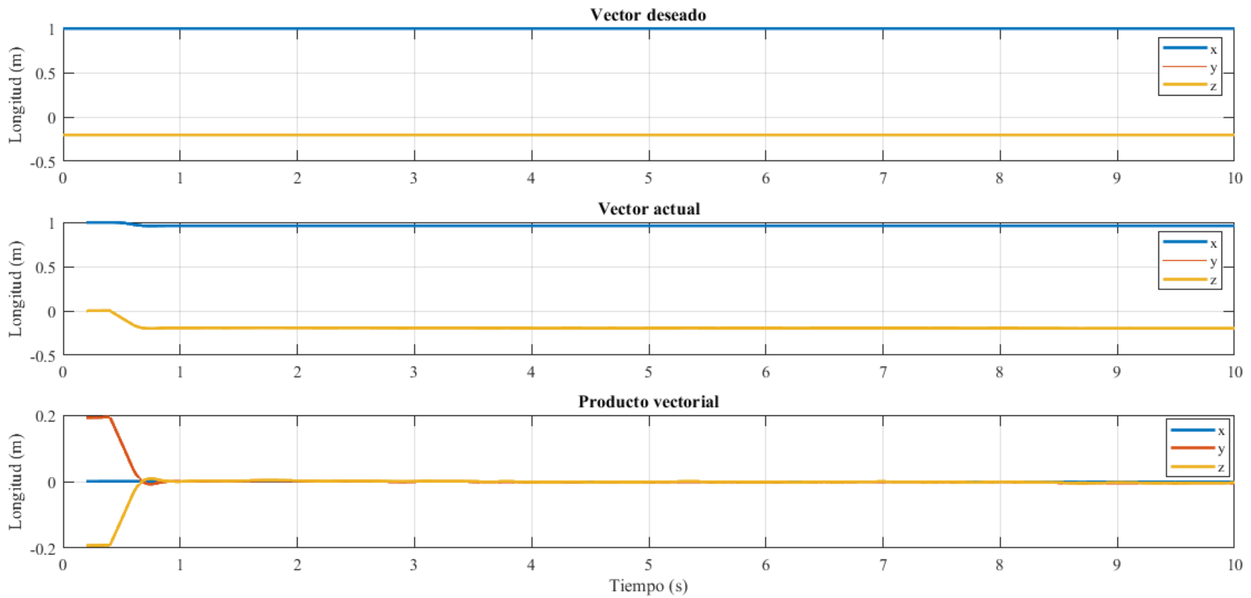

Figura 4-6. Representación del vector deseado, vector actual y el producto vectorial

Comprobamos que el vector dado por el giro del vector inicial según el cuaternio de la plataforma converge adecuadamente hacia el vector deseado, haciendo el producto vectorial entre estos tender a cero. Es sabido que el producto vectorial entre dos vectores es cero cuando éstos son paralelos entre sí.

Finalmente, vemos los resultados de las velocidades deseadas dadas por el controlador:

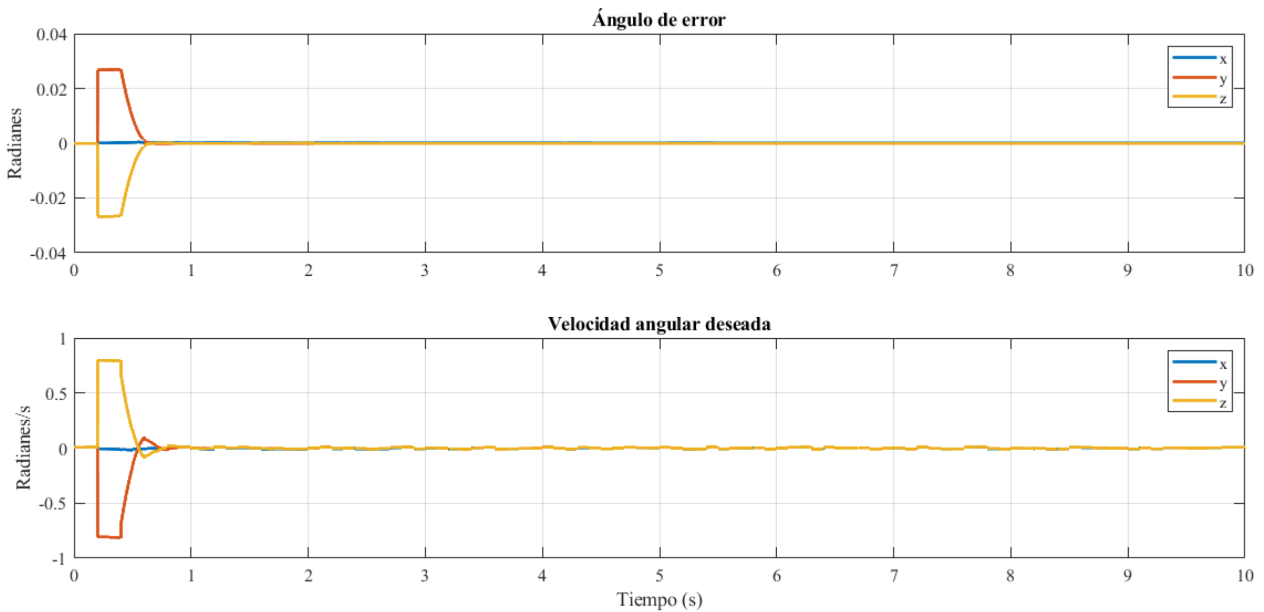

Figura 4-7. Representación del ángulo de error y la velocidad angular deseada

Tanto los ángulos de error como las velocidades tienden a cero, lo que confirma la estabilidad del sistema.

5 ESTIMACIÓN DE LA POSICIÓN DEL OBJETIVO

La estimación de la posición de un objetivo es uno de los problemas más recurrentes en el campo de la visión artificial. Conocer este dato es muy útil para un sinfín de aplicaciones, especialmente para el tracking de objetivos. El ser humano y otras especies animales han evolucionado su sentido de la vista de tal forma que somos capaces de estimar la distancia a la que están los objetos que estamos contemplando. Esto se consigue gracias a la visión estereoscópica, de la que hablaremos más adelante, la cual se basa en la obtención de 2 imágenes simultáneas desde diferentes puntos (los ojos).

5.1 Visión por computadora

La visión por computadora es una disciplina que tiene por objetivo hacer que un sistema automático entienda su entorno por medio de imágenes [19]. Las principales áreas de estudio de esta son:

- *Reconocimiento de patrones*
- *Reconstrucción tridimensional y fotogrametría*
- *Detección de objetos en movimiento (Tracking)*
- *Reconocimiento de objetos tridimensionales*

Antes de continuar, vamos a aclarar el significado de fotogrametría.

La fotogrametría es la ciencia de obtener información sobre las propiedades de las superficies y objetos sin contacto físico con estos, además de la medición e interpretación de esta información.

Esto puede usarse en diversos problemas como la topología, el seguimiento de objetivos a través de imágenes, estimación de la altura de UAVs o medición de estructuras.

El proceso mediante el cual a partir de una imagen se obtiene una medición, interpretación o decisión es llamado Análisis de Imágenes. Este consiste en cinco etapas [13]:

- *Adquisición de la imagen:* Se obtiene una imagen adecuada para su estudio.
- *Preprocesamiento:* Se emplean filtros digitales para mejorar la calidad de la imagen.
- *Segmentación:* Se identifica el objeto de estudio.
- *Medición:* Se obtienen las características deseadas del objeto.
- *Interpretación:* Según los valores obtenidos en la medición, se interpreta la escena.

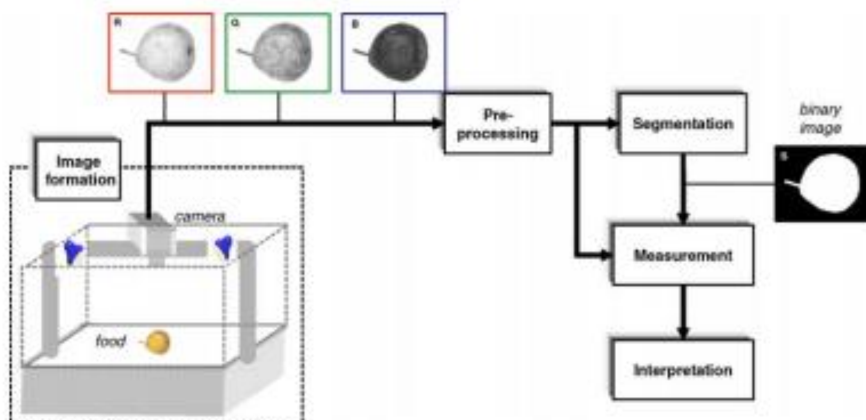

Figura 5-1. Etapas del proceso de análisis de imágenes [47]

La visión por computadora es una de las disciplinas más complejas de la automática y la robótica, especialmente si intentamos emular el sentido de la vista del ser humano y otros seres vivos. Los humanos, gracias a nuestra visión binocular, somos capaces incluso de medir el mundo en tres dimensiones (visión estereoscópica), segmentar los objetos gracias a la visión en color, y adaptarnos continuamente a los cambios de luz y de entorno.

5.1.1 OpenCV

OpenCV (Open Source Computer Vision Library) [20] es una librería de código abierto que incluye cientos de algoritmos dedicados a la visión por computadora. Tiene una estructura modular, lo que significa que incluye varias librerias estáticas o compartidas. Los siguientes módulos están disponibles [21]:

- *Core functionality (core):* Define estructuras de datos básicas, incluyendo la clase Mat, que puede ser usada para almacenar vectores y matrices reales y complejas, así como imágenes en escala de grises o color, nubes de puntos, tensores, histogramas…
- *Image Processing (imgproc):* Un módulo de procesamiento de imágenes que incluye filtrado, transformaciones geométricas, conversión de espacio de color, histogramas…
- *Video Analysis (video):* Un módulo de análisis de vídeo que incluye estimación del movimiento, algoritmos de seguimiento de objetos…
- *Camera Calibration and 3D reconstruction (calib3d):* Algoritmos geométricos con múltiples perspectivas, calibración simple y estéreo, estimación de la pose, reconstrucción 3D
- *2D Features Framework (features2d):* Detector de características, descriptores.
- *Object Detection (objdetect):* Detección de objetos predefinidos.
- *Vídeo I/O (videoio):* Interfaz para capturar y grabar vídeo.

Los módulos que se usarán en este trabajo son, principalmente, el de Vídeo I/O, para obtener y procesar el flujo de vídeo en tiempo real y el módulo Camera Calibration and 3D reconstruction, para calibrar la cámara y obtener soluciones para el problema de obtener la posición del objetivo mediante imágenes.

5.1.2 AprilTag

AprilTag [22] es un sistema fiducial visual, útil para un amplio rango de tareas como realidad aumentada, robótica y calibración de cámaras. Los marcadores pueden ser obtenidos desde una simple impresora, y el software de detección de AprilTag computa con precisión la orientación, posición e identidad de los marcadores con relación a la cámara. La librería AprilTag es implementada en C sin dependencias externas. Está diseñada para ser fácilmente incluida en otras aplicaciones, así como en dispositivos embarcados. Puede ser perfectamente usado en aplicaciones en tiempo real incluso con procesadores del tipo de los teléfonos móviles.

Hay ejemplos del uso de AprilTag en tracking de objetivos. Por ejemplo, en [23].

En esta figura mostramos un ejemplo del reconocimiento de marcadores AprilTag mediante un script en Python, usando una librería adecuada a este lenguaje de programación [24]. La información que se obtiene cuando se reconoce un marcador es la que está representada en la imagen: El centro del marcador (en rojo), los cuatro puntos del bounding box que contiene el marcador (en verde), el número de marcador según el orden en el que los ha reconocido el algoritmo (en amarillo) y el nombre de la familia del marcador (en ocre), que en este caso es “tag36h11”. Esta librería se puede usar con varias familias de diversos marcadores, en las cuales los tenemos hasta en forma circular. La información de los puntos obtenidos se obtiene en número de píxeles X e Y, según el sistema de coordenadas del plano imagen.

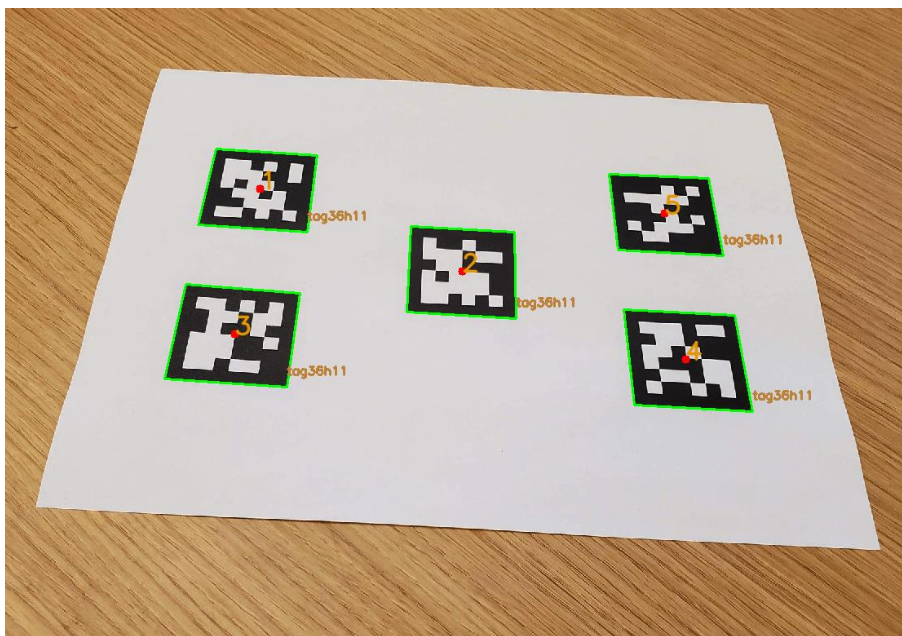

Figura 5-2. Ejemplo de reconocimiento de AprilTags en una imagen

5.2 Triangulación

Una cámara estereoscópica o cámara 3D, nombrada así debido a la visión estereoscópica humana (3D), es una cámara capaz de capturar imágenes (fotografías) en tres dimensiones. La visión binocular humana, produce dos imágenes (una para cada ojo) que luego se mezclan en el cerebro creando la imagen 3D. Estas cámaras, intentan imitar este comportamiento, utilizando dos objetivos (o dos cámaras separadas estratégicamente) captando la fotografía en el mismo instante, y como resultado se obtienen las imágenes 3D [25].

La visión estereoscópica está basada en la triangulación. El problema de la triangulación se define de la siguiente forma [26]: Siendo x un punto en $\mathbb{R}^3$ visible en dos imágenes, suponemos las matrices de dos cámaras P y P' (estas matrices fueron definidas en la ecuación 3.5) conocidas. Introducimos u y u' como las proyecciones del punto x en las dos imágenes. Con estos datos, podemos obtener los dos rayos en el espacio correspondientes a los puntos en la imagen. El problema de la triangulación es el de obtener la intersección de estos dos rayos.

Hay varios métodos para resolver este problema. Nosotros vamos a escoger el método lineal [26], ya que es el que está implementado en la librería OpenCV.

Suponemos $u = Px$, y escribimos u en coordenadas homogéneas $u = w(u, v, 1)^T$, donde (u, v) son las coordenadas del punto observado en la cámara y w un factor de escala desconocido. Denotando ${p_i}^T$ la fila i-ésima de la matriz P, podemos escribir la ecuación como:

$$wu = {p_1}^T x \;, \qquad wv = {p_2}^T x \;, \qquad w = {p_3}^T x \tag{5.1}$$

Eliminando w usando la tercera ecuación, llegamos a:

$$\begin{aligned} u{p_3}^T x &= {p_1}^T x \\ v{p_3}^T x &= {p_2}^T x \end{aligned} \tag{5.2}$$

Análogamente, podemos obtener las ecuaciones de la otra cámara, obteniendo así 4 ecuaciones lineales que pueden ser escritas de forma $Ax = 0$. Con la solución no nula de este sistema obtendremos las coordenadas del punto x.

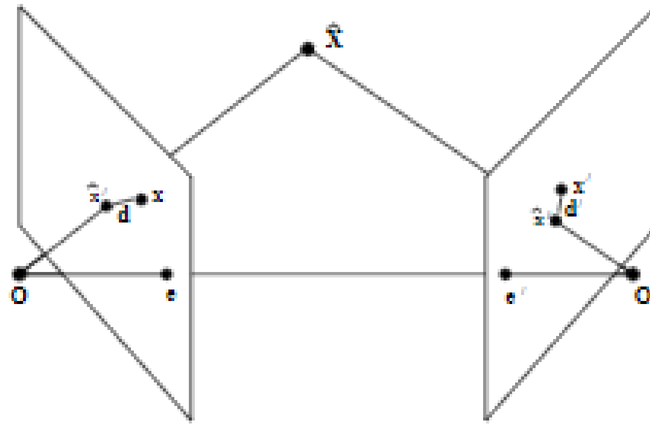

Figura 5-3. Triangulación con dos imágenes [16]

5.3 Calibración

Antes de comenzar con los procedimientos de obtención de la posición del punto objetivo, debemos obtener la matriz de calibración de la cámara, vista en el Capítulo 3. Para ello, se ha de realizar un algoritmo de calibración.

La matriz de calibración K proporciona la transformación entre un punto en la imagen y un rayo en el espacio euclídeo en 3 dimensiones [16]. Una cámara calibrada es como un sensor de dirección, capaz de medir la dirección de los rayos.

Rescatando la ecuación 3.5, $P = K[R|t]$, la matriz P expresa la correspondencia entre el punto de la imagen (x) y el punto en el espacio (X), lo que se expresa como $x = PX$.

El algoritmo de calibración tiene dos partes [16]:

- *Parte 1:* Calculamos la matriz P a través de correspondencias entre un grupo de puntos
- *Parte 2:* Descomponemos P en K, R y t a través de factorización QR

Para realizar la calibración, tendremos que realizar varias fotografías de un patrón predefinido.

Cada correspondencia $x_i = PX_i$ genera dos ecuaciones:

$$u_i = \frac{p_{11}X_i + p_{12}Y_i + p_{13}Z_i + p_{14}}{p_{31}X_i + p_{32}Y_i + p_{33}Z_i + p_{34}} \qquad v_i = \frac{p_{21}X_i + p_{22}Y_i + p_{23}Z_i + p_{24}}{p_{31}X_i + p_{32}Y_i + p_{33}Z_i + p_{34}} \tag{5.3}$$

Multiplicando, obtenemos las siguientes ecuaciones lineales:

$$\begin{gathered} u_i(p_{31}X_i + p_{32}Y_i + p_{33}Z_i + p_{34}) = p_{11}X_i + p_{12}Y_i + p_{13}Z_i + p_{14} \\ v_i(p_{31}X_i + p_{32}Y_i + p_{33}Z_i + p_{34}) = p_{21}X_i + p_{22}Y_i + p_{23}Z_i + p_{24} \end{gathered} \tag{5.4}$$

Que se pueden reorganizar como:

$$\begin{pmatrix} X & Y & Z & 1 & 0 & 0 & 0 & 0 & -uX & -uY & -uZ & -u \\ 0 & 0 & 0 & 0 & X & Y & Z & 1 & -vX & -vY & -vZ & -v \end{pmatrix} p = 0 \tag{5.5}$$

Con $p = (p_{11}, p_{12}, p_{13}, p_{14}, p_{21}, p_{22}, p_{23}, p_{24}, p_{31}, p_{32}, p_{33}, p_{34})^T$ como un vector de dimension 12.

Concatenamos las ecuaciones de $n \geq 6$ correspondencias para generar $2n$ ecuaciones, que pueden ser escritas como $Ap = 0$, donde A es una matriz $2n\ x\ 12$.

En general, esto no tendrá una solución exacta, pero sí una solución lineal que minimiza $|Ap|$ sujeto a $|p| = 1$, obtenida del autovector con menor autovalor de $A^T A$.

Una vez obtenida P, M es la primera submatriz 3x3, la cual es el producto ($M = KR$) de una matriz triangular superior y una matriz de rotación.

Se factoriza M en KR usando factorización QR. Entonces,

$$t = K^{-1}(p_{14}, p_{24}, p_{34})^T \tag{5.6}$$

Finalmente, obtenemos la ecuación de calibración de la cámara, que ya habíamos definido en la ecuación 3.3.

$$K = \begin{bmatrix} \alpha_x & s & u_o \\ 0 & \alpha_y & v_o \\ 0 & 0 & 1 \end{bmatrix} \tag{5.7}$$

Para resolver el problema de la calibración, vamos a ayudarnos de los algoritmos implementados en la librería de OpenCV [27], mediante el cual también estudiaremos la distorsión tangencial y radial de la cámara, conceptos introducidos en el Apartado 3.2.

La distorsión tangencial se puede expresar como:

$$u_{dist} = u + 2p_1uv + p_2(r^2 + 2u^2) \tag{5.8}$$

$$v_{dist} = v + p_1(r^2 + 2v^2) + 2p_2uv$$

Mientras que la distorsión radial quedaría como:

$$u_{dist} = u(1 + k_1r^2 + k_2r^4 + k_3r^6) \tag{5.9}$$

$$v_{dist} = v(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

Siendo r la distancia euclídea entre el punto distorsionado de la imagen y el centro, y $(k_1, k_2, k_3, p_1, p_2)$ los coeficientes de distorsión.

Para el cálculo de la calibración, usaremos imágenes de un patrón predefinido, en este caso un tablero de ajedrez. En el tablero, encontraremos puntos cuya posición relativa sea conocida, como, por ejemplo, las esquinas de las casillas. Conocemos las coordenadas de estos puntos en coordenadas globales y en las coordenadas de la imagen, por lo que podemos resolver el problema de la calibración. Para obtener mejores resultados, es recomendable usar al menos 10 fotografías del patrón.

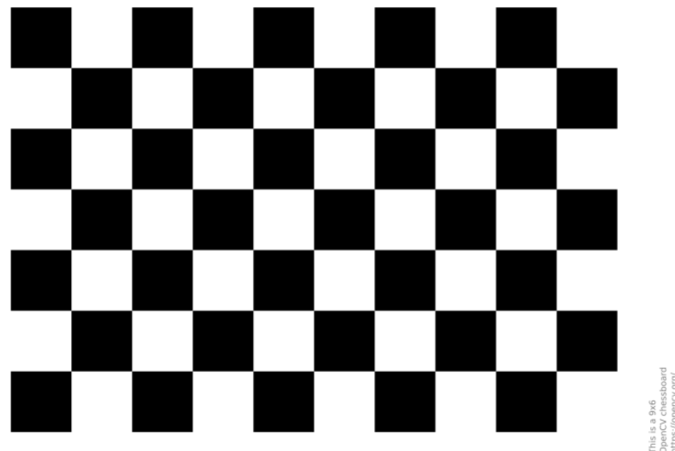

Figura 5-4. Patrón usado para la calibración. Tablero de ajedrez 10x7

Usando el código proporcionado en [27], particularizado para nuestro patrón, obtenemos los siguientes resultados:

$$Matriz\ de\ calibración\ (K) = \begin{bmatrix} 1637.04 & 0 & 949.36 \\ 0 & 1638.64 & 532.63 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.10)$$

$$Dist = [0.017377 \quad -0.365425 \quad 0.013127 \quad -0.002851 \quad 0.802158] \quad (5.11)$$

Figura 5-5. Ejemplo de las imágenes tomadas para la calibración

5.4 Obtención de la posición del objetivo. Primer método.

El primer método que vamos a desarrollar es el de la triangulación, explicado teóricamente en el Apartado 5.2. Este método es el más preciso, pero a su vez es el más complicado de implementar, especialmente a nivel del hardware necesario, ya que necesitaríamos más de una cámara. También sería imprescindible conocer siempre la orientación y distancia relativas entre las dos cámaras en todo momento, lo que requeriría usar sistemas GPS, además de que necesitaríamos varios UAV, y para desarrollar este trabajo no vamos a poder disponer de toda esta arquitectura.

Sin embargo, para comprobar la validez de esta solución de cara a futuros trabajos y proyectos en los que se disponga del material necesario, vamos a llevar a cabo un experimento sencillo, en el que tomaremos fotos del objetivo desde diferentes puntos en el espacio conocidos, con orientaciones de la cámara también definidas, para así realizar la triangulación entre las imágenes. Es evidente que esto es una limitación de cara a probar el algoritmo, ya que el mero hecho de ejecutar el programa en tiempo real incluirá ruidos de medida introducidos, por ejemplo, por el movimiento del objeto, o por fallos en la detección, que seguramente nos lleven a necesitar filtrar los resultados.

En nuestra simulación, vamos a suponer que disponemos de 3 cámaras, de forma que podamos hacer triangulaciones con cada par de dispositivos, de forma que se reduzca y compense el posible error de medición. Así se resuelve en [28], donde se realiza una triangulación con las medidas de sensores de radiofrecuencia portados por tres UAV. De esta triangulación por pares se obtienen 3 medidas de la posición, las cuales se usan para definir un triángulo en el espacio, cuyo centroide será tomado como la posición final del objetivo.

Antes de continuar, vamos a definir el eje de coordenadas de la cámara:

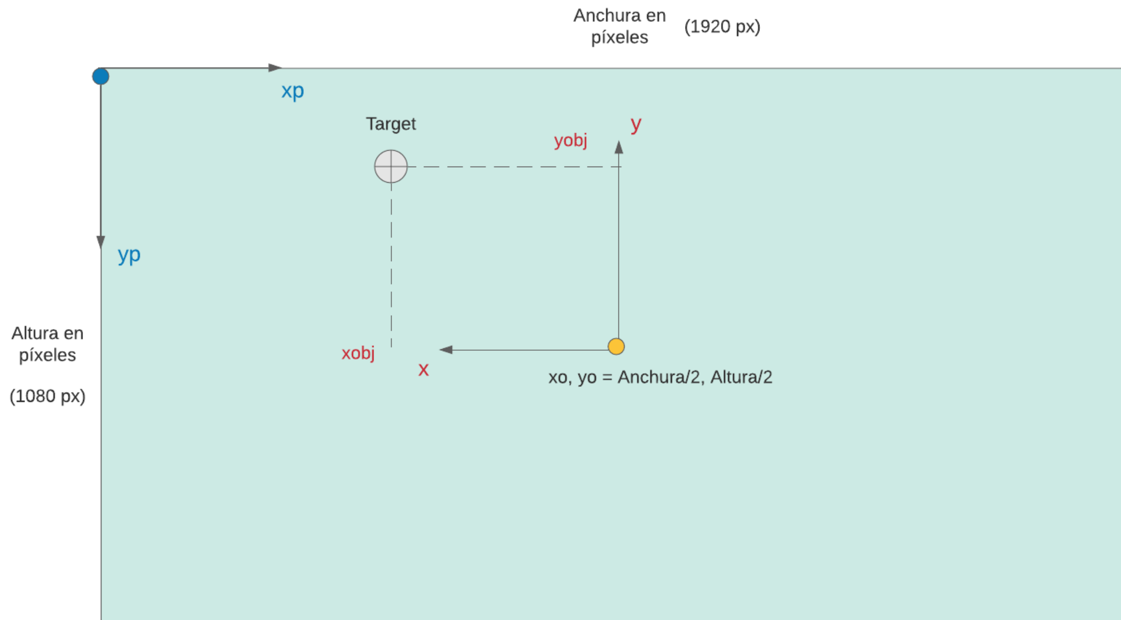

Figura 5-6. Eje de coordenadas de la cámara y de la imagen

El eje z se sitúa respetando un sistema dextrógiro con x, y. Los ejes xp, yp identifican las coordenadas del plano imagen, y se miden en píxeles. A su vez, x_o e y_o definen el centro de la imagen.

La altura y anchura en píxeles están definidas por la resolución de la cámara.

Recordamos que para computar la triangulación, necesitamos la matriz P de la cámara. Esta matriz, como se introdujo en la ecuación 3.5, se define como P= $K[R|t]$, donde K es la matriz de calibración de la cámara, obtenida en el Apartado 5.3, y $[R|t]$, es la matriz conocida como matriz de parámetros extrínsecos, que determina la orientación y traslación de la cámara con respecto al origen.

A continuación, vamos a definir en la siguiente tabla las traslaciones y rotaciones que hay entre las imágenes que hemos tomado para realizar la triangulación. Los ángulos están expresados en Ángulos de Euler (Roll, Pitch, Yaw).

Tabla 5–1. Posición relativa de la cámara en la toma de imágenes

Traslación posición 2-1 (metros)	(-0.68, 0, 0.62)
Rotación posición 2-1 (grados)	(0, 90, 0)
Traslación posición 3-1 (metros)	(-0.50, 0, 0)
Rotación posición 3-1 (grados)	(0, 45, 0)
Traslación posición 3-2 (metros)	(0.62, 0, 0.18)
Rotación posición 3-2 (grados)	(0, -45, 0)

Para calcular la matriz R en cada una de las imágenes, nos basamos en las matrices de rotación para cada uno de los Ángulos de Euler, ecuaciones 2.2, 2.3, 2.4. Recordamos que la imagen 1 la hemos tomado como el origen de coordenadas.

Las matrices de parámetros extrínsecos en cada una de las imágenes sería la siguiente:

$$Extrinsic_{1-0} = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \tag{5.12}$$

$$Extrinsic_{2-1} = \left[\begin{array}{ccc|c} \cos(90) & 0 & sin(90) & -0.68 \\ 0 & 1 & 0 & 0 \\ -\sin(90) & 0 & \cos(90) & 0.62 \\ 0 & 0 & 0 & 1 \end{array}\right] \tag{5.13}$$

$$Extrinsic_{3-1} = \left[\begin{array}{ccc|c} \cos(45) & 0 & sin(45) & -0.5 \\ 0 & 1 & 0 & 0 \\ -\sin(45) & 0 & \cos(45) & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \tag{5.14}$$

$$Extrinsic_{3-2} = \left[\begin{array}{ccc|c} \cos(-45) & 0 & sin(-45) & 0.62 \\ 0 & 1 & 0 & 0 \\ -\sin(-45) & 0 & \cos(-45) & 0.18 \\ 0 & 0 & 0 & 1 \end{array}\right] \tag{5.15}$$

Con esto, ya tenemos todos los datos necesarios para realizar la triangulación. Para ello, nos vamos a ayudar de la función de OpenCV “`triangulatePoints`” [29]. Esta función utiliza internamente el método de triangulación lineal explicado en el Apartado 5.2.

La función recibe como parámetros de entrada un array con los puntos del centro del objeto medidos en píxeles, en cada una de las dos imágenes tomadas, así como las matrices de proyección P de cada imagen. La salida es, como es de esperar, un punto en tres dimensiones que define la posición del punto objetivo en el espacio.

El código completo del programa se puede observar en el Anexo A.

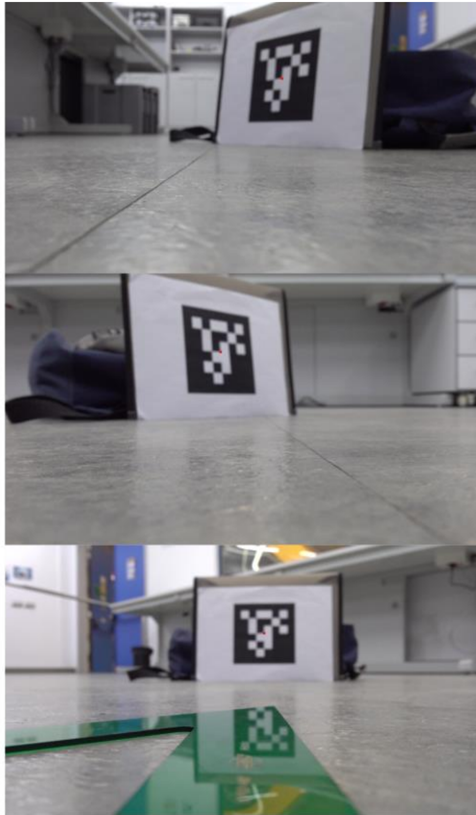

Figura 5-8. Imágenes usadas para el cálculo de la triangulación

En la siguiente tabla, mostramos los resultados obtenidos con cada una de las triangulaciones, así como la posición real del punto en tres dimensiones.

Tabla 5–2. Resultados de la triangulación (metros)

Punto 1-2	(0.059, -0.130, 0.643)
Punto 1-3	(0.063, -0.114, 0.670)
Punto 2-3	(0.098, -0.149, 0.667)
Punto real	(0.060, -0.110, 0.670)
Punto calculado (centroide)	(0.073, -0.131, 0.660)

El error lo determinamos como el módulo del vector que une el punto calculado y el punto real, que se trata de 0.0193 metros, es decir, en torno a 2 centímetros, lo que supone un error relativo con respecto al módulo del vector de posición real del 2.85%.

A continuación, mostramos el resultado representado en el espacio.

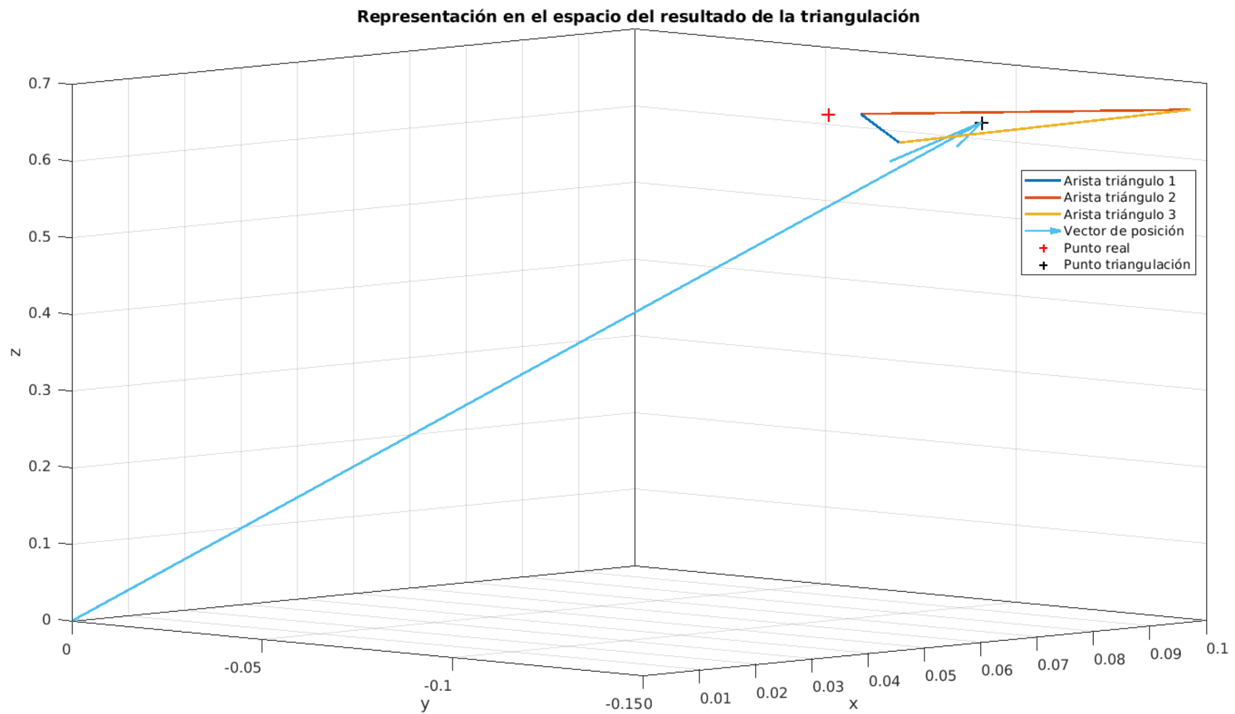

Figura 5-9. Representación en el espacio del resultado de la triangulación

5.5 Obtención de la posición del objetivo. Segundo método.

En este apartado, vamos a presentar un método para obtener la posición que sea implementable en tiempo real con el equipo disponible en este trabajo. Para ello, nos vamos a basar en el modelo Pinhole, introducido en el Capítulo 3.

Considerando que R y t expresan rotación y traslación nulas:

$$Z\begin{bmatrix}u\\v\\1\end{bmatrix}=\begin{bmatrix}\alpha_x & 0 & u_o & 0\\0 & \alpha_y & v_o & 0\\0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix} \tag{5.16}$$

Esta expresión no es más que la ecuación 3.1, incluyendo la matirz de calibración.

De aquí, inferimos que:

$$X=\frac{Z*(u-u_o)}{\alpha_x} \tag{5.17}$$

$$Y=\frac{Z*(v-v_o)}{\alpha_y} \tag{5.18}$$

Comprobamos que lo necesario para obtener las posiciones X e Y en el espacio del punto objetivo son los parámetros de calibración de la cámara (ya obtenidos) y la distancia Z hasta el objeto. Dado que este algoritmo está pensado para funcionar en un UAV a una altura determinada, rastreando objetivos a nivel del suelo, se puede considerar Z constante y conocida, en una simplificación realizada para poder implementar el algoritmo previamente sin necesidad de conseguir una triangulación que no necesitaría conocer la distancia al objetivo, sino que sería calculada.

Entonces, tenemos una constante directamente proporcional a la distancia Z e inversamente proporcional al parámetro de distancia focal expresado en píxeles.

$$Distance\ parameter\ X = Z * 0.0006108 \tag{5.19}$$

$$Distance\ parameter\ Y = Z * 0.0006102 \tag{5.20}$$

Multiplicando este parámetro por el número de píxeles que distan del centro horizontal y verticalmente, obtenemos la posición X e Y relativas a los ejes de la cámara.

Recuperando la Figura 5-6, podemos obtener la ecuación según los ejes ilustrados en la imagen:

$$x = Z * 0.0006108 * (x_o - x_{p-obj}) \tag{5.21}$$

$$y = Z * 0.0006102 * (y_o - y_{p-obj}) \tag{5.22}$$

Las ecuaciones 5.21 y 5.22 son en las que nos vamos a basar a la hora de crear nuestro programa.

5.5.1 Filtro de Kalman Adaptativo

El Filtro de Kalman es ampliamente usado en visión artificial. La dificultad de aplicar este filtro a este tipo de sistemas son las no linealidades. La aproximación más común es la de linealizar el sistema. En [30], se propone un Filtro de Kalman Adaptativo (AKF) para un sistema de control servovisual mediante una cámara estereoscópica solidaria a un robot. Otras aproximaciones del Filtro de Kalman Adaptativo consisten en modificar los parámetros estadísticos del mismo, mientras que el analizado en este trabajo utiliza un modelo de estado del sistema que se modifica a lo largo del tiempo, es decir, un modelo adaptativo del sistema. Esto es especialmente útil en este tipo de situaciones, en las que el conocimiento del movimiento del objetivo en el espacio es nulo a priori.

5.5.1.1 Filtro de Kalman general

Describimos un proceso lineal con la representación en espacio de estados. En primer lugar, tenemos la ecuación del proceso [30]:

$$X_{k+1} = AX_k + Bu_k + Dw_k \tag{5.23}$$

Donde $X_k \in \mathbb{R}^n$ es el vector de estados. El modelo del estado es caracterizado por la matriz de transición de estado $A \in \mathbb{R}^{n*n}$, que expresa la evolución del sistema en el tiempo. $B \in \mathbb{R}^{n*r}$ es la matriz de entradas, $D \in \mathbb{R}^{n*s}$ es la matriz de ruido, expresado como $w_k \in \mathbb{R}^s$. k denota el índice discreto de tiempo. En el proceso pueden aplicarse entradas $u_k \in \mathbb{R}^r$.

La ecuación de medición es expresada como:

$$Y_k = CX_k + v_k \tag{5.24}$$

Donde $C \in \mathbb{R}^{p*n}$ es conocida como la matriz de observabilidad, $v_k \in \mathbb{R}^p$ es el ruido de medición y, finalmente, $Y_k \in \mathbb{R}^p$ es la salida medible del sistema.

Los vectores de ruido w_k y v_k son estadísticamente independientes, lo que conlleva a que $E\left[w_i v_j^{\,T}\right] = 0$ para todo i, j, donde $E[.]$ denota la esperanza matemática. También asumimos la incorrelación del vector de estados con el ruido, por lo que $E\left[x_i v_j^{\,T}\right] = 0$ y $E\left[w_i x_j^{\,T}\right] = 0$ para todo i, j.

Si el proceso no es lineal, el modelo debe ser linealizado en torno a su estimación actual. Esto se conoce como Filtro de Kalman Extendido (EKF).

El Filtro de Kalman (KF) corrige iterativamente la ganancia de Kalman $K \in \mathbb{R}^{n*p}$ para hacer que la estimación del vector de estados tienda a una solución óptima. Esto se define con las siguientes ecuaciones:

$$X_{k+1|k} = AX_{k|k} + Bu_k \tag{5.25}$$

$$P_{k+1|k} = AP_{k|k}A^T + DQD^T \tag{5.26}$$

$$K_{k+1} = P_{k+1|k}C^T \left(CP_{k+1|k}C^T + R\right)^{-1} \tag{5.27}$$

$$X_{k+1|k+1} = X_{k+1|k} + K_{k+1}\left(Y_{k+1} - CX_{k+1|k}\right) \tag{5.28}$$

$$P_{k+1|k+1} = \left(I - K_{k+1}C\right) P_{k+1|k} \tag{5.29}$$

El KF controla el modelo del estado calculado la matriz de ganancia variable en el tiempo K_{k+1}. La estimación es óptima si w_k y v_k son ruidos blancos de media cero con distribución Gaussiana. Estos son definidos respectivamente por sus matrices de covarianza $Q_k \in \mathbb{R}^{s*s}$ y $R_k \in \mathbb{R}^{p*p}$ de forma que $E[w_k w_i^{\,T}] = Q_k \delta_{ki}$ y $E[v_k v_i^{\,T}] = R_k \delta_{ki}$, siendo δ_{ki} el operador de Kronecker. Si estas afirmaciones se respetan, el filtro da una estimación óptima de las variables de estado.

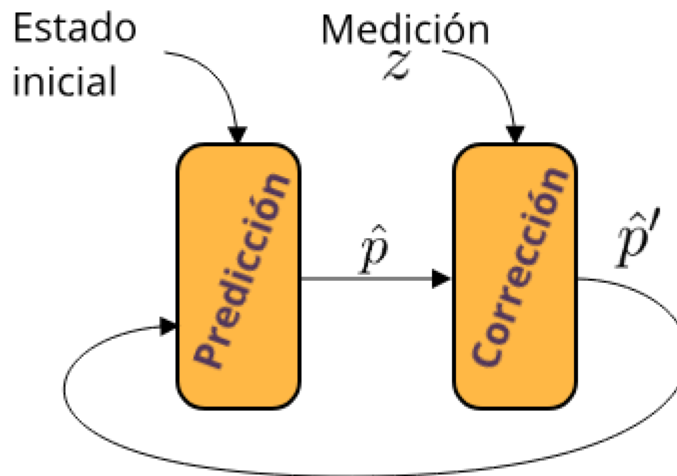

Figura 5-10. Iteración en Filtro de Kalman [50]

5.5.1.2 Implementación AKF

El filtro adaptativo se va a conseguir mediante una matriz de transición A dependiente del tiempo. Consideramos que las entradas vistas en la ecuación 5.25 son parte de las perturbaciones del sistema. Para calcular la matriz A, usaremos los dos últimos estados: $\hat{A}_k = X_k X^+{}_{k-1}$. La inversa $X^+{}_{k-1}$ será calculada usando una aproximación mediante mínimos cuadrados. Esto nos proporciona, para una iteración k, la matriz de transición estimada:

$$\hat{A}_k = X_k (X^T{}_{k-1} X_{k-1})^{-1} X^T{}_{k-1} \tag{5.30}$$

En cada iteración k, se calcula la matriz $\hat{A}_k$, para luego continuar con las ecuaciones 5.25, 5.26, 5.27, 5.28, 5.29.

En nuestro problema, hemos definido el vector de estados como:

$$X_k = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{5.31}$$

Siendo x, y, z las posiciones del objeto seguido en el espacio.

La matriz de observabilidad C, como:

$$C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.32}$$

Considerando que tenemos en cada iteración las mediciones de x, y, z.

Las matrices Q, R:

$$Q = R = \begin{bmatrix} 0.005 & 0 & 0 \\ 0 & 0.005 & 0 \\ 0 & 0 & 0.005 \end{bmatrix} \tag{5.33}$$

Elegidas mediante experimentación.

El AKF nos proporcionará una estimación óptima de la posición del objetivo, considerando un modelo en espacio de estados con una matriz de transición variable en el tiempo, según los estados anteriores.

Los resultados del Filtro de Kalman Adaptativo podrán revisarse en el Capítulo 7, “Resultados”, donde se analizarán todos los resultados del sistema completo, incluido el comportamiento del AKF en el algoritmo de tracking.

6 INTEGRACIÓN DEL SISTEMA

Tras presentar en los capítulos anteriores los modelos y algoritmos que vamos a usar para completar con éxito el obetivo del trabajo, ahora debemos integrar todos ellos y garantizar que hay una comunicación completa entre los diferentes periféricos que vamos a usar: la cámara, la plataforma gimbal, y un ordenador donde se ejecutarán los diferentes programas. En este capítulo, introduciremos las herramientas que vamos a usar para garantizar esta correcta integración y comunicación, y explicaremos pormenorizadamente los diferentes programas para cada parte del algoritmo, de forma que al final del apartado quede claramente explicado el funcionamiento total de la parte práctica del proyecto.

6.1 Comunicación con la plataforma. MAVLink

Para abordar el problema de la comunicación con el gimbal, usaremos el protocolo MAVLink, ya que es el protocolo de comunicación soportado por la plataforma Pixy U de Gremsy.

MAVLink [31] es un protocolo de mensajería para la comunicación con drones y sus componentes de a bordo más comunes. Sigue un diseño híbrido entre conexión mediante publicación-subscripción y protocolo punto a punto (PPP). Los datos son enviados como tópicos, mientras los subprotocolos de configuración como el protocolo de misión o de parámetros, usan PPP. Los mensajes son definidos dentro de ficheros XML. Cada fichero define el mensaje soportado por un sistema MAVLink particular, también llamado dialecto. Los mensajes de referencia que se usan en la mayoría de estaciones de tierra y autopilotos se definen en common.xml.

MAVLink es muy eficiente. Su versión 2.0. (la usada en este trabajo), solo usa 14 bytes de overhead por mensaje (el overhead es la información necesaria que es indispensable introducir en un protocolo de mensajería para el envío del mismo).

Es confiable, ha sido usado desde 2009 para la comunicación entre diferentes vehículos, estaciones de tierra y otros nodos, sobre variados y desafiantes canales de comunicación (latencia alta, ruido…). Provee métodos para detectar la pérdida y corrupción de paquetes, así como su autentificación.

Acepta 255 sistemas concurrentes en la red. Permite comunicación “offboard” y “onboard”, por ejemplo, entre la estación de tierra y un dron (offboard) y entre el autopiloto de un dron y una cámara y gimbal que usen MAVLink.

Puede ser usado en múltiples sistemas operativos y microcontroladores, como ARM7, ATMega, dsPic, STM32 and Windows, Linux, MacOS, Android, IOs…

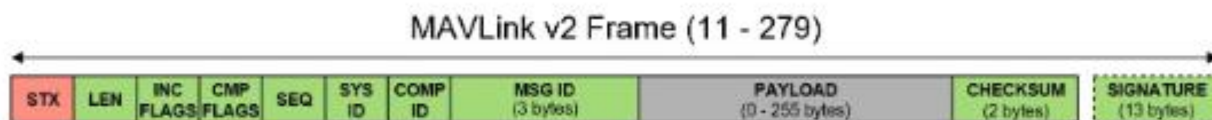

Figura 6-1. Estructura de un mensaje MAVLink [32]

Los campos más importantes del mensaje contienen los siguientes apartados [32] :

- *STX:* Describe el inicio del mensaje y siempre será 0xFD.
- *LEN:* Es el valor de la longitud del PAYLOAD.
- *SEQ:* Contiene la secuencia del paquete. El primer mensaje es representado con un 0.
- *SYS:* Representa el ID del sistema autónomo.

- *COMP:* Representa el componente del sistema que está enviando el mensaje.
- *MSG*: Define el tipo de mensaje.
- *Payload*: Contiene los datos reales del mensaje, dependientes del tipo de este.
- *Checksum*: Suma de comprobación. Garantiza la integridad del mensaje.

Para conseguir la comunicación con la plataforma vía MAVLink, el fabricante provee una SDK [33] (kit de desarrollo de software) para facilitar el diseño de soluciones para el control del hardware con este protocolo de mensajería.

Está desarrollada para sistemas Unix, garantizando la compatibilidad con nuestro SO (en este caso usamos Ubuntu 18.04).

Los mensajes MAVLink que se usan en esta SDK son:

- *HEARTBEAT (ID #0):* Mensaje de comprobación para garantizar la correcta conexión entre los dispositivos.

- *SYS_STATUS (ID #1):* Información de estado del gimbal. Estado de los motores, sensores, modos de operación.

- *RAW_IMU (ID: #27):* Envía los valores de la unidad de medida inercial. Aceleración en los 3 ejes y velocidad en los 3 ejes.

- *MOUNT_STATUS (ID: #158):* Provee los valores de los encoders de cada motor.

- *MOUNT_ORIENTATION (ID: #265):* Envía los valores de la orientación del gimbal, tanto con yaw relativo al vehículo como con yaw absoluto. Este mensaje es el que se debe usar como feedback para el control en velocidad o posición.

- *COMMAND_LONG (ID: #76):* Envía un mensaje al dispositivo. Los comandos que se usan en esta SDK son:
  - *MAV_CMD_DO_MOUNT_CONFIGURE:* Este comando selecciona el modo de funcionamiento en el que queremos que se encuentre el gimbal. Pueden ser, para cada ángulo; control en posición relativa al UAV, control en velocidad, control en posición global.
  - *MAV_CMD_DO_MOUNT_CONTROL:* Este comando envía el ángulo o velocidad deseadas en cada eje, expresados en grados o grados/s.

- *COMMAND_ACK (ID: #77):* Garantiza que se ha enviado el COMMAND_LONG.

6.2 Robot Operating System (ROS)

Una vez solucionada la comunicación con la plataforma gimbal, ahora debemos garantizar el envío de datos por parte de la cámara y del procesamiento de las imágenes que vamos a obtener, así como otros procesos como la grabación de video o la sintonización de parámetros de control.

Para unificar el protocolo de comunicación entre todos estos programas y procesos, vamos a usar la herramienta ROS.

ROS [34] es un meta sistema operativo para robots. Proporciona los servicios comunes a un sistema operativo, incluido capa de abstracción de hardware, control de dispositivos de bajo nivel, implementación de funcionalidades usadas asiduamente, comunicación mediante mensajes entre procesos y gestión de paquetes.

Proporciona herramientas y librerías para obtener, compilar, escribir y ejecutar código a través de múltiples ordenadores.

ROS fue creado para fomentar el desarrollo de software colaborativo para robots, es decir, para integrar soluciones creadas por diferentes especialistas en diversos campos de la robótica. Por ejemplo, un laboratorio desarrolla algoritmos para mapear localizaciones internas, otros expertos usan los mapas para gestionar la navegación autónoma del robot y otro equipo desarrolla un programa para reconocer objetos determinados. ROS fue diseñado específicamente para que grupos como estos pudieran colaborar y complementar el trabajo de cada uno [35].

ROS computation graph level [36]*:* El computation graph es la red peer-to-peer de los procesos de ROS que están procesando datos a la vez. Los conceptos básicos del computation graph son nodos, Master, Parameter Server, messages, services, topics, y bags. Todos ellos suministran datos al graph de diferentes formas.

Los **nodos** son procesos que realizan alguna computación. ROS está diseñado para ser modular, por lo que un sistema robótico normalmente se compone de varios nodos. Por ejemplo, un nodo controla un LIDAR, otro nodo controla las ruedas motoras, otro nodo realiza la localización, otro nodo la planificación de trayectorias, otro nodo proporciona una vista gráfica del sistema… Todos los nodos tienen un graph resource name que los identifica para el resto del sistema. Por ejemplo, /hoyuko_node puede ser el nombre de un sensor láser Hoyuko.

El **ROS Master** proporciona el nombre y registro para el resto de los nodos del sistema ROS. Lleva a los publicadores y subscriptores a los topicos y servicios. El rol del Master es permitir a los nodos individuales encontrarse unos a otros. Una vez que se han localizado, se comunican entre ellos de forma peer-to-peer. Una red peer-to-peer es una red en la que todos o algunos aspectos funcionan sin clientes ni servidores fijos, sino una serie de nodos que se comportan como iguales entre sí [37]. El Master también proporciona el Parameter Server, que permite almacenar los datos en una localización central.

Mensajes: Los nodos se comunican entre ellos mediante mensajes. Un mensaje es una estructura de datos. Soportan los tipos estandar (entero, flotante, booleano etc). Pueden incluir estructuras anidadas y arrays.

Topicos: Los mensajes son transportados mediante un sistema publicar/subscribir. Un nodo envía un mensaje publicándolo en un tópico dado. El tópico es un nombre que es usado para identificar el contenedor del mensaje. Un nodo que está interesado en un dato específico se subscribirá al tópico apropiado. Puede haber varios publicadores y subscriptores de forma concurrente en un mismo tópico, y a su vez, un nodo puede publicar y subscribirse a varios tópicos diferentes. En general, los publicadores y subscriptores no son conscientes de la existencia de cada uno de ellos. La idea es desacoplar la producción de información de su consumo. Puede decirse que un tópico es un bus con un tipo de mensaje propio. Cada bus tiene un nombre, y cualquiera puede conectarse al bus para enviar o recibir mensajes, siempre y cuando sean del tipo correcto.

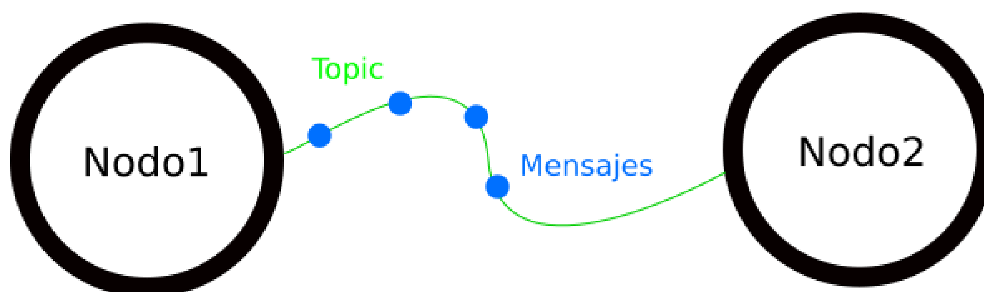

Figura 6-2. Comunicación entre dos nodos, por mensajes a través de un tópico [51]

Ejemplo: Para contolar un buscador láser Hoyuko, podemos crear un hoyuko_nodo, que publica la información del laser mediante un mensaje “sensor_msgs/LaserScan” en un tópico llamado scan. Para procesar esos datos, escribimos un nodo llamado laser_filters que se subscribe a los mensajes del tópico scan. Se ve como los dos lados están desacoplados. El nodo hoyuko_nodo publica los datos del láser, sin tener ningún conocimiento de quién está escuchando esos mensajes. El nodo laser_filter, recibe los mensajes por el topic scan sin saber si hay alguien publicando. Los dos nodos pueden ser ejectutados, terminados y reiniciados en cualquier oden, sin que ello suponga ningún tipo de error.

Servicios: El modelo publicar/subscribir es un paradigma de comunicación muy flexible, pero su forma de transporte no es adecuada para interacciones solicitar / responder, que son requeridas a menudo en un sistema distribuido. Estas interacciones se realizan mediante los servicios, que son definidos con un par de estructuras de mensaje: Uno para solicitar y otro para responder. Un nodo ofrece un servicio bajo un nombre, y un cliente usa el servicio enviando el mensaje de solicitud y esperando la respuesta.

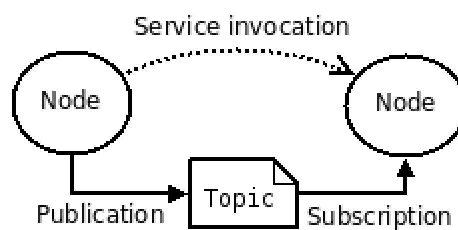

Figura 6-3. Diferencia entre comunicación por topic y servicios [36]

Bags: Los Bags son un formato para guardar y reproducir de nuevo datos de mensajes de ROS. Son un mecanismo importante para guardar datos, como datos de un sensor, que pueden ser difíciles de recolectar, pero son necesarios para desarrollar y probar algoritmos.

El protocolo más usado en ROS es llamado TCPROS, que usa sockets TCP/IP estándar

6.3 Sistema completo. Nodos, tópicos, mensajes y funcionamiento

En este apartado, explicaremos al completo todos los nodos usados para conseguir el correcto funcionamiento del programa.

Antes, recordaremos mediante un esquema el funcionamiento iterativo del algoritmo:

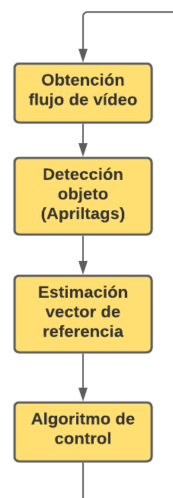

Figura 6-4. Esquema del programa de tracking

6.3.1 Nodo ros_gremsy

Este nodo [38] es una interfaz para controlar los gimbal de Gremsy, basada en la SDK explicada en el apartado 6.1, usando el protocolo MAVLink.

Contiene un archivo de configuración, *config.yaml*, que sirve para establecer diferentes parámetros del gimbal.

El nodo publica en los siguientes tópicos:

- /ros_gremsy/imu/data*:* En este tópico publica un mensaje de ROS de tipo *sensor_msgs/Imu*, que contiene los valores de la velocidad angular y aceleración.
- /ros_gremsy/encoder: En este tópico publica un mensaje de ROS de tipo *geometry_msgs/Vector3Stamped*, que contiene los valores de los encoder.
- /ros_gremsy/mount_orientation_global_yaw: En este tópico se publica un mensaje de ROS de tipo *geometry_msgs/Quaternion*, que contiene la orientación en el eje de coordenadas global.
- /ros_gremsy/mount_orientation_local_yaw: En este tópico se publica un mensaje de ROS de tipo *geometry_msgs/Quaternion*, que contiene la orientación en el eje de coordenadas global para el roll y el pitch, y relativa al vehículo para el yaw.
- /ros_gremsy/angulos_actual: En este tópico se publica un mensaje de ROS de tipo *geometry_msgs/Vector3Stamped*, que contiene los valores de los Ángulos de Euler roll, pitch, yaw en coordenadas globales en los dos primeros y locales en el último.

El nodo se subscribe a los siguientes tópicos:

- /ros_gremsy/goals*:* Espera un mensaje de tipo *geometry_msgs/Vector3Stamped*, que contiene los ángulos deseados para cada eje, o velocidad angular, dependiendo del modo en el que esté el gimbal, configurado en el archivo *config.yaml.*

Como se puede ver, es una adaptación de la SDK de Gremsy para ROS, proporcionando la misma información y permitiendo los mismos comandos, solo que adaptado al entorno de esta herramienta.

6.3.2 Nodo control_gremsy

Este nodo se encargará de recibir el vector de posición del objetivo relativo a la cámara, con lo que calculará la velocidad angular necesaria para garantizar el correcto seguimiento del mismo. Esta velocidad la calcula mediante el algoritmo introducido en el Capítulo 4.

El nodo publica en los siguientes tópicos:

- /ros_gremsy/goals*:* En este tópico publica un mensaje de ROS de tipo *geometry_msgs/Vector3Stamped*, que contiene los valores deseados de velocidad angular para cada motor.

El nodo se subscribe a los siguientes tópicos:

- /ros_gremsy/imu/data*:* Recibe un mensaje de ROS de tipo *sensor_msgs/Imu*, que contiene los valores de la velocidad angular y aceleración.
- /ros_gremsy/encoder: Recibe un mensaje de ROS de tipo *geometry_msgs/Vector3Stamped*, que contiene los valores de los encoder.
- /ros_gremsy/mount_orientation_global_yaw: Recibe un mensaje de ROS de tipo *geometry_msgs/Quaternion*, que contiene la orientación en el eje de coordenadas global.
- /ros_gremsy/mount_orientation_local_yaw: Recibe un mensaje de ROS de tipo *geometry_msgs/Quaternion*, que contiene la orientación en el eje de coordenadas global para el roll y el pitch, y relativa al vehículo para el yaw.

- /ros_gremsy/angulos_actual: Recibe un mensaje de ROS de tipo *geometry_msgs/Vector3Stamped*, que contiene los valores de los Ángulos de Euler roll, pitch, yaw en coordenadas globales en los dos primeros y locales en el último.
- /vector_goals: Recibe un mensaje de ROS de tipo *geometry_msgs/Vector3Stamped,* que contiene el vector de posición del objetivo con respecto a la cámara.
- /detección: Recibe un mensaje de ROS de tipo *std_msgs/Bool*, que contiene una variable booleana que indica si se está produciendo o no la detección del objetivo.
- /sincro: Recibe un mensaje de ROS de tipo *std_msgs/Bool*, que contiene una variable booleana que sirve para sincronizar las gráficas de los datos de interés de los diferentes nodos a la frecuencia requerida.
- /parámetros: Este tópico solo se usa para sintonizar el controlador. Recibe un mensaje de ROS de tipo std_msgs/Float32MultiArray con los parámetros de control deseados.

Para neutralizar el ruido de alta frecuencia con el que se recibe desde la IMU la velocidad angular de la plataforma, hemos utilizado un filtro paso bajo digital de primer orden, cuya ecuación es:

$$w_{filtrada} = w_k * (beta) + w_{k-1} * (1 - beta) \tag{6.1}$$

Con $beta = 0.3$. Se ha elegido un valor alto de la constante del filtro porque el sistema acumula retrasos debido al envío de las imágenes, procesamiento, retraso intrínseco del controlador del gimbal, y un valor bajo de esta constante añadiría mayor retraso al seguimiento.

El código completo de este nodo, escrito en C++, se puede encontrar en el Anexo B.

6.3.3 Nodo Procesamiento

Este nodo se encargará de recibir el flujo de video de la cámara, aplicar la detección mediante el algoritmo de AprilTag, introducido en el apartado 5.1.2 y procesar los resultados, calculando el vector de posición de la forma explicada en el apartado 5.5, incluyendo el Filtro de Kalman Adaptativo explicado en el apartado 5.5.1, para luego enviar ese vector al nodo control_gremsy.

Además, este nodo incluye una funcionalidad diseñada para cuando la cámara pierde de vista al objetivo. Esta funcionalidad tiene en cuenta la última posición registrada del objetivo, para en función de cuál haya sido, buscar al objeto en esa dirección.

Definimos un rectángulo en torno al plano imagen:

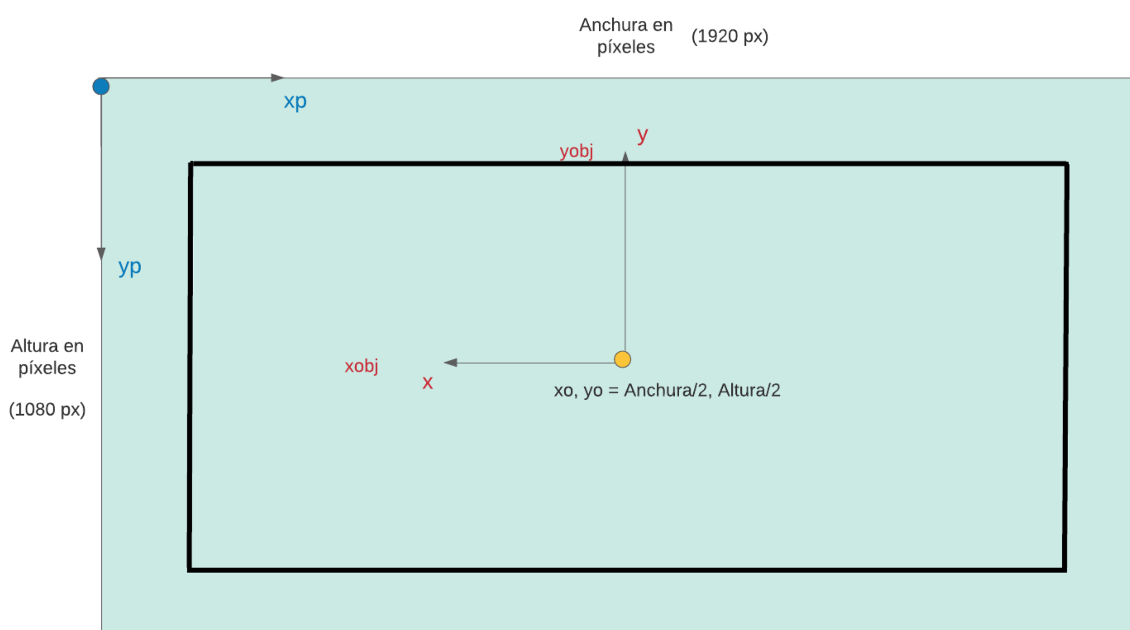

Figura 6-5. Rectángulo usado representado en el plano imagen

Si la última posición registrada del objetivo se encuentra dentro del rectángulo, podemos asumir que es una perturbación por la que se ha perdido la posición del objetivo, por lo que el vector de posición enviado será 0 en el eje x e y, ya que estamos enviando un vector relativo a la cámara (un vector 0 en esos ejes significará que no haya movimiento de la plataforma).

Si la última posición, trasciende los bordes del rectángulo en el eje x (tamaño 1060px), el eje x del vector de posición enviado será calculado como si estuviera en el borde del rectángulo, es decir, será de 500 o -500 multiplicado por la constante de la ecuación 5.19. El eje y del vector de posición lo dejaremos tal cual se ha registrado en la última posición detectada. Como estamos enviando el vector relativo a la cámara, situar ese vector en el rectángulo supondrá un movimiento suave hacia la última posición registrada, por donde tiene gran probabilidad de encontrarse el objetivo, ya que ha salido del plano imagen por esa dirección.

Si la última posición, trasciende los bordes del rectángulo en el eje y (tamaño 600px), el eje x del vector de posición enviado será calculado como si estuviera en el borde del rectángulo, es decir, será de 300 o -300 multiplicado por la constante de la ecuación 5.20. El eje x del vector de posición lo dejaremos tal cual se ha registrado en la última posición detectada.

Si ambos ejes sobrepasan el rectángulo, el vector enviado será el de la esquina correspondiente. Si el gimbal supera unos ángulos umbrales en la búsqueda del objetivo, se devuelve a su estado inicial.

El nodo publica en los siguientes tópicos:

- /vector_goals*:* En este tópico publica un mensaje de ROS de tipo *geometry_msgs/Vector3Stamped*, que contiene el vector del objetivo con respecto a la cámara.
- /detección: En este tópico publica un mensaje de ROS de tipo *std_msgs/Bool*, que contiene una variable booleana que indica si se está produciendo o no la detección del objetivo.

El nodo se subscribe a los siguientes tópicos:

- /ros_gremsy/angulos_actual: Recibe un mensaje de ROS de tipo *geometry_msgs/Vector3Stamped*, que contiene los valores de los Ángulos de Euler roll, pitch, yaw en coordenadas globales en los dos primeros y locales en el último.
- /sincro: Recibe un mensaje de ROS de tipo *std_msgs/Bool*, que contiene una variable booleana que sirve para sincronizar las gráficas de los datos de interés de los diferentes nodos a la frecuencia requerida.
- /UAV_1/usb_cam/Image_raw: Recibe un mensaje de ROS de tipo *sensor_msgs/Image*, que contiene una imagen capturada por la cámara.

El código completo de este nodo, escrito en Python, se encuentra en el Anexo C.

6.3.4 Nodo Grabación

Este nodo se usa para capturar el vídeo grabado desde la cámara.

El nodo se subscribe a los siguientes tópicos:

- /UAV_1/usb_cam/Image_raw: Recibe un mensaje de ROS de tipo *sensor_msgs/Image*, que contiene una imagen capturada por la cámara.

El código completo de este nodo, escrito en Python, se encuentra en el Anexo D.

6.3.5 Nodo Sincronización

Este nodo se usa para obtener en el mismo instante de tiempo las gráficas con los datos que nos interesan de cara a evaluar el programa.

El nodo publica en los siguientes tópicos:

- /sincro: Envía un mensaje de ROS de tipo *std_msgs/Bool*, que contiene una variable booleana que sirve para sincronizar las gráficas de los datos de interés de los diferentes nodos a la frecuencia requerida.

El código completo de este nodo, escrito en Python, se encuentra en el Anexo E.

6.3.5 Nodo Parámetros

Este nodo es de utilidad para sintonizar los parámetros del algoritmo usado en el control en velocidad.

El nodo publica en los siguientes tópicos:

- /parámetros: Este tópico solo se usa para sintonizar el controlador. Envía un mensaje de ROS de tipo std_msgs/Float32MultiArray con los parámetros de control deseados.

Los parámetros se introducen por pantalla, con esta interfaz:

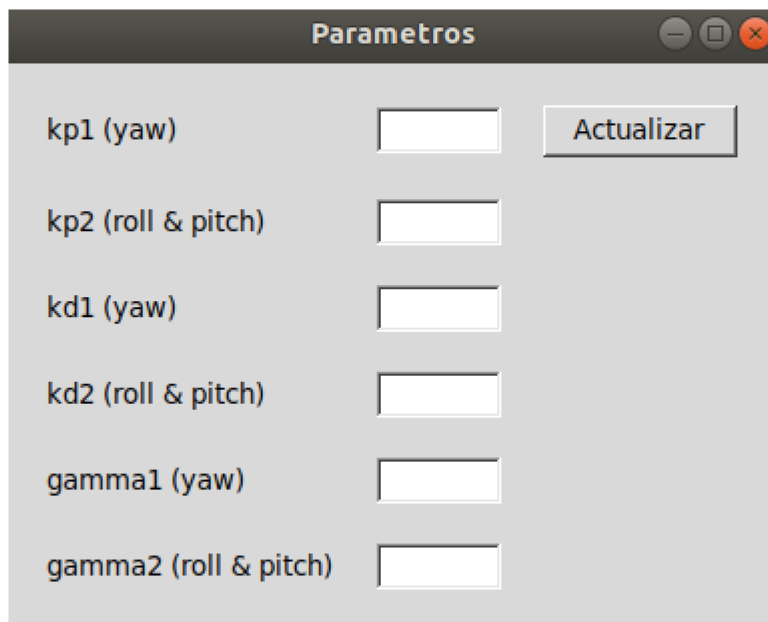

Figura 6-6. Interfaz gráfica para introducir los parámetros deseados

El código completo de este nodo, escrito en Python, se encuentra en el Anexo F.

6.3.6 Nodo usb_cam

Este nodo [39] sirve para crear una interfaz con diferentes tipos de cámara.

El nodo publica en los siguientes tópicos:

- /UAV_1/usb_cam/Image_raw: Envía un mensaje de ROS de tipo *sensor_msgs/Image*, que contiene una imagen capturada por la cámara.

6.3.7 Esquema general

Usando la herramienta *rqt_graph* [40], podemos obtener un esquema en tiempo real de los nodos y tópicos que están en funcionamiento:

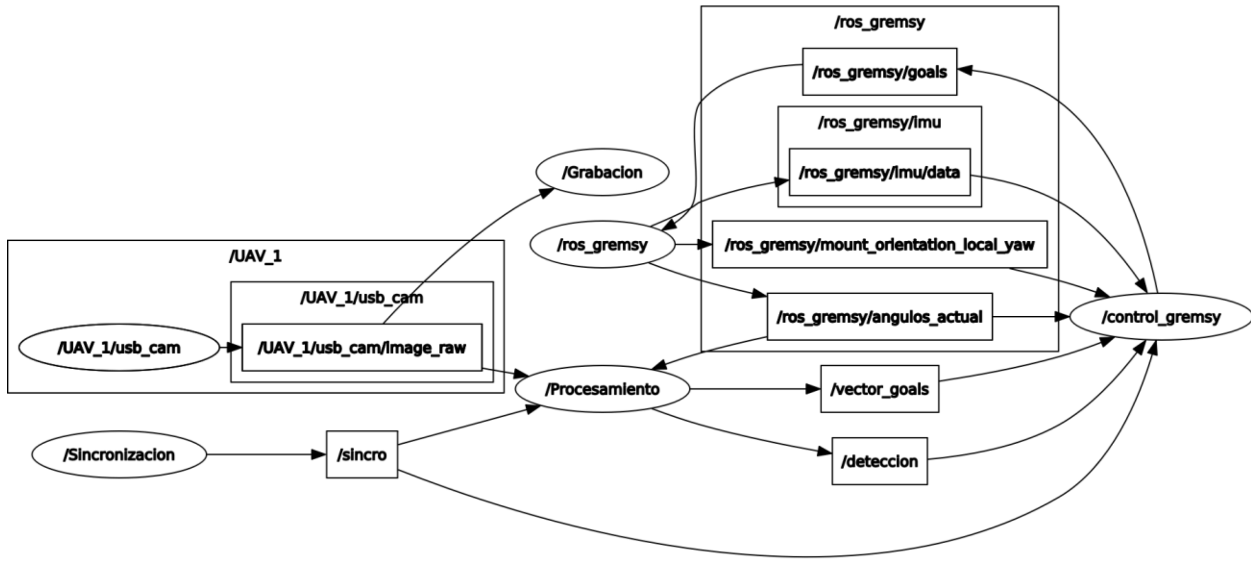

Figura 6-7. rqt_graph durante el funcionamiento del programa

En este gráfico no estábamos usando el nodo Parámetros.

A continuación, mostramos otro gráfico que tal vez contenga de forma menos liosa todos los nodos y tópicos:

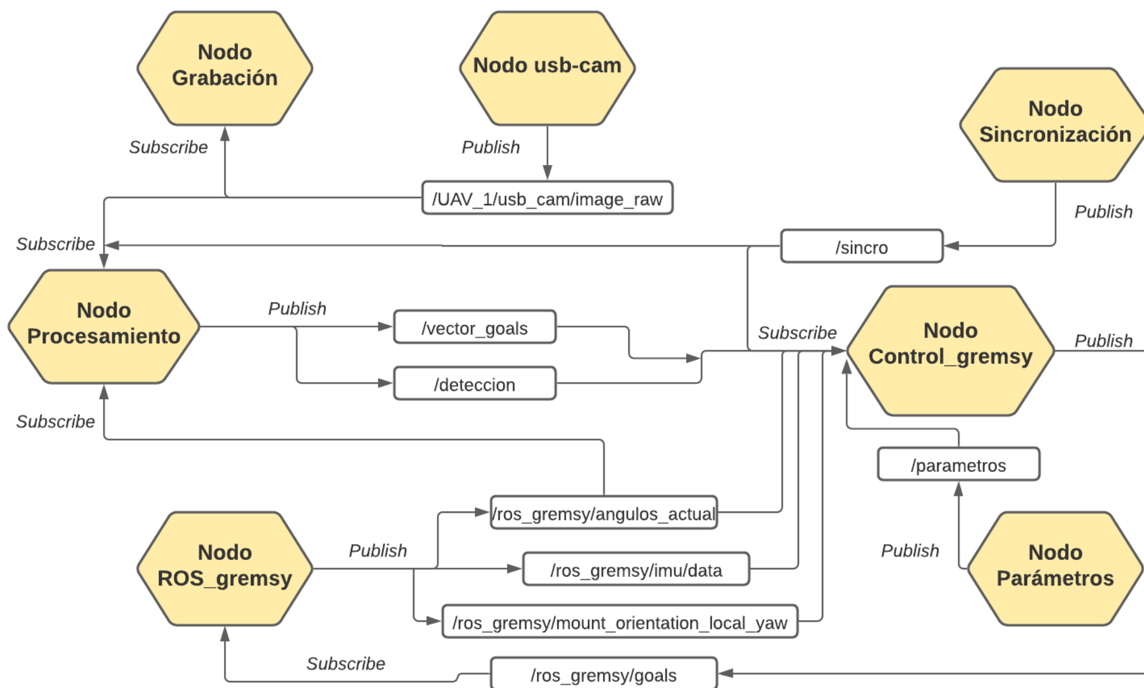

Figura 6-8. Nodos y tópicos del programa completo

7 RESULTADOS

En este capítulo, analizaremos los resultados obtenidos en varios tipos diferentes de pruebas. En primer lugar, veremos si el sistema se comporta correctamente en un entorno en el que el objetivo nunca sale del campo de visión. Una vez comprobado el funcionamiento en el supuesto más sencillo, probaremos la funcionalidad en la que la plataforma intenta perseguir al objetivo que sale del campo de visión. Luego, nos cercionaremos de que el sistema funciona correctamente cuando movemos el UAV en el que se encuentra la plataforma, para asegurarnos de que es viable la aplicación en vuelo del algoritmo.

Además, supondremos en todas las pruebas una distancia al objetivo de 1 metro, comprobando de esta forma que el algoritmo es robusto frente a distancias diferentes de la real.

7.1 Prueba 1. Objetivo siempre en el rango de visión

Los parámetros de control usados en esta prueba los tenemos en la siguiente tabla:

Tabla 7–1. Parámetros de control. Prueba 1

Kp (yaw)	8
Kp (roll)	0.0001
Kp (pitch)	8
Kd (yaw)	0.0005
Kd (roll)	0.0005
Kd (pitch)	0.0005
Gamma (yaw)	25
Gamma (roll)	15
Gamma (pitch)	15

Al poder desacoplar en controlador en 3 ejes, podemos usar diferentes parámetros para cada uno, ya que cada uno de los ejes tiene sus propias inercias.

A continuación, analizaremos gráficamente los datos obtenidos directamente de la prueba. Como se ha comentado en la introducción, en esta primera prueba el objetivo nunca saldrá del campo de visión, por lo que se comprobará el funcionamiento correcto del algoritmo de seguimiento sin búsqueda de objetivo.

En primer lugar, mostraremos los datos relativos al control de la plataforma, es decir, todos aquellos que se obtienen del nodo control_gremsy.

Comenzaremos con los datos de los cuaternios utilizados:

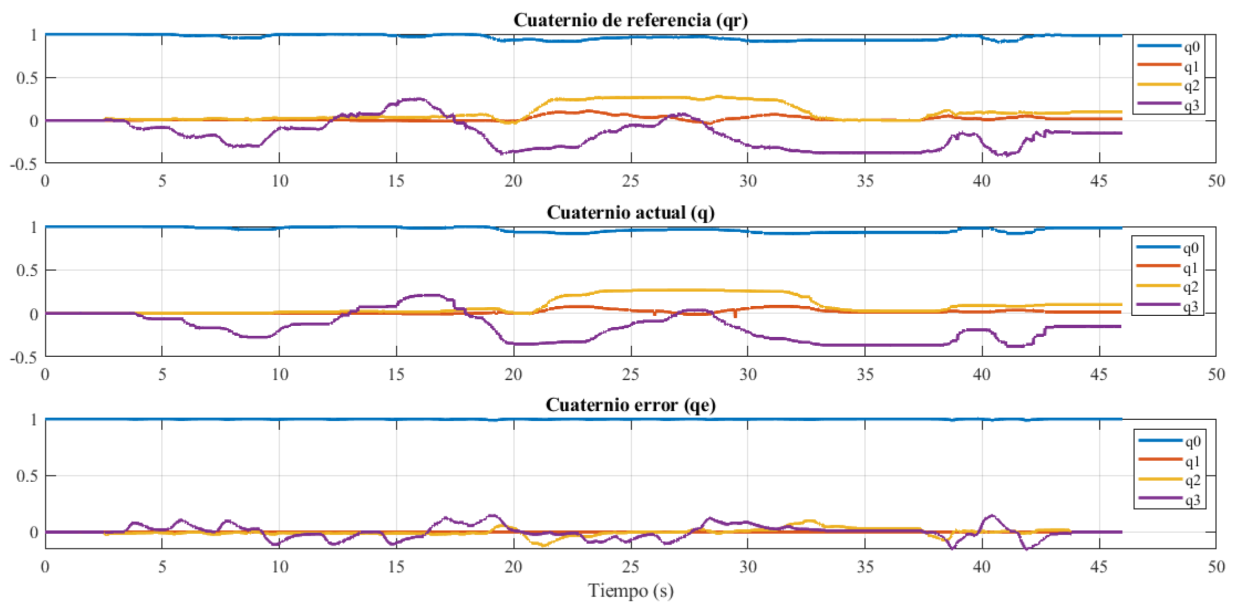

Figura 7-1. Representación gráfica de cuaternios. Prueba 1

Podemos observar que la orientación de la plataforma expresada en cuaternio q, sigue perfectamente la orientación de referencia q_r. El cuaternio de error q_e tiende a cero en cada uno de los movimientos realizados por el objetivo, por lo que se puede considerar que el desempeño del controlador es acorde a lo deseado.

Ahora veremos cómo evoluciona el vector de apuntamiento de la cámara:

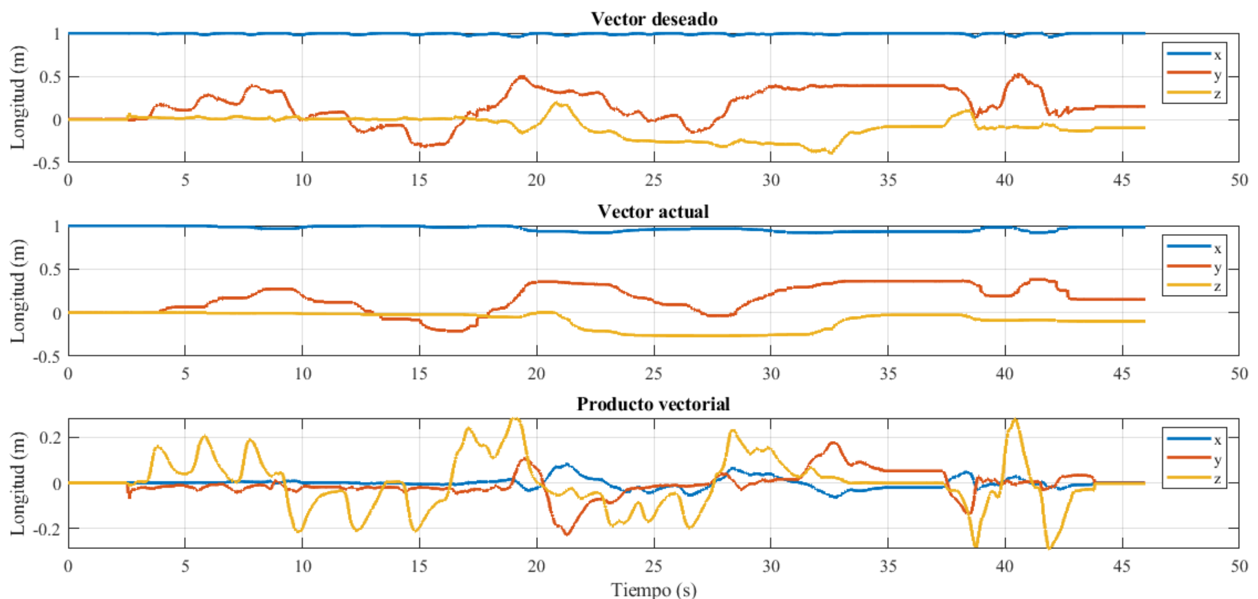

Figura 7-2. Representación gráfica del vector de apuntamiento. Prueba 1

En esta gráfica, vemos igualmente cómo el vector de apuntamiento de la cámara sigue en la prueba al vector de apuntamiento deseado, mostrando algo más de error que en el caso de los cuaternios, pero manteniéndose estable.

Continuamos comprobando las velocidades angulares comandadas (no olvidemos que la referencia que enviamos al gimbal son las velocidades deseadas), las medidas, y las filtradas:

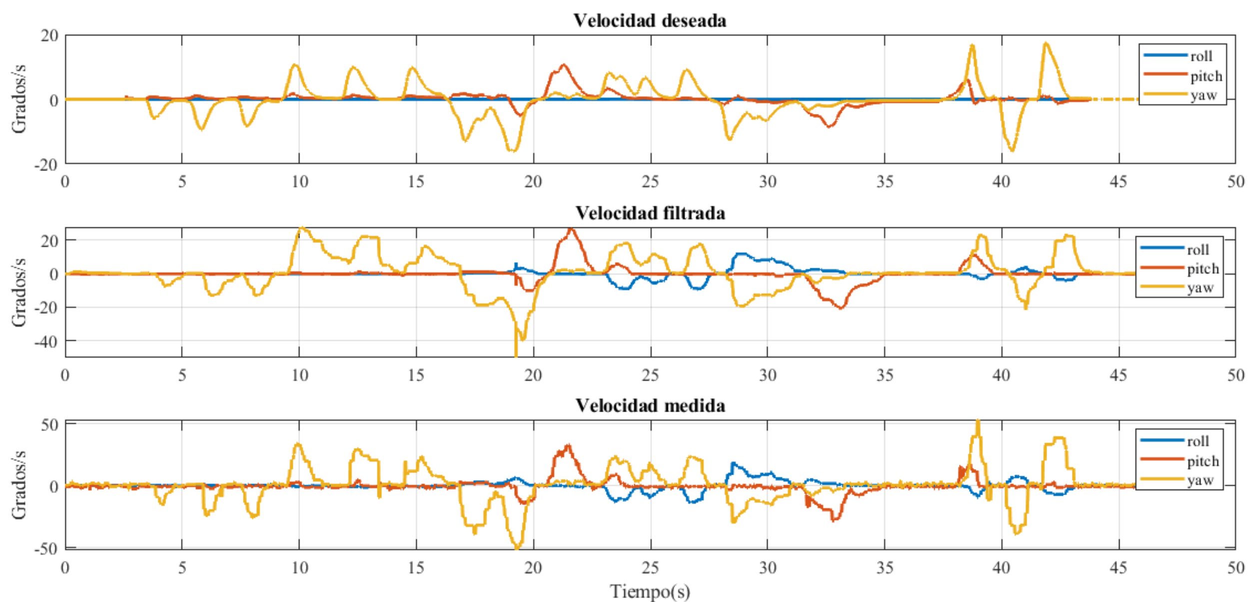

Figura 7-3. Representación de las velocidades angulares. Prueba 1

Observamos que las velocidades medidas son consecuentes con las velocidades que estamos enviando al controlador del gimbal. El filtro paso bajo funciona bien, anulando las componentes de alta frecuencia y suavizando la señal sin introducir un retraso reseñable.

Por último, mostraremos gráficamente la orientación en la otra representación que hemos analizado en este trabajo, los Ángulos de Euler. Los ángulos que se muestran están expresados en coordenadas globales, excepto el yaw, que es expresado con respecto al vehículo.

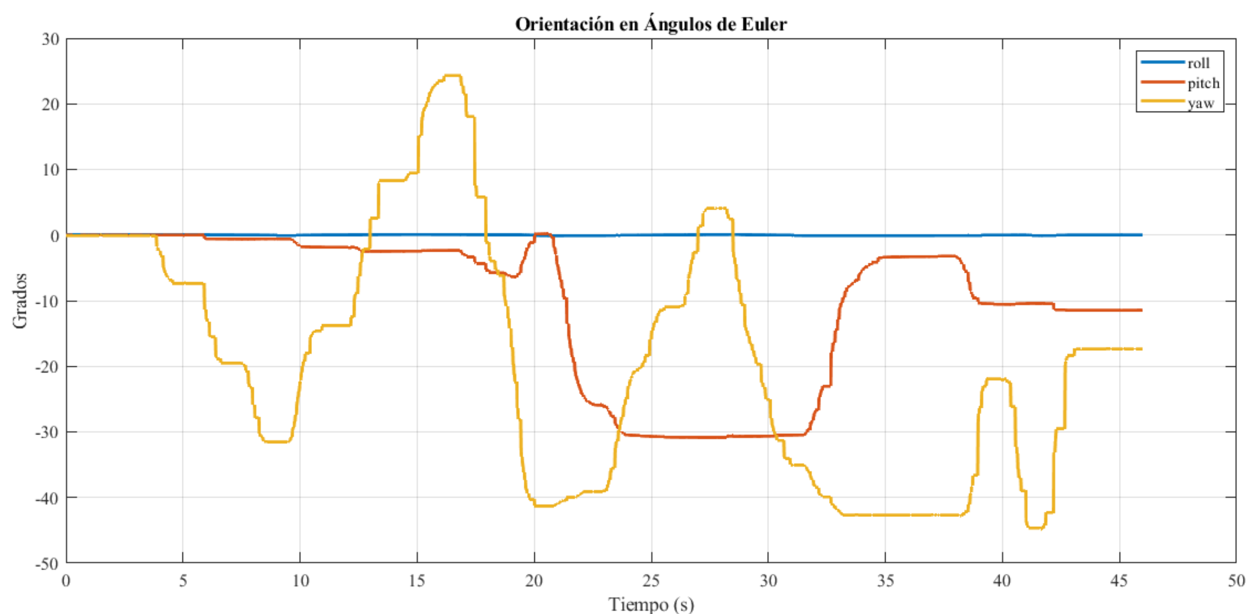

Figura 7-4. Representación de la orientación expresada en Ángulos de Euler. Prueba 1

Una vez hemos terminado con las gráficas del nodo de control, pasaremos a analizar los datos del nodo Procesamiento.

En primer lugar, analizaremos los valores del vector de posición estimado del objetivo relativo a la cámara.

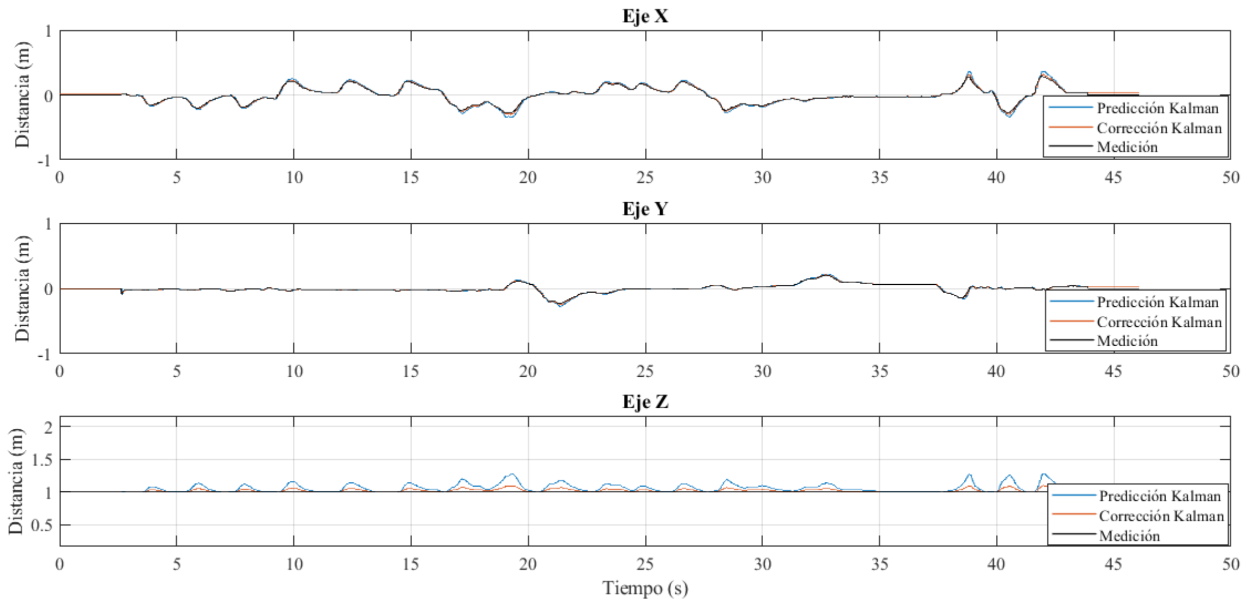

Figura 7-5. Representación gráfica del vector de posición del objetivo estimado. Prueba 1.

Recordamos que este vector de posición estimado es el que se envía al nodo control_gremsy para realizar el algoritmo de control. Como suponemos que la distancia es constante e igual a 1, la Z del vector que enviamos es siempre 1. Sin embargo, para los ejes X e Y, se envía la corrección del vector realizada con el Filtro de Kalman Adaptativo. Se puede observar que el vector tiende a cero cada vez que se mueve el objetivo, por lo que indica que se está centrando en el plano imagen adecuadamente.

Vamos a ver una imagen ampliada de las componentes X e Y en diferentes instantes de tiempo, en los que se produce un cambio rápido en la posición del objetivo, para comprobar los valores proporcionados por el Filtro de Kalman.

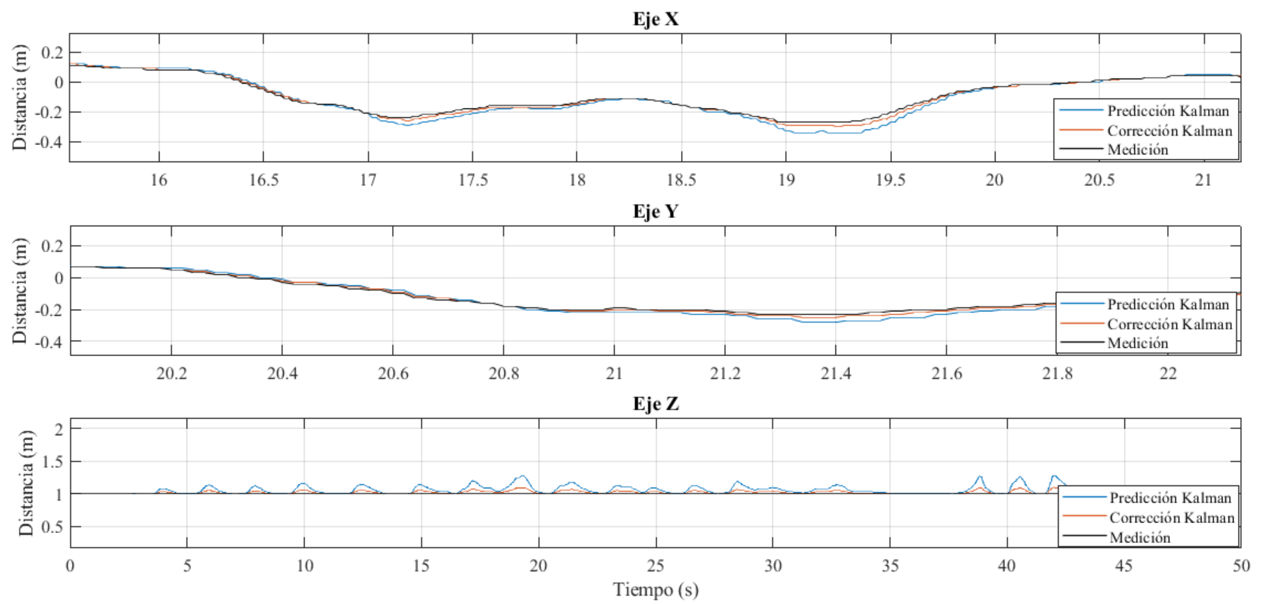

Figura 7-6. Representación gráfica del vector de posición. Ampliado. Prueba 1

Se puede ver que la predicción del Filtro de Kalman Adaptativo (color azul) tiende a aumentar conforme el movimiento del objetivo se ve reflejado en la gráfica, ya que los valores de la matriz de transición son calculados mediante los estados anteriores.

La corrección (color rojo) pondera entre la medición (color negro) y la predicción, y es el valor que usamos para realizar el control.

Para apoyar a la figura del vector de posición relativo, mostraremos cómo evoluciona la distancia en píxeles del objetivo detectado al centro de la imagen, lo que confirmará que se produce un centrado del objeto en el plano.

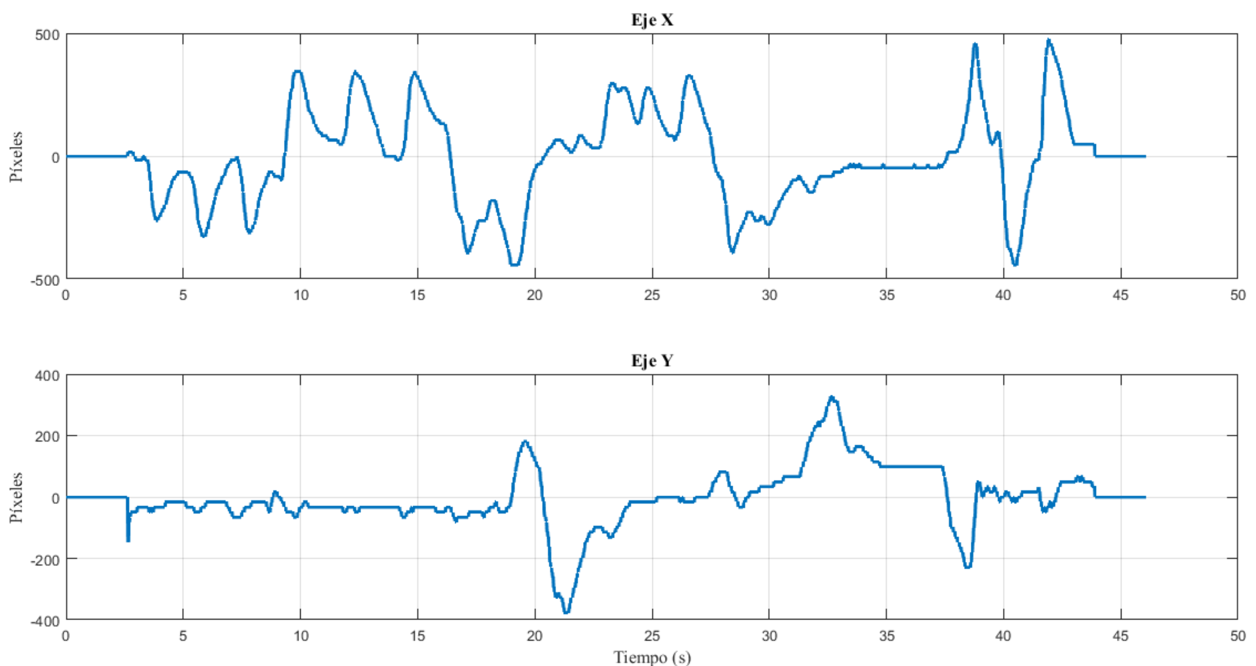

Figura 7-7. Representación de la distancia en píxeles al centro del objeto en el plano imagen. Prueba 1

Observamos que se confirma la tendencia hacia el centro de la imagen de la detección del objetivo.

Ahora, mostramos gráficamente cómo evoluciona el valor de la traza matriz P de covarianza del algoritmo del Filtro de Kalman Adaptativo.

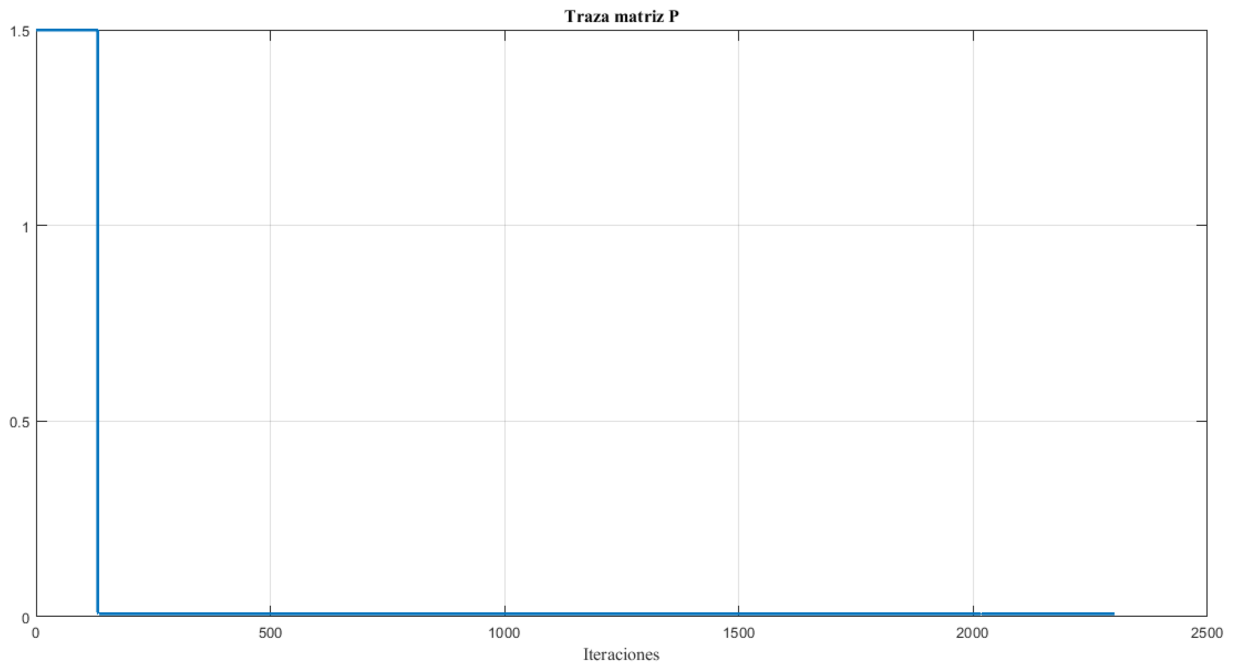

Figura 7-8. Traza de la matriz P. Prueba 1

La matriz evoluciona hasta un valor bajo, que concretamente es algo menor a 0.01.

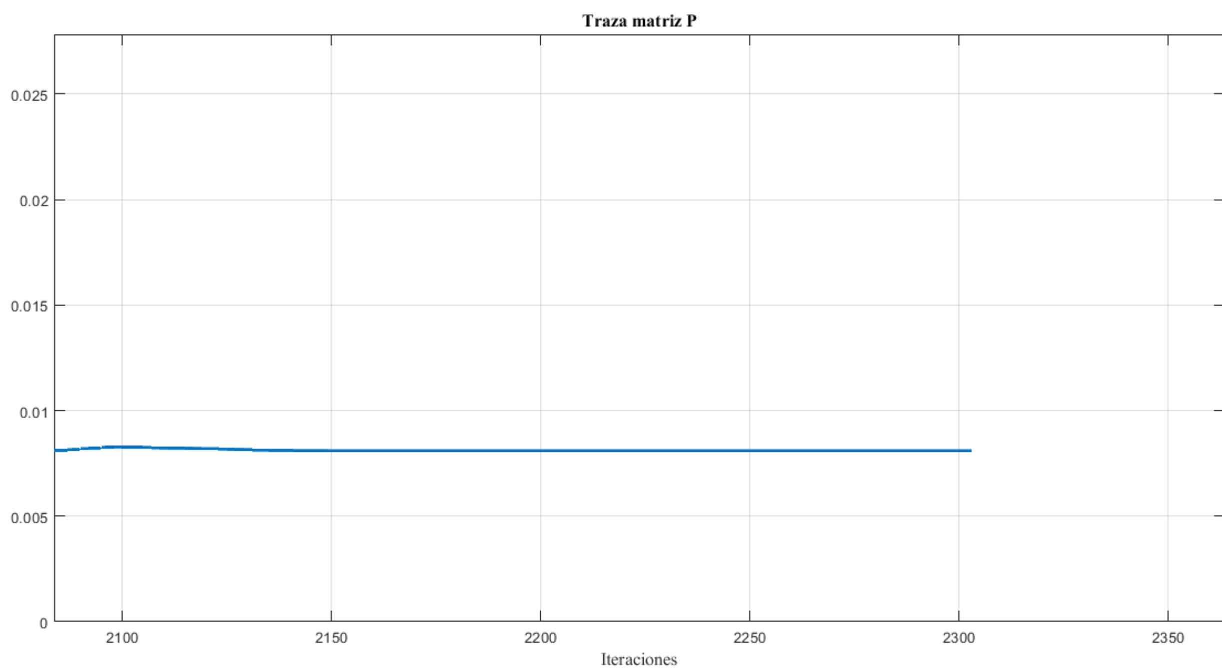

Figura 7-9. Traza matriz P. Ampliado. Prueba 1

Siguiendo con el análisis de las matrices del filtro, vemos cómo varía la traza de la matriz de ganancia K.

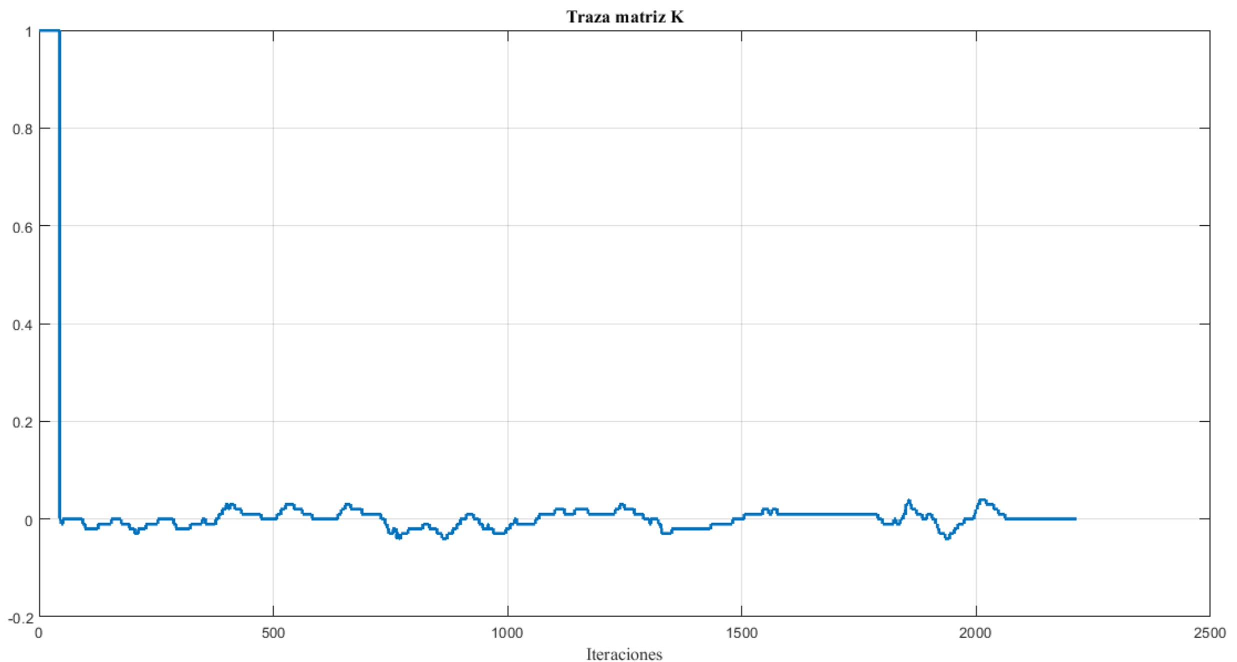

Figura 7-10. Traza matriz K. Prueba 1.

Esta ganancia de Kalman va variando en el tiempo, de forma que pondera más o menos la diferencia entre la medición y la estimación, pero en general oscila sobre el cero, por lo que da la mayoría del peso a la estimación, es decir, a la predicción obtenida por la matriz de transición adaptativa.

Para concluir con el análisis del comportamiento del Filtro de Kalman Adaptativo, mostraremos una figura en la que se observa cómo evolucionan las componentes de la matriz A, la matriz de transición, a lo largo de la prueba.

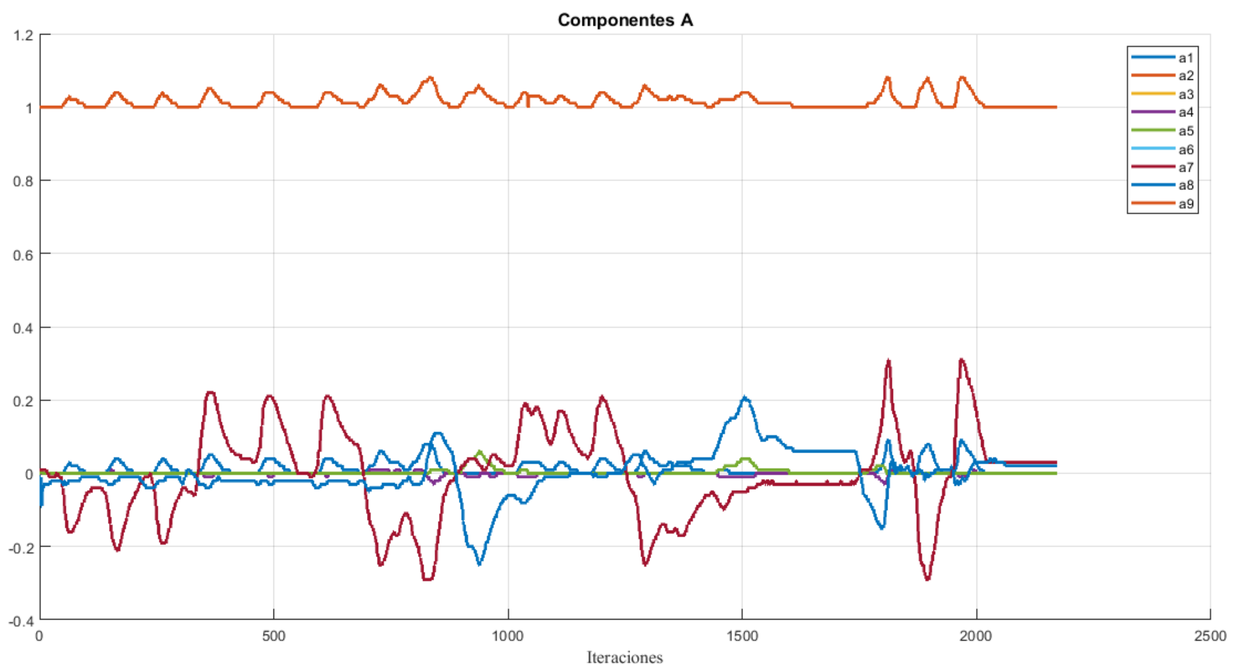

Figura 7-11. Componentes de la matriz A. Prueba 1

Para dar una idea al lector de cómo se desarrollan las pruebas, plasmamos aquí varios fotogramas sacados directamente de la grabación de la cámara, en la que además se ve representada la detección del marcador de AprilTag.

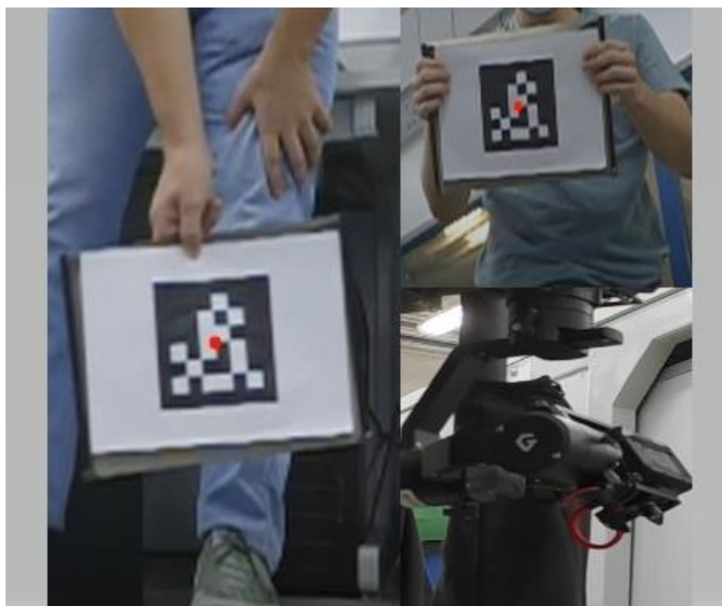

Figura 7-12. Imágenes de la prueba

7.2 Prueba 2. El objetivo escapa del rango de vision

En esta prueba, comprobaremos el correcto funcionamiento de la solución propuesta para el problema de perder al objetivo.

Los parámetros de control usados son los mismos que los usados en la Prueba 1, que se pueden encontrar en la Tabla 7.1.

En esta prueba, como en las siguientes, nos limitaremos a introducir las gráficas más importantes y relevantes, para no sobrecargar la memoria de figuras.

En el control, podemos ver que la orientación de la plataforma sigue perfectamente la referencia.

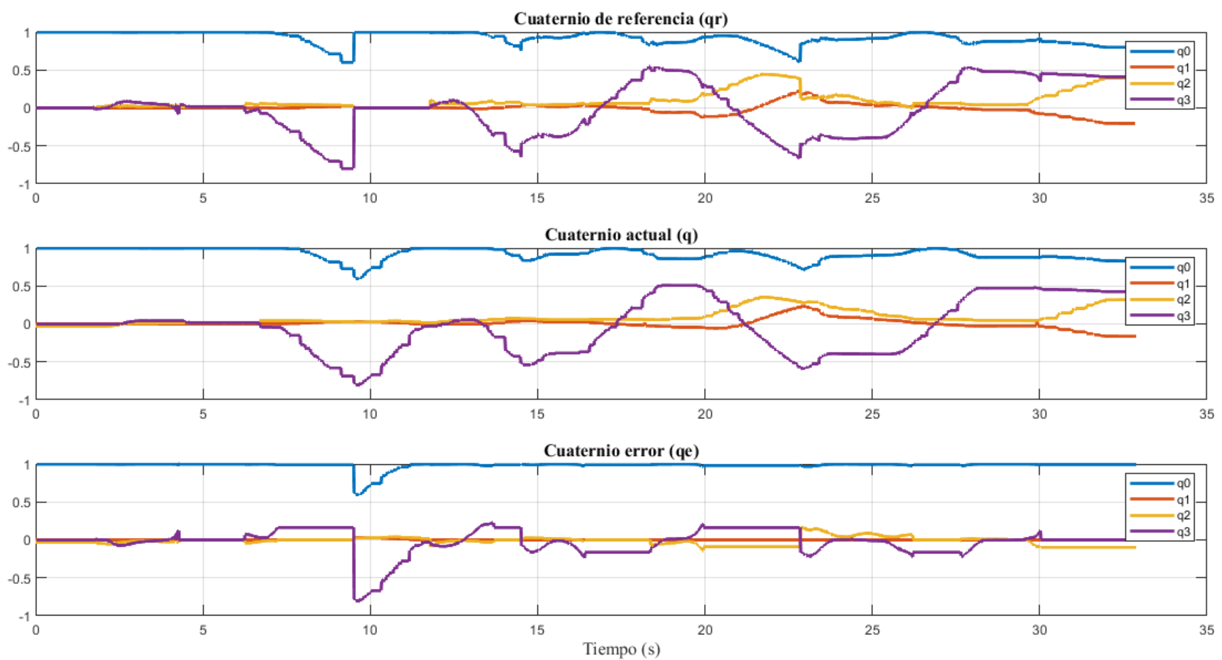

Figura 7-13. Representación gráfica de cuaternios. Prueba 2

Para esta prueba, es más interesante apreciar las gráficas provenientes del nodo Procesamiento.

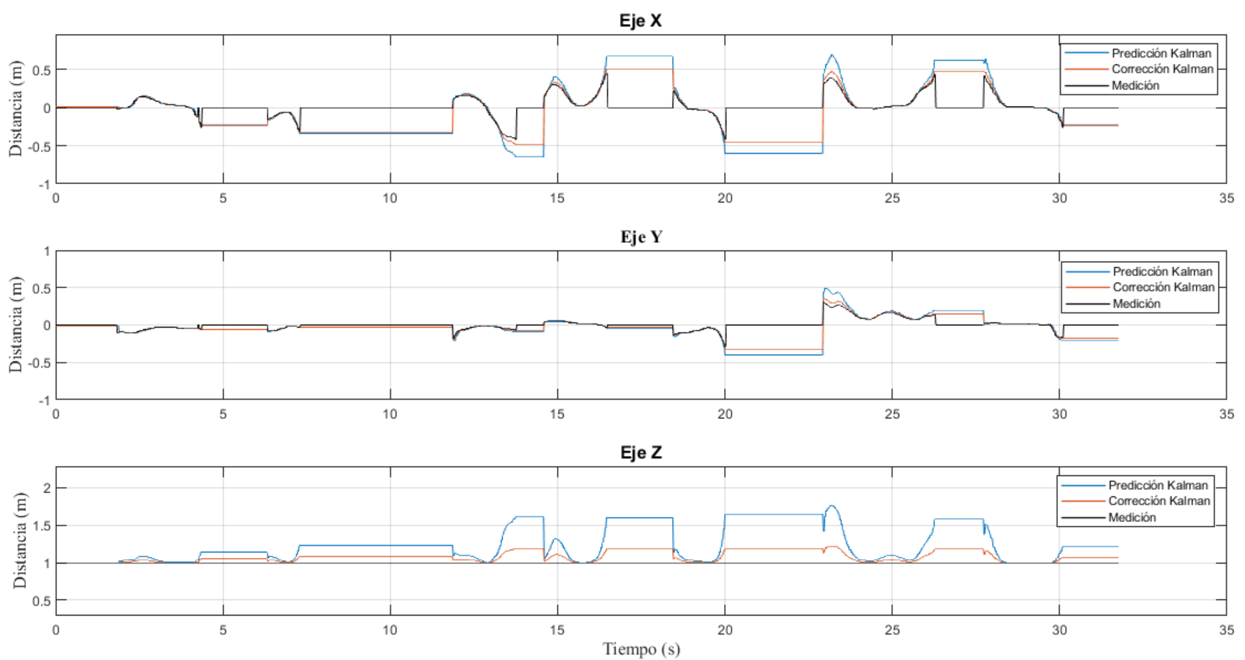

Figura 7-14. Representación gráfica del vector de posición. Ampliado. Prueba 1

Para explicar bien el funcionamiento de esta funcionalidad, nos vamos a centrar en lo que ocurre entre los segundos 7 y 12 aproximadamente. La cámara pierde al objetivo, y vemos que el vector de posición que se envía al controlador (relativo a la cámara) es constante, fijado por el rectángulo que definimos en el apartado 6.3.3, Figura 6.5. Eso provoca un movimiento de la plataforma que se puede confirmar en cómo evoluciona el cuaternio de referencia y el cuaternio de orientación actual en la Figura 7.13. Este movimiento de búsqueda lo realiza hasta que alcanza una orientación límite. Llegado a ese punto, la referencia lo lleva hacia el origen.

7.3 Prueba 3. UAV en movimiento

En esta prueba, oscilaremos el UAV de forma manual, con el objetivo de ver cómo se comporta el algoritmo frente a estas perturbaciones, ya que el objetivo final de este tipo de algoritmos (aunque hay muchos supuestos en los que no necesariamente van a estar en un vehículo, como, por ejemplo, una cámara de seguridad) es usarse para rastrear objetivos desde drones o UGV (Unmanned Ground Vehicle).

Los parámetros de control usados son los mismos que los usados en la Prueba 1, que se pueden encontrar en la Tabla 7.1.

En esta prueba, enseguida nos damos cuenta de que se presentan unas extrañas oscilaciones cuando los ángulos del gimbal son muy extremos.

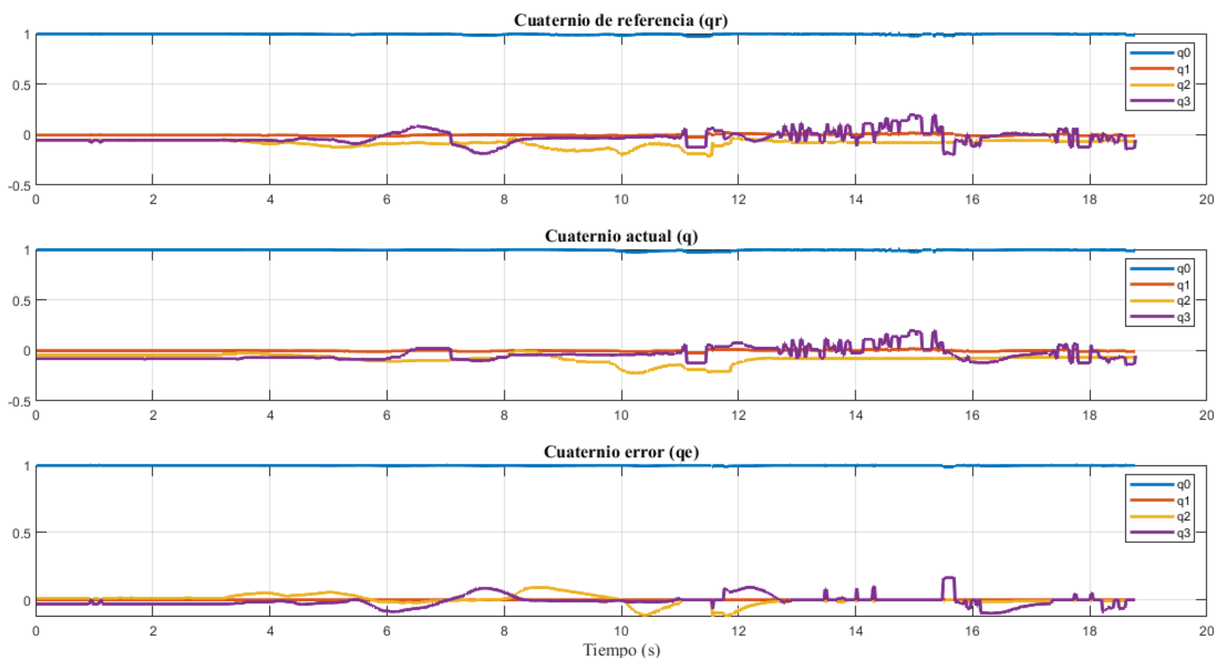

Figura 7-15. Representación gráfica de cuaternios. Prueba 3

Al estar el objetivo quieto y el dron girando, el gimbal tiene que mantener siempre la orientación inicial de forma aproximada. Se puede observar que lo hace perfectamente, hasta que se produce el comportamiento oscilatorio.

Aquí podemos ver la orientación expresada en Ángulos de Euler.

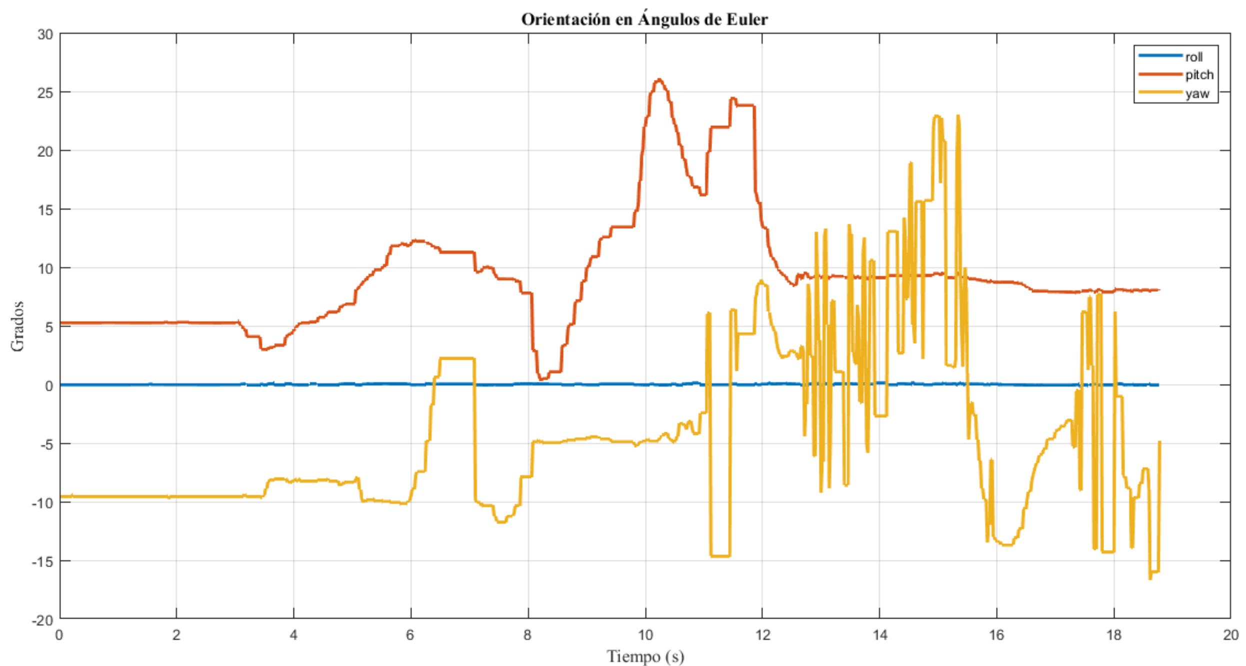

Figura 7-16. Representación gráfica de orientación en Ángulos de Euler. Prueba 3

Observamos que es el ángulo de yaw el que oscila.

Tras investigar de forma exhaustiva el problema, decidimos cambiar la forma de control del gimbal a posición, para ver si así se consigue mejor estabilidad en este tipo de supuestos.

7.2 Prueba 4. UAV en movimiento. Control en posición

Antes de comenzar con el análisis de la prueba, nos dentendremos a explicar el cambio que se ha realizado en el algoritmo de control, específicamente en el nodo control_gremsy.

Como explicamos en el Capítulo 6, al gimbal se le podia comandar en modo velocidad angular y en modo ángulos. Esto consiste en que, simplemente, enviaremos una orientación al nodo ros_gremsy y el gimbal la replicará.

Para ello, nos remontamos al algoritmo de control explicado en el Capítulo 4, concretamente a la ecuación 4.7:

$$q_e = {q_d}^* \circ q \tag{4.7}$$

En el algoritmo de control en velocidad, se usaba esta orientación q_d para modelar un controlador proporcional-derivativo. En nuestro algortimo de control en posición, usaremos esta orientación para comandarlos directamente a la plataforma. Como esta orientación q_d expresa el giro necesario relativo a la cámara para colocar el objetivo en el centro de la imagen, multiplicando ese cuaternio por la orientación actual, obtendremos la nueva orientación que tenemos que comandar:

$$q_{comand} = q_d \circ q \tag{7.1}$$

El código completo del algoritmo de control en posición, es decir, del nuevo nodo control_gremsy, se encuentra desarrollado en el Anexo G.

La prueba que se ha realizado es muy parecida a la realizada en la Prueba 3, para comprobar si el nuevo programa es capaz de comportarse de forma adecuada ante el movimiento del dron. La diferencia con la Prueba 3 es que, al principio, se realizan unos movimientos con el dron quieto para comprobar que puede comportarse correctamente en este supuesto al igual que el algoritmo anterior.

Continuando, procedemos a analizar las gráficas del control.

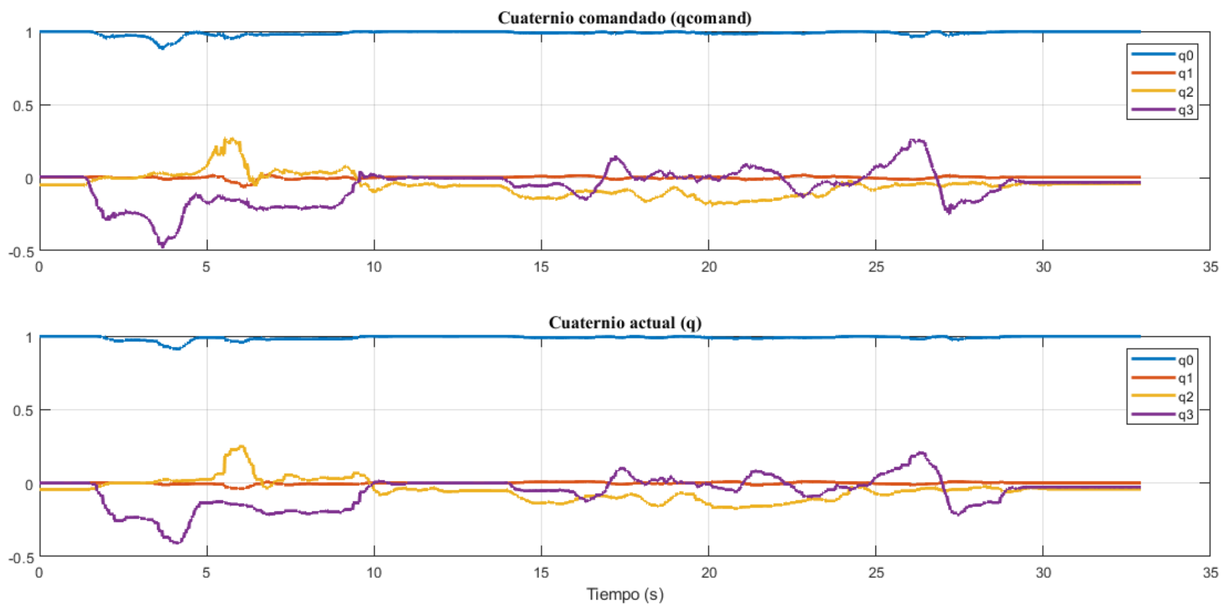

Figura 7-17. Representación de cuaternios. Prueba 4

En los primeros 10 segundos de simulación, el objeto está en movimiento. El cuaternio comandado es correctamente alcanzado por la plataforma.

Luego, durante el resto de la prueba, la orientación se mantiene aproximadamente estable, mientras realizamos diferentes giros y movimientos con el UAV. Claramente se han eliminado las molestas oscilaciones que teníamos en la Prueba 3.

Para confirmar el buen funcionamiento del algoritmo, terminaremos analizando las gráficas obtenidas del algoritmo de procesamiento de la imagen, del nodo Procesamiento.

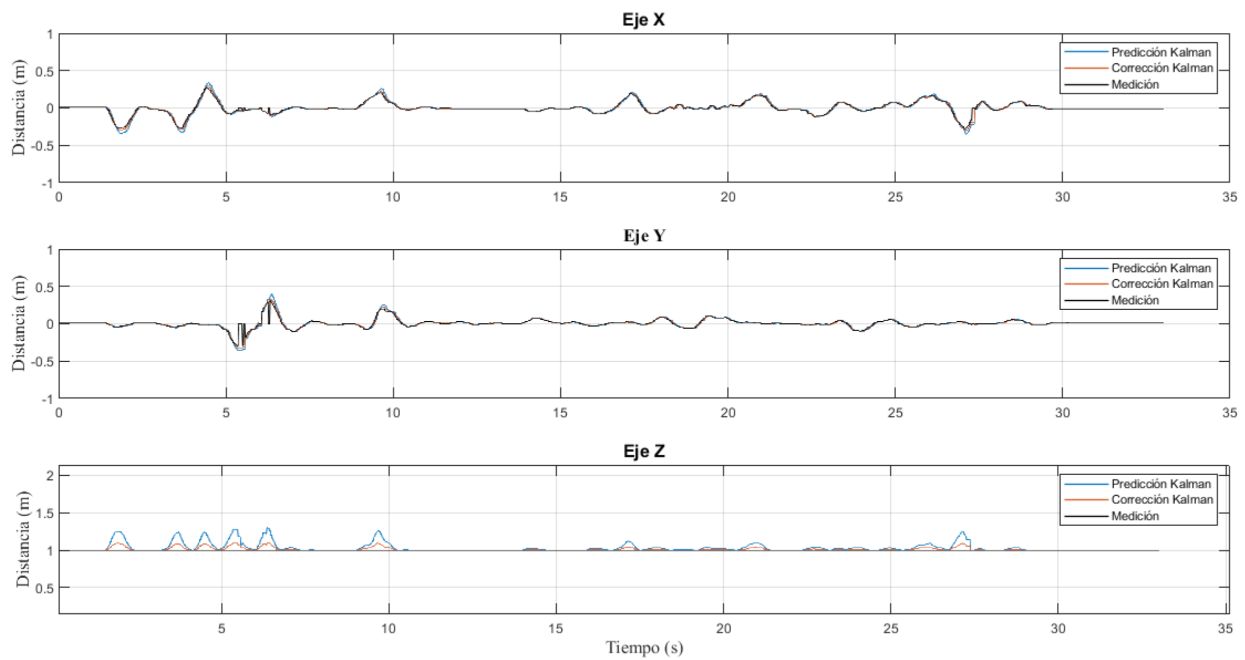

Figura 7-18. Representación del vector del objetivo relativo a la cámara. Prueba 4

El vector tiende a cero durante toda la simulación, lo que indica que el objetivo se está manteniendo correctamente en el centro del plano imagen.

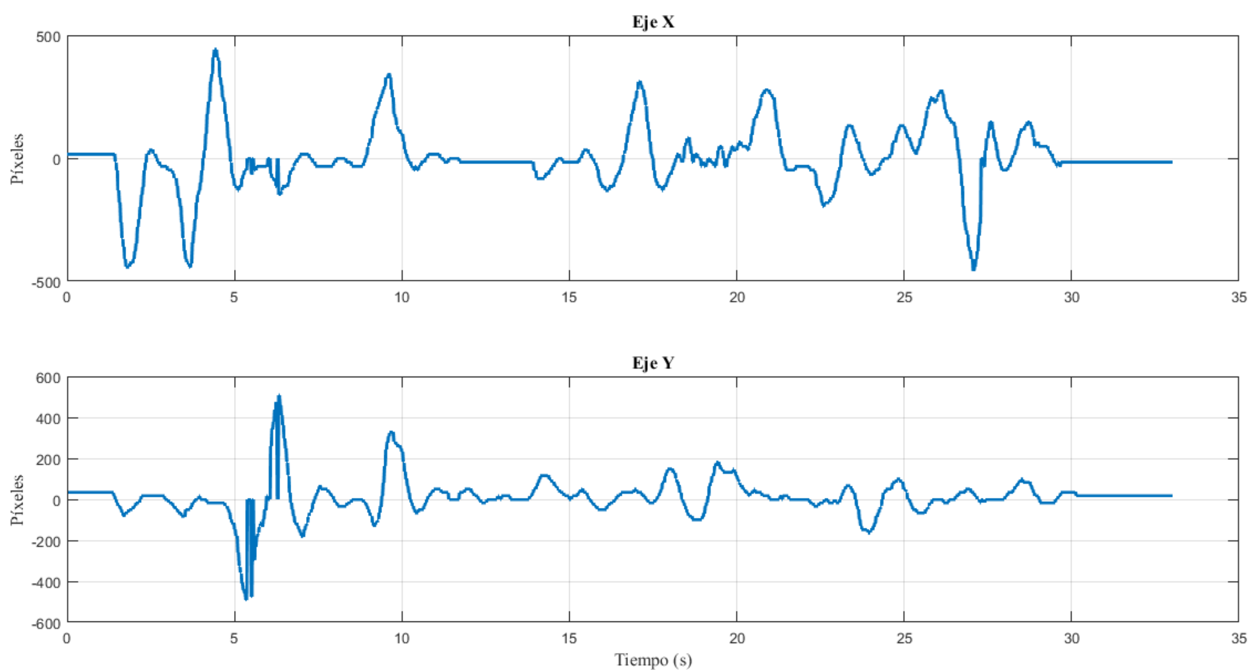

Figura 7-19. Representación de la distancia en píxeles al centro del objeto en el plano imagen. Prueba 4

Igualmente, la distancia en píxeles del objetivo al centro de la imagen también tiene a cero durante toda la simulación, por lo que damos por todo un éxito el nuevo algoritmo de control en posición.

Con esto, hemos terminado de exponer las pruebas realizadas en este trabajo.

8 CONCLUSIÓN Y TRABAJOS FUTUROS

Terminamos este proyecto cumpliendo los objetivos que establecimos en la introducción. Tenemos un algoritmo completamente capaz de realizar la función de seguimiento de objetivo, centrado a este en el plano de la imagen. Además, los beneficios de haber usado un sistema operativo como ROS nos permitirán usar este programa en el futuro para múltiples sistemas compatibles con este entorno.

Los objetivos que seguirían la línea temporal de este trabajo serían varios. Primeramente, habría que desarrollar un algoritmo de detección en función de qué tipos de objetivos queremos seguir. Esta es una tarea que bien podría ser desarrollada en otro Trabajo de Fin de Grado.

Seguidamente, nos faltaría comprobar el funcionamiento de un sistema de estimación de la posición como el de la triangulación, el cual nos permitiría saber con exactitud la situación del objetivo si tenemos varias cámaras. Esto sería muy útil para diferentes casos de uso.

Finalmente, hubiera sido ideal probar este algoritmo en un vuelo real, pero no ha sido posible.

REFERENCIAS

[1] D. Z. Rabanal Carretero, «Integración de un sistema UAV con control autónomo en un equipo aéreo para agricultura de precisión,» *Pontificia Universidad Católica del Perú,* 2011.

[2] W. A. Richard y R. T. Rysdyk, «UAV Coordination for Autonomous Target Tracking,» *University of Washington,* 2006.

[3] S. Jung y H. Kim, «Analysis of Amazon Prime Air UAV Delivery Service,» *Journal of Knowledge Information Technology and Systems,* vol. 12, nº 2, pp. 253-266, 2017.

[4] R. Rajesh y P. Kavitha, «Camera Gimbal Stabilization Using Conventional PID Controller And Evolutionary Algorithms,» de *IEEE International Conference on Computer Communication and Control, IC4 2015*, Indore, 2015.

[5] E. W. Weisstein, «"Euler's Rotation Theorem." From MathWorld--A Wolfram Web Resource.,» [En línea]. Available: https://mathworld.wolfram.com/RotationMatrix.html.

[6] E. W. Weisstein, «"Rotation Matrix." From MathWorld--A Wolfram Web Resource.,» [En línea]. Available: https://mathworld.wolfram.com/EulersRotationTheorem.html.

[7] J. B. Kuipers, «Quaternions and Rotation Sequences,» de *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace and Virtual Reality*, Princeton University Press, 1999, pp. 127-142.

[8] A. L. Schwab, «Quaternions, Finite Rotation and Euler Parameters,» *Laboratory for Engineering Mechanics, Delft University of Technology,* 2002.

[9] H. Abaunza , «Robust Tracking of Dynamic Targets with Aerial Vehicles Using Quaternion-based Techniques,» *Université de Technologie de Compiègne,* 2019.

[10] L. Perumal, «Quaternion and Its Application in Rotation Using Sets of Regions,» de *International Journal of Engineering and Technology Innovation, 35-52, 1(1)*, 2011.

[11] «Apollo11Space,» [En línea]. Available: https://apollo11space.com/apollo-and-gimbal-lock/.

[12] «https://gremsy.com/pixy-u-spec,» [En línea].

[13] D. Mery, Visión por Computador, Santiago de Chile: Universidad Católica de Chile, 2004.

[14] «https://es.wikipedia.org/wiki/C%C3%A1mara_oscura,» [En línea].

[15] «https://www.xatakafoto.com/camaras/sensores-con-tecnologia-ccd-vs-cmos,» [En línea].

[16] R. Hartley y A. Zisserman, Multiple View Geometry, 2000.

[17] «https://www.sony.es/electronics/camaras-compactas-cyber-shot/dsc-

rx0m2/specifications#specifications,» [En línea].

[18] «Simulink. Simulación y diseño basado en modelos.,» [En línea]. Available: https://es.mathworks.com/products/simulink.html?s_tid=srchtitle.

[19] A. I. Barranco Gutiérrez, «Visión esteroscópica por computadora,» *Instituto Politécnico Nacional. Centro de Investigación en Ciencia Aplicada y Tecnología Avanzada,* 2007.

[20] «OpenCV,» [En línea]. Available: https://opencv.org/about/.

[21] «Documentos OpenCV,» [En línea]. Available: https://docs.opencv.org/4.5.2/d1/dfb/intro.html.

[22] «AprilTag,» [En línea]. Available: https://april.eecs.umich.edu/software/apriltag.

[23] X. Liang, G. Chen, S. Zhao y Y. Xiu, «Moving target tracking method for unmanned aerial vehicle/unmanned ground vehicle heterogeneous system based on AprilTags,» *Measurement and Control (United Kingdom),* 2020.

[24] B. Winkler, «Apriltag,» [En línea]. Available: https://pypi.org/project/apriltag/.

[25] «Visión estereoscópica,» [En línea]. Available: https://es.wikipedia.org/wiki/Visi%C3%B3n_estereosc%C3%B3pica.

[26] R. I. Hartley y P. Sturm, «Triangulation,» *Computer Vision and Image Understanding,* vol. 68, nº 2, pp. 146-157, 1997.

[27] «Camera Calibration-OpenCV,» [En línea]. Available: https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html.

[28] G. York y D. Pack, «Comparative study on time-varying target localization methods using multiple unmanned aerial vehicles: Kalman estimation and triangulation techniques,» *IEEE Networking, Sensing and Control, ICNSC2005 - Proceedings,* pp. 305-310, 2005.

[29] «OpenCV-Triangulation,» [En línea]. Available: https://docs.opencv.org/3.4/d0/dbd/group__triangulation.html.

[30] P. Wira y J. P. Urban, «A New Adaptive Kalman Filter Applied To Visual Servoing Tasks,» de *Knowledge-Based Intelligent Engineering Systems and Allied Technologies*, Brighton, UK, 2000.

[31] L. Meier, «https://mavlink.io/en/,» [En línea].

[32] A. Koubaa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith y M. Khalgui, «Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey,» *IEEE Access,* pp. 87658-87680, 2019.

[33] Gremsy. [En línea]. Available: https://github.com/Gremsy/gSDK.

[34] «Robot Operating System,» [En línea]. Available: http://wiki.ros.org/ROS/Introduction.

[35] «Robot Operating System,» [En línea]. Available: https://www.ros.org/.

[36] «Robot Operating System,» [En línea]. Available: http://wiki.ros.org/ROS/Concepts.

[37] «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Peer-to-peer.

[38] Flova y gChuNguyen. [En línea]. Available: https://github.com/Flova/ros_gremsy.

[39] «Robot Operating System,» [En línea]. Available: http://wiki.ros.org/usb_cam.

[40] «Robot Operating System,» [En línea]. Available: http://wiki.ros.org/rqt_graph.

[41] E. W. Weisstein, « "Euler Angles." From MathWorld--A Wolfram Web Resource.,» [En línea]. Available: https://mathworld.wolfram.com/EulerAngles.html.

[42] Auawise, «An image showing all three axes. Wikipedia.,» 2008. [En línea]. Available: https://commons.wikimedia.org/wiki/File:Yaw_Axis.svg.

[43] Cone83, «Inscription on Broom Bridge,» 2017. [En línea]. Available: commons.wikipedia.org.

[44] D. Malan, «Representation of how the axis and angle rotation representation can be visualized,» 2004. [En línea]. Available: https://commons.wikimedia.org/wiki/File:Euler_AxisAngle.png.

[45] Autor, «Este es el ejemplo de una cita,» *Tesis Doctoral,* vol. 2, nº 13, 2012.

[46] O. Autor, «Otra cita distinta,» *revista,* p. 12, 2001.

[47] D. Mery y F. Pedreschi, «Segmentation of colour food images using a robust algorithm,» *Journal of Food Engineering,* 2004.

[48] [En línea]. Available: https://opencv.org/resources/media-kit/.

[49] «OpenCV-Triangulation,» [En línea]. Available: https://docs.opencv.org/3.4/d0/dbd/group__triangulation.html.

[50] [En línea]. Available: https://www.laboratoriogluon.com/filtro-de-kalman-deduccion-ejemplos/.

[51] «Rostutorial,» [En línea]. Available: http://rostutorial.com/4-nodos-topics-y-mensajes-turtlesim/.

ANEXOS

Anexo A: Triangulación

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri May 14 12:01:23 2021

@author: nacho
"""
import cv2
import numpy as np
import apriltag

"""Matriz calculada en la calibración"""
mtx=np.array([[1.63704531e+03, 0, 9.49359532e+02],
 [0, 1.63864093e+03, 5.32636161e+02],
 [0, 0, 1]])
dist=np.array([0.01737745, -0.36542463, 0.01312702, -0.00285169,
  0.80215804])

"""Leemos imagen 1"""
image1 = cv2.imread('im0.png')
gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
detector1 = apriltag.Detector()
results1 = detector1.detect(gray1)
for r in results1:
    (cX1, cY1) = (int(r.center[0]), int(r.center[1]))
    x1=np.array([float(cX1), float(cY1)])
    cv2.circle(image1, (cX1, cY1), 5, (0, 0, 255), -1)
cv2.namedWindow('im1', cv2.WINDOW_NORMAL)
cv2.resizeWindow('im1', 800, 800)
cv2.imshow('im1', image1)
cv2.imwrite('imagen1tri.png', image1)
cv2.waitKey(0)

"""Creamos matriz mejorada"""
h,  w = image1.shape[:2]
newcameramtx, roi = cv2.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1, (w,h))
intrinsic=np.array(newcameramtx)

"""Parametros imagen 1"""
rot1=np.identity(3)
trans1=np.array([[0, 0, 0]])
extrinsic1=np.concatenate((rot1, trans1.T), axis=1)
```

```
mat1=np.matmul(intrinsic,extrinsic1)

"""Leemos imagen 2"""
image2 = cv2.imread('im1.png')
gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
detector2 = apriltag.Detector()
results2 = detector2.detect(gray2)
for r in results2:
    (cX2, cY2) = (int(r.center[0]), int(r.center[1]))
    x2=np.array([float(cX2), float(cY2)])
    cv2.circle(image2, (cX2, cY2), 5, (0, 0, 255), -1)
cv2.namedWindow('im2', cv2.WINDOW_NORMAL)
cv2.resizeWindow('im2', 800, 800)
cv2.imshow('im2', image2)
cv2.imwrite('imagen2tri.png', image2)
cv2.waitKey(0)

"""Parametros imagen 2"""
roll2=0
pitch2=3.14159*90/180
yaw2=0

rotx2=np.array([[1, 0, 0], [0, np.cos(roll2), -
np.sin(roll2)], [0, np.sin(roll2), np.cos(roll2)]])
roty2=np.array([[np.cos(pitch2), 0, np.sin(pitch2)], [0, 1, 0], [-
np.sin(pitch2), 0, np.cos(pitch2)]])
rotz2=np.array([[np.cos(yaw2), -
np.sin(yaw2), 0], [np.sin(yaw2), np.cos(yaw2), 0], [0, 0, 1]])
rot2=np.matmul(rotz2,roty2)
rot2=np.matmul(rot2,rotx2)
trans2=np.array([[-0.68, 0, 0.62]])
extrinsic2=np.concatenate((rot2, trans2.T), axis=1)
mat2=np.matmul(intrinsic,extrinsic2)

"""Leemos imagen 3"""
image3 = cv2.imread('im2.png')
gray3 = cv2.cvtColor(image3, cv2.COLOR_BGR2GRAY)
detector3 = apriltag.Detector()
results3 = detector3.detect(gray3)
for r in results3:
    (cX3, cY3) = (int(r.center[0]), int(r.center[1]))
    x3=np.array([float(cX3), float(cY3)])
    cv2.circle(image3, (cX3, cY3), 5, (0, 0, 255), -1)
cv2.namedWindow('im3', cv2.WINDOW_NORMAL)
cv2.resizeWindow('im3', 800, 800)
cv2.imshow('im3', image3)
cv2.imwrite('imagen3tri.png', image3)
cv2.waitKey(0)
```

```
"""Parametros imagen 3"""
roll31=0
pitch31=3.14159*45/180
yaw31=0

roll32=0
pitch32=-3.14159*45/180
yaw32=0

rotx31=np.array([[1, 0, 0], [0, np.cos(roll31), -
np.sin(roll31)], [0, np.sin(roll31), np.cos(roll31)]])
roty31=np.array([[np.cos(pitch31), 0, np.sin(pitch31)], [0, 1, 0], [-
np.sin(pitch31), 0, np.cos(pitch31)]])
rotz31=np.array([[np.cos(yaw31), -
np.sin(yaw31), 0], [np.sin(yaw31), np.cos(yaw31), 0], [0, 0, 1]])
rot31=np.matmul(rotz31,roty31)
rot31=np.matmul(rot31,rotx31)
trans31=np.array([[-0.5, 0, 0]])
extrinsic31=np.concatenate((rot31, trans31.T), axis=1)
mat31=np.matmul(intrinsic,extrinsic31)

rotx32=np.array([[1, 0, 0], [0, np.cos(roll32), -
np.sin(roll32)], [0, np.sin(roll32), np.cos(roll32)]])
roty32=np.array([[np.cos(pitch32), 0, np.sin(pitch32)], [0, 1, 0], [-
np.sin(pitch32), 0, np.cos(pitch32)]])
rotz32=np.array([[np.cos(yaw32), -
np.sin(yaw32), 0], [np.sin(yaw32), np.cos(yaw32), 0], [0, 0, 1]])
rot32=np.matmul(rotz32,roty32)
rot32=np.matmul(rot32,rotx32)
trans32=np.array([[0.62, 0, 0.18]])
extrinsic32=np.concatenate((rot32, trans32.T), axis=1)
mat32=np.matmul(intrinsic,extrinsic32)

"""Primer punto"""
p1 = cv2.triangulatePoints(mat1, mat2, x1.T, x2.T)
# Transformamos a coordenadas no homogeneas
p1/=p1[3]

"""Segundo punto"""
p2 = cv2.triangulatePoints(mat1, mat31, x1.T, x3.T)
p2/=p2[3]

"""Tercer punto"""
p3 = cv2.triangulatePoints(mat1, mat32, x2.T, x3.T)

p3/=p3[3]
# Transformamos punto al sistema origen
p3 = np.matmul(extrinsic2,p3)

print('Punto 1 calculado',  p1.T)
```

```
print('Punto 2 calculado',  p2.T)
print('Punto 3 calculado',  p3.T)

"""Calcular centroide del triángulo"""
xc1=float((p1[0]+p2[0])/2)
yc1=float((p1[1]+p2[1])/2)
zc1=float((p1[2]+p2[2])/2)

xc2=float((p1[0]+p3[0])/2)
yc2=float((p1[1]+p3[1])/2)
zc2=float((p1[2]+p3[2])/2)

v10=float(xc1-p3[0])
v11=float(yc1-p3[1])
v12=float(zc1-p3[2])

v20=float(xc2-p2[0])
v21=float(yc2-p2[1])
v22=float(zc2-p2[2])

A=np.array([[1, 0, 0, -v10, 0], [0, 1, 0, -v11, 0], [0, 0, 1, -
v12, 0], [1, 0, 0, 0, -v20], [0, 1, 0, 0, -v21]])
B=np.array([[xc1], [yc1], [zc1], [xc2], [yc2]])

x=np.linalg.solve(A,B)

p=x[0:3]

print('Projected point from openCV:',  p.T)
```

Anexo B: Nodo Control_gremsy

```
#include "ros/ros.h"
#include <std_msgs/Bool.h>
#include <std_msgs/Float32.h>
#include <std_msgs/Float32MultiArray.h>
#include <geometry_msgs/Quaternion.h>
#include <geometry_msgs/Vector3Stamped.h>
#include <sstream>
#include <sensor_msgs/Imu.h>
#include <unistd.h>
#include <cmath>
#include <iostream>
#include <stdlib.h>
#include <fstream>
#include <ctime>



using namespace std;

#define DEG_TO_RAD (M_PI / 180.0)
#define RAD_TO_DEG (180.0 / M_PI)

//Variables globales

ofstream archivo;
ros:: Publisher control_goal_pub;

//Variables a graficar
float desired_vector[3];
float current_vector[3]={0,0,0};
float qr[4];
float qe[4];
float omega_real[3]={0,0,0};
//Declaracion angulos en grados
float pitch_deg;
float yaw_deg;
float roll_deg;
float omega_e[3];

//Guardar valores subscritos
float qros[4]={1,0,0,0};
int16_t omega_act[3]={0,0,0};
int16_t omega_ant[3]={0,0,0};
int16_t omega_raw[3]={0,0,0};
float vector_goals[3]={0,0,0};
float pitch_act;
float yaw_act;
float roll_act;
```

```
float qant[4];

//Guardar parámetros de control obtenidos
//Declaración Parámetros
float Kp1=8;
float Kroll=0.0001;
float Kd1=0.0005;
float gamma1=25;
float Kp2=8;
float Kd2=0.0005;
float gamma2=15;


//Variables auxiliares
bool det;
int t;
int tiempo;


void publishStateTimerCallback(const ros::TimerEvent &event)
{
// Publish Gimbal goals
geometry_msgs::Vector3Stamped goals;

//Declaración Vectores y Velocidades
float origin_vector[3];
float theta_e[3];

//Declaración Cuaterniones
float qyaw[4]={1,0,0,0};
float quat_current_vector[4];
float quat_origin_vector[4]={0,1,0,0};
float qt[4];
float enorm;

//Declaración Auxiliares
int k=0;
int k_ant=0;
float rtb_Sqrt1;
float vecprod[3];
float dotprod;
float raizdotprod;
float Prod;
float Acos;
```

```
//Asignacion goals
desired_vector[0] = vector_goals[2];
desired_vector[1] = -vector_goals[0];
desired_vector[2] = -vector_goals[1];

//Vector origen(global frame)
origin_vector[0]=1;
origin_vector[1]=0;
origin_vector[2]=0;


//Normalizacion vector deseado
rtb_Sqrt1 = 1.0 / (sqrt((desired_vector[0] * desired_vector[0] + desired_vector[
1]*
desired_vector[1]) + desired_vector[2]  * desired_vector[2] ) + 1.0E-10);
desired_vector[0] = desired_vector[0] * rtb_Sqrt1;
desired_vector[1] = desired_vector[1] * rtb_Sqrt1;
desired_vector[2] = desired_vector[2] * rtb_Sqrt1;

//Normalizacion vector origen
rtb_Sqrt1 = 1.0 / (sqrt((origin_vector[0] * origin_vector[0] + origin_vector[1]
* origin_vector[1]) +
origin_vector[2] * origin_vector[2]) + 1.0E-10);
origin_vector[0] = origin_vector[0] * rtb_Sqrt1;
origin_vector[1] = origin_vector[1] * rtb_Sqrt1;
origin_vector[2] = origin_vector[2] * rtb_Sqrt1;

//Producto vectorial
vecprod[0] = (desired_vector[1] * origin_vector[2])-
(desired_vector[2] * origin_vector[1]);
vecprod[1]= (desired_vector[2] * origin_vector[0]) -
(desired_vector[0] * origin_vector[2]);
vecprod[2] = (desired_vector[0] * origin_vector[1])-
(desired_vector[1] * origin_vector[0]);

//Normalizacion producto vetorial
rtb_Sqrt1 = (1.0/(sqrt((vecprod[0]*vecprod[0] + vecprod[1]*vecprod[1]) + vecprod
[2]*vecprod[2]) + 1.0E-10));
vecprod[0] *= rtb_Sqrt1;
vecprod[1] *= rtb_Sqrt1;
vecprod[2] *= rtb_Sqrt1;

//Producto escalar
dotprod = (desired_vector[0]*origin_vector[0] + desired_vector[1]*origin_vector[
1]) + desired_vector[2]*origin_vector[2];

raizdotprod = sqrt((1.0 - dotprod) / 2.0);

//Obtencion cuaternio qt
qt[0] = sqrt((1.0 + dotprod) / 2.0);
```

```
qt[1] = raizdotprod * vecprod[0];
qt[2] = raizdotprod * vecprod[1];
qt[3] = raizdotprod * vecprod[2];

//Obtener qactual
qros[1] = (sin(roll_act/2) * cos(pitch_act/2) * cos(yaw_act/2)) -
 (cos(roll_act/2) * sin(pitch_act/2) * sin(yaw_act/2));
qros[2] = (cos(roll_act/2) * sin(pitch_act/2) * cos(yaw_act/2)) + (sin(roll_act/
2) * cos(pitch_act/2) * sin(yaw_act/2));
qros[3] = (cos(roll_act/2) * cos(pitch_act/2) * sin(yaw_act/2)) -
 (sin(roll_act/2) * sin(pitch_act/2) * cos(yaw_act/2));
qros[0] = (cos(roll_act/2) * cos(pitch_act/2) * cos(yaw_act/2)) + (sin(roll_act/
2) * sin(pitch_act/2) * sin(yaw_act/2));


//Obtener current vector

quat_current_vector[0]=(qros[0]*quat_origin_vector[0])-
(qros[1]*quat_origin_vector[1])-(qros[2]*quat_origin_vector[2])-
(qros[3]*quat_origin_vector[3]);
quat_current_vector[1]=(qros[0]*quat_origin_vector[1])+(qros[2]*quat_origin_vect
or[3])+(qros[1]*quat_origin_vector[0])-(qros[3]*quat_origin_vector[2]);
quat_current_vector[2]=(qros[0]*quat_origin_vector[2])+(qros[2]*quat_origin_vect
or[0])+(qros[3]*quat_origin_vector[1])-(qros[1]*quat_origin_vector[3]);
quat_current_vector[3]=(qros[0]*quat_origin_vector[3])+(qros[1]*quat_origin_vect
or[2])-(qros[2]*quat_origin_vector[1])+(qros[3]*quat_origin_vector[0]);

quat_current_vector[0]=(quat_origin_vector[0]*qros[0])-(quat_origin_vector[1]*-
qros[1])-(quat_origin_vector[2]*-qros[2])-(quat_origin_vector[3]*-qros[3]);
quat_current_vector[1]=(quat_origin_vector[0]*-qros[1])+(quat_origin_vector[2]*-
qros[3])+(quat_origin_vector[1]*qros[0])-(quat_origin_vector[3]*-qros[2]);
quat_current_vector[2]=(quat_origin_vector[0]*-
qros[2])+(quat_origin_vector[2]*qros[0])+(quat_origin_vector[3]*-qros[1])-
(quat_origin_vector[1]*-qros[3]);
quat_current_vector[3]=(quat_origin_vector[0]*-qros[3])+(quat_origin_vector[1]*-
qros[2])-(quat_origin_vector[2]*-qros[1])+(quat_origin_vector[3]*qros[0]);

current_vector[0]=quat_current_vector[1];
current_vector[1]=quat_current_vector[2];
current_vector[2]=quat_current_vector[3];



if (det==true){
//Obtenemos qreferencia
qr[0]=(qros[0]*qt[0])-(qros[1]*qt[1])-(qros[2]*qt[2])-(qros[3]*qt[3]);
qr[1]=(qros[0]*qt[1])+(qros[2]*qt[3])+(qros[1]*qt[0])-(qros[3]*qt[2]);
qr[2]=(qros[0]*qt[2])+(qros[2]*qt[0])+(qros[3]*qt[1])-(qros[1]*qt[3]);
qr[3]=(qros[0]*qt[3])+(qros[1]*qt[2])-(qros[2]*qt[1])+(qros[3]*qt[0]);
```

```
k=1;

}
else{
k=0;
qr[0]=qt[0];
qr[1]=qt[1];
qr[2]=qt[2];
qr[3]=qt[3];
}


//Obtener qe
qe[0]=(qr[0]*qros[0])-(-qr[1]*qros[1])-(-qr[2]*qros[2])-(-qr[3]*qros[3]);
qe[1]=(qr[0]*qros[1])+(-qr[2]*qros[3])+(-qr[1]*qros[0])-(-qr[3]*qros[2]);
qe[2]=(qr[0]*qros[2])+(-qr[2]*qros[0])+(-qr[3]*qros[1])-(-qr[1]*qros[3]);
qe[3]=(qr[0]*qros[3])+(-qr[1]*qros[2])-(-qr[2]*qros[1])+(-qr[3]*qros[0]);

Acos = acos(qe[0]);

enorm = sqrt(qe[1]*qe[1] + qe[2]*qe[2] + qe[3]*qe[3]);

//Multiplicamos el vector componente por componente
if (enorm > 0.0001){
  Prod = Acos*qe[1]*2.0/enorm;
  theta_e[0] = Prod;
  Prod = Acos*qe[2]*2.0/enorm;
  theta_e[1] = Prod;
  Prod = Acos*qe[3]*2.0/enorm;
  theta_e[2] = Prod;
}
else{
  theta_e[0] = 0;
  theta_e[1] = 0;
  theta_e[2] = 0;
}
//Multiplicamos el vector componente por componente
Prod = Acos*qe[1]*2.0;
theta_e[0] = Prod*180/3.1415;
Prod = Acos*qe[2]*2.0;
theta_e[1] = Prod*180/3.1415;
Prod = Acos*qe[3]*2.0;
theta_e[2] = Prod*180/3.1415;

//Obtenemos la omega
//Leer omegas actuales
//Filtro paso bajo
float beta=0.1;
omega_act[0]=(beta*omega_act[0]+(1-beta)*omega_ant[0]);
omega_act[1]=(beta*omega_act[1]+(1-beta)*omega_ant[1]);
```

```
omega_act[2]=(beta*omega_act[2]+(1-beta)*omega_ant[2]);

omega_ant[0]=omega_act[0];
omega_ant[1]=omega_act[1];
omega_ant[2]=omega_act[2];

//Calculo formula omega
float term1[3];
term1[0] = Kroll * theta_e[0];
term1[1] = Kp2 * theta_e[1];
term1[2] = Kp1 * theta_e[2];

float term2[3];
term2[0] = Kd1 * omega_act[0];
term2[1] = Kd2 * omega_act[1];
term2[2] = Kd1 * omega_act[2];

term2[0] = term2[0]+term1[0];
term2[1] = term2[1]+term1[1];
term2[2] = term2[2]+term1[2];

//Normalizacion term2
float term2_norm;
term2_norm = (sqrt((term2[0] * term2[0] + term2[1] * term2[1]) +
term2[2] * term2[2]) + 1.0E-10);

//Calculo omegas
float k1;
float k2;
k1=(-gamma1*tanh(term2_norm/gamma1))/term2_norm;
k2=(-gamma2*tanh(term2_norm/gamma2))/term2_norm;
omega_e[0]=k2*term2[0];
omega_e[1]=k2*term2[1];
omega_e[2]=k1*term2[2];

float setpoint_pitch = DEG_TO_RAD*omega_e[1];
float setpoint_roll  = DEG_TO_RAD*omega_e[0];
float setpoint_yaw   = DEG_TO_RAD*omega_e[2];

//Establezco y publico goals
goals.vector.x = setpoint_roll;
goals.vector.y = setpoint_pitch;
goals.vector.z = setpoint_yaw;

control_goal_pub.publish(goals);

//Defino ángulos
pitch_deg=RAD_TO_DEG*pitch_act;
roll_deg=RAD_TO_DEG*roll_act;
```

```
yaw_deg=RAD_TO_DEG*yaw_act;


//Salida por pantalla
t++;
if(t%60000==0){
printf("Ejecutando...\n");
}

}

void getGoalsCallback(geometry_msgs::Vector3Stamped message_goals)
{
  vector_goals[0] = message_goals.vector.x;
  vector_goals[1] = message_goals.vector.y;
  vector_goals[2] = message_goals.vector.z;
}
void getImuCallback(sensor_msgs::Imu message_imu)
{
  omega_act[0] = message_imu.angular_velocity.y;
  omega_act[1] = -message_imu.angular_velocity.x;
  omega_act[2] = message_imu.angular_velocity.z;
  omega_real[0] = message_imu.angular_velocity.y;
  omega_real[1] = -message_imu.angular_velocity.x;
  omega_real[2] = message_imu.angular_velocity.z;
}

void getOrientationCallback(geometry_msgs::Quaternion message_orientation)
{
  qros[0] = message_orientation.w;
  qros[1] = message_orientation.x;
  qros[2] = -message_orientation.y;
  qros[3] = message_orientation.z;
}

void getDetCallback(std_msgs::Bool message_deteccion)
{
  det = message_deteccion.data;
}
void getAnglesCallback(geometry_msgs::Vector3Stamped message_angles)
{
  roll_act = message_angles.vector.x;
  pitch_act = message_angles.vector.y;
  yaw_act = message_angles.vector.z;
}
void getParametrosCallback(std_msgs::Float32MultiArray message_parametros)
{
  Kp1 = message_parametros.data[0];
  Kp2 = message_parametros.data[1];
  Kd1 = message_parametros.data[2];
```

```
  Kd2 = message_parametros.data[3];
  gamma1 = message_parametros.data[4];
  gamma2 = message_parametros.data[5];
}

void getSincroCallback(std_msgs::Bool message_sincro)
{
archivo << qr[0] << endl;
archivo << qr[1] << endl;
archivo << qr[2] << endl;
archivo << qr[3] << endl;
archivo << qe[0] << endl;
archivo << qe[1] << endl;
archivo << qe[2] << endl;
archivo << qe[3] << endl;
archivo << qros[0] << endl;
archivo << qros[1] << endl;
archivo << qros[2] << endl;
archivo << qros[3] << endl;
archivo << omega_e[0] << endl;
archivo << omega_e[1] << endl;
archivo << omega_e[2] << endl;
archivo << pitch_deg << endl;
archivo << roll_deg << endl;
archivo << yaw_deg << endl;
archivo << omega_act[0] << endl;
archivo << omega_act[1] << endl;
archivo << omega_act[2] << endl;
archivo << desired_vector[0] << endl;
archivo << desired_vector[1] << endl;
archivo << desired_vector[2] << endl;
archivo << current_vector[0] << endl;
archivo << current_vector[1] << endl;
archivo << current_vector[2] << endl;
archivo << omega_real[0] << endl;
archivo << omega_real[1] << endl;
archivo << omega_real[2] << endl;
}

int main(int argc, char **argv)
{
sleep(1);
archivo.open("/home/xavier1/Documentos/datoscontrol.txt",ios::out);
    if(archivo.fail()){
        cout<<"No se pudo abrir el archivo";
        exit(1);
    }
ros::init(argc, argv, "control_gremsy");
ros::NodeHandle pnh;
```

```
ros::NodeHandle nh;


ros:: Subscriber control_imu_sub = pnh.subscribe("ros_gremsy/imu/data", 1, getIm
uCallback);
ros:: Subscriber control_orientation_sub = pnh.subscribe("ros_gremsy/mount_orien
tation_local_yaw", 1, getOrientationCallback);
ros:: Subscriber control_vector_sub = pnh.subscribe("vector_goals", 1, getGoalsC
allback);
ros:: Subscriber deteccion_sub = pnh.subscribe("deteccion", 1, getDetCallback);
ros:: Subscriber angles_sub = pnh.subscribe("ros_gremsy/angulos_actual", 1, getA
nglesCallback);
ros:: Subscriber param_sub = pnh.subscribe("parametros",1, getParametrosCallback
);
ros:: Subscriber sincro_sub = pnh.subscribe("sincro",1, getSincroCallback);

control_goal_pub = pnh.advertise <geometry_msgs::Vector3Stamped>("ros_gremsy/goa
ls", 10);

ros::Timer timer =
       nh.createTimer(ros::Duration(1 / 50), publishStateTimerCallback);

ros::spin();

archivo.close();
return 0;
```

Anexo C: Nodo Procesamiento

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed May 26 09:32:04 2021
CODIGO PARA PROBAR PYTHON Y OPENCV EN ROS
@author: nacho
"""
#Python libraries
import sys, time
import numpy as np
import apriltag
#Open CV

import cv2

#ROS libraries
import rospy

#ROS messages
from gimbalpack.msg import Vector
from geometry_msgs.msg import Vector3Stamped
from std_msgs.msg import Bool
from sensor_msgs.msg import Image

#VARIABLES GLOBALES
k=0
img_counter=0
det=1
distancia=1
GSD=0.00061*distancia
X0=1920/2
Y0=1080/2
cX=X0-1
cY=Y0-1
x_hat=np.matrix([0.01, -0.01, 1])
P_ant=0.5*np.identity(3)
x_hat=x_hat.T
x_ant=x_hat
y_ant=x_ant
x_vector=[]
P_vector=[]
x_corr=x_hat
P_corr=P_ant
K=x_hat
y=y_ant
x_hat_vector=[]
x_med=[]
```

```
K_vector=[]
x_vector=[]
P_vector=[]
A_vector=[]
translation_matrix=[]
search=0
detector = apriltag.Detector()
ultrefx=0
ultrefy=0
A_search=[]
flag_busqueda=0
A=[]

def talker(goal,det):
    pub = rospy.Publisher("vector_goals", Vector3Stamped, queue_size=10)
    pub2 = rospy.Publisher("deteccion", Bool, queue_size=10)
    rospy.loginfo('publicando')
    rospy.loginfo(goal)
    pub.publish(goal)
    pub2.publish(det)

def callback(imagen):

    global detector
    global flag_busqueda
    global ultrefx
    global ultrefy
    global search
    global A_search
    global x_hat
    global x_ant
    global y_ant
    global y
    global P_corr
    global x_corr
    global x_hat
    global K
    global A
    global P_ant

    VERBOSE=False
    goal=Vector3Stamped()
    dtype = np.dtype("uint8")
    dtype = dtype.newbyteorder('>' if imagen.is_bigendian else '<')
    imagencv = np.ndarray(shape=(imagen.height, imagen.width, 3),
                    dtype=dtype, buffer=imagen.data)

    imagencv = cv2.cvtColor(imagencv, cv2.COLOR_RGB2BGR)
    gray = cv2.cvtColor(imagencv, cv2.COLOR_BGR2GRAY)
    results = detector.detect(gray)
```

```
    # Bucle con los resultados de apriltags
    if not results:
       print("No hay resultado")
       xb=ultrefx
       yb=ultrefy
       print('ultrefx:')
       print(ultrefx)
       print('ultrefy:')
       print(ultrefy)

       if (search==1):
           flag_busqueda=1
           det=1
           if(xb>530):
               goal.vector.x=530*GSD

           if(yb>300):
               goal.vector.y=300*GSD

           if(xb<-530):
               goal.vector.x=-530*GSD

           if(yb<-300):
               goal.vector.y=-300*GSD

           if(xb<530.0 and xb>-530.0 and yb<300.0 and yb>-
300.0): #Si estamos dentro del cuadrado, voy a asumir que es una perturbacion, m
e quedo como estoy
               goal.vector.x=float(0)
               goal.vector.y=float(0)


           goal.vector.z=float(1)
           print('Buscando...')
           talker(goal,det)
       else:
           det=0
           goal.vector.x=float(0)
           goal.vector.y=float(0)
           goal.vector.z=float(1)
       print('NO BUSCANDO')
       y=([0, 0, 1])
       talker(goal,det)


    else:
        for r in results:
            (cX, cY) = (int(r.center[0]), int(r.center[1]))
```

```
        cv2.circle(imagencv, (cX, cY), 5, (0, 0, 255), -1)

    det=1

    X=(cX-X0)*GSD
    Y=(cY-Y0)*GSD
    Z=distancia

    ultrefx=X/GSD
    ultrefy=Y/GSD

    y = np.matrix([X, Y, Z])
    y = y.T

    if (flag_busqueda==1):
        y_ant=y
        x_ant=y
        flag_busqueda=0

    A = y_ant * y_ant.T
    P, D, Q = np.linalg.svd(A, full_matrices=False)
    A = P @ np.diag(D) @ Q
    A = (y.T * A).T
    A = A * y_ant.T

    Q=np.matrix([[0.005, 0, 0], [0, 0.005, 0], [0, 0, 0.005]])
    R=Q

    x_hat = A * x_ant


    P = (A * P_ant * A.T) + Q


    K = P * ((P+R)**-1)


    x_corr = x_hat + K*(y-x_hat)


    P_corr = (np.identity(3)-K) * P


    x_ant=x_corr
    y_ant=y
    P_ant=P_corr

    goal.vector.x=float(x_corr[0])
    goal.vector.y=float(x_corr[1])
    goal.vector.z=float(1)
```

```
        talker(goal,det)
        search=1


def listener():

    rospy.init_node('Procesamiento')

    rospy.Subscriber("/UAV_1/usb_cam/image_raw", Image, callback)

    rospy.Subscriber("ros_gremsy/angulos_actual", Vector3Stamped, callback2)

    rospy.Subscriber("sincro", Bool, callback3)

    rospy.spin()

    with open('/home/nacho/pruebas/outfile1.txt','wb') as f:
       for line in x_vector:
           np.savetxt(f, line, fmt='%.2f')

    with open('/home/nacho/pruebas/outfile2.txt','wb') as f:
       for line in P_vector:
           np.savetxt(f, line, fmt='%.5f')

    with open('/home/nacho/pruebas/outfile3.txt','wb') as f:
       for line in x_hat_vector:
           np.savetxt(f, line, fmt='%.2f')

    with open('/home/nacho/pruebas/outfile4.txt','wb') as f:
       for line in K_vector:
           np.savetxt(f, line, fmt='%.2f')

    with open('/home/nacho/pruebas/outfile5.txt','wb') as f:
       for line in x_med:
           np.savetxt(f, line, fmt='%.2f')

    with open('/home/nacho/pruebas/outfile6.txt','wb') as f:
       for line in A_vector:
           np.savetxt(f, line, fmt='%.2f')

def callback2(mensaje):
   global search
   if ((mensaje.vector.y*180/3.1415)>55.0 or (mensaje.vector.y*180/3.1415)<-
55.0 or (mensaje.vector.z*180/3.1415)>100.0 or (mensaje.vector.z*180/3.1415)<-
100.0):
       search=0
       print(mensaje.vector.y*180/3.1415)
       print(mensaje.vector.z*180/3.1415)
       print(search)
```

```
def callback3(mensaje):

        global x_hat
        global x_ant
        global y_ant
        global P_ant
        global x_vector
        global P_vector
        global x_hat_vector
        global K_vector
        global K
        global x_corr
        global y
        global P_corr
        global A
        global A_vector

        x_vector.append(x_corr)
        P_vector.append(P_corr)
        x_hat_vector.append(x_hat)
        K_vector.append(K)
        x_med.append(y)
        A_vector.append(A)
        rospy.loginfo('graficando')

if __name__ == '__main__':

    listener()
```

Anexo D: Nodo Grabación

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed May 26 09:50:04 2021
PRUEBA PYTHON OPEN CV Y ROS
@author: nacho
"""
#Python libraries
import sys, time
import numpy as np
import apriltag

#Open CV
import cv2

#ROS libraries
import roslib
import rospy

#ROS messages
from gimbalpack.msg import Vector
from geometry_msgs.msg import Vector3
from std_msgs.msg import Float32MultiArray
from sensor_msgs.msg import Image
salida = cv2.VideoWriter('/home/nacho/pruebas/VideoSalida.avi',cv2.VideoWriter_f
ourcc(*'XVID'),7.0,(1920,1080))
detector = apriltag.Detector()

def callback(imagen):
    dtype = np.dtype("uint8")
    dtype = dtype.newbyteorder('>' if imagen.is_bigendian else '<')
    imagencv = np.ndarray(shape=(imagen.height, imagen.width, 3),
                        dtype=dtype, buffer=imagen.data)
    print('recibiendo imagen')
    imagencv = cv2.cvtColor(imagencv, cv2.COLOR_RGB2BGR)
    gray = cv2.cvtColor(imagencv, cv2.COLOR_BGR2GRAY)
    results = detector.detect(gray)
    for r in results:
            (cX, cY) = (int(r.center[0]), int(r.center[1]))
            cv2.circle(imagencv, (cX, cY), 5, (0, 0, 255), -1)

    salida.write(imagencv)

def listener():


    rospy.init_node('Grabacion')
```

```
    rospy.Subscriber("/UAV_1/usb_cam/image_raw", Image, callback)

    rospy.spin()

if __name__ == '__main__':
    listener()
```

Anexo E: Nodo Sincronización

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Jul  5 11:11:17 2021

@author: nacho
"""
import rospy
from std_msgs.msg import Bool

k=Bool()

def sincro():
        pub = rospy.Publisher("sincro", Bool, queue_size=10)
        rospy.init_node('Sincronizacion')
        k=True
        rate=rospy.Rate(50)
        while not rospy.is_shutdown():
           rospy.loginfo('sincronizando graficas')
           pub.publish(k)
           rate.sleep()


if __name__ == '__main__':
    try:
        sincro()
    except rospy.ROSInterruptException:
        pass
```

Anexo F: Nodo Parámetros

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 10 13:45:43 2021
INTERFAZ ROS PARA PASAR LOS PARAMETROS DE CONTROL POR TECLADO EN TIEMPO REAL
@author: nacho
"""
import rospy
from tkinter import *
from tkinter import ttk
from std_msgs.msg import Float32MultiArray

k=Float32MultiArray()

def actualizar(*args):
    try:
        pub = rospy.Publisher("parametros", Float32MultiArray, queue_size=10)
        rospy.init_node('Parametros_Control')
        k.data.append((float(kp1.get())))
        k.data.append((float(kp2.get())))
        k.data.append((float(kd1.get())))
        k.data.append((float(kd2.get())))
        k.data.append((float(gamma1.get())))
        k.data.append((float(gamma2.get())))
        pub.publish(k)
        k.data=[]
    except ValueError:
        pass

root = Tk()
root.title("Parametros")

s=ttk.Style()
s.theme_use('alt')

mainframe = ttk.Frame(root, padding="10 10 12 12")
mainframe.grid(column=10, row=10, sticky=(N, W, E, S))
root.columnconfigure(0, weight=1)
root.rowconfigure(0, weight=1)

kp1 = StringVar()
kp1_entry = ttk.Entry(mainframe, width=7, textvariable=kp1)
kp1_entry.grid(column=2, row=1, sticky=(W, E))
ttk.Label(mainframe, text="kp1 (yaw)").grid(column=1, row=1, sticky=W)

kp2 = StringVar()
kp2_entry = ttk.Entry(mainframe, width=7, textvariable=kp2)
kp2_entry.grid(column=2, row=2, sticky=(W, E))
```

```
ttk.Label(mainframe, text="kp2 (roll & pitch)").grid(column=1, row=2, sticky=W)

kd1 = StringVar()
kd1_entry = ttk.Entry(mainframe, width=7, textvariable=kd1)
kd1_entry.grid(column=2, row=3, sticky=(W, E))
ttk.Label(mainframe, text="kd1 (yaw)").grid(column=1, row=3, sticky=W)

kd2 = StringVar()
kd2_entry = ttk.Entry(mainframe, width=7, textvariable=kd2)
kd2_entry.grid(column=2, row=4, sticky=(W, E))
ttk.Label(mainframe, text="kd2 (roll & pitch)").grid(column=1, row=4, sticky=W)

gamma1 = StringVar()
gamma1_entry = ttk.Entry(mainframe, width=7, textvariable=gamma1)
gamma1_entry.grid(column=2, row=5, sticky=(W, E))
ttk.Label(mainframe, text="gamma1 (yaw)").grid(column=1, row=5, sticky=W)

gamma2 = StringVar()
gamma2_entry = ttk.Entry(mainframe, width=7, textvariable=gamma2)
gamma2_entry.grid(column=2, row=6, sticky=(W, E))
ttk.Label(mainframe, text="gamma2 (roll & pitch)").grid(column=1, row=6, sticky=
W)



btn=ttk.Button(mainframe, text="Actualizar", command=actualizar).grid(column=7,
row=1, sticky=(W,E))

for child in mainframe.winfo_children():
    child.grid_configure(padx=10, pady=10)

root.bind("<Return>", actualizar)

root.mainloop()
```

Anexo G: Nodo Control_gremsy. Control en posición

```
#include "ros/ros.h"
#include <std_msgs/Bool.h>
#include <std_msgs/Float32.h>
#include <std_msgs/Float32MultiArray.h>
#include <geometry_msgs/Quaternion.h>
#include <geometry_msgs/Vector3Stamped.h>
#include <geometry_msgs/PointStamped.h>
#include <sstream>
#include <sensor_msgs/Imu.h>
#include <unistd.h>
#include <cmath>
#include <iostream>
#include <stdlib.h>
#include <fstream>
#include <ctime>


using namespace std;

#define DEG_TO_RAD (M_PI / 180.0)
#define RAD_TO_DEG (180.0 / M_PI)

//Variables globales

ofstream archivo;
ros:: Publisher control_goal_pub;

ros:: Timer *puntero_timer;

//Variables a graficar
float desired_vector[3];
float current_vector[3]={0,0,0};
float encoder[3]={0,0,0};
float qr[4];
float qe[4];
float omega_real[3]={0,0,0};
//Declaracion angulos en grados
float pitch_deg;
float yaw_deg;
float roll_deg;
float omega_e[3];

//Guardar valores subscritos
float theta_ant[3]={0,0,0};
float theta_e[3]={0,0,0};
int16_t omega_raw[3]={0,0,0};
```

```
float vector_goals[3]={0,0,0};
float pitch_act;
float yaw_act;
float roll_act;
float qant[4];
float qros[4];

//Variables auxiliares
bool det=0;
int t;
int tiempo;



void publishStateTimerCallback(const ros::TimerEvent &event)
{
// Publish Gimbal goals
geometry_msgs::Vector3Stamped goals;
//Declaración Vectores y Velocidades

float origin_vector[3];


//Declaración Cuaterniones
float qyaw[4]={1,0,0,0};
float quat_current_vector[4];
float quat_origin_vector[4]={0,1,0,0};
float qt[4];


//Declaración Auxiliares
int k=0;
int k_ant=0;
float rtb_Sqrt1;
float vecprod[3];
float dotprod;
float raizdotprod;
float Prod;
float Acos;
float enorm;

float setpoint_pitch =  0;
float setpoint_roll  =  0;
float setpoint_yaw   =  0;

//Asignacion goals
desired_vector[0] = vector_goals[2];
desired_vector[1] = -vector_goals[0];
desired_vector[2] = -vector_goals[1];
```

```
//Vector origen(global frame)
origin_vector[0]=1;
origin_vector[1]=0;
origin_vector[2]=0;

//Normalizacion vector deseado
rtb_Sqrt1 = 1.0 / (sqrt((desired_vector[0] * desired_vector[0] + desired_vector[
1]*
desired_vector[1]) + desired_vector[2]  * desired_vector[2] ) + 1.0E-10);
desired_vector[0] = desired_vector[0] * rtb_Sqrt1;
desired_vector[1] = desired_vector[1] * rtb_Sqrt1;
desired_vector[2] = desired_vector[2] * rtb_Sqrt1;

//Normalizacion vector origen
rtb_Sqrt1 = 1.0 / (sqrt((origin_vector[0] * origin_vector[0] + origin_vector[1]
* origin_vector[1]) +
origin_vector[2] * origin_vector[2]) + 1.0E-10);
origin_vector[0] = origin_vector[0] * rtb_Sqrt1;
origin_vector[1] = origin_vector[1] * rtb_Sqrt1;
origin_vector[2] = origin_vector[2] * rtb_Sqrt1;

//Producto vectorial
vecprod[0] = (desired_vector[1] * origin_vector[2])-
(desired_vector[2] * origin_vector[1]);
vecprod[1]= (desired_vector[2] * origin_vector[0]) -
(desired_vector[0] * origin_vector[2]);
vecprod[2] = (desired_vector[0] * origin_vector[1])-
(desired_vector[1] * origin_vector[0]);

//Normalizacion producto vetorial
rtb_Sqrt1 = (1.0/(sqrt((vecprod[0]*vecprod[0] + vecprod[1]*vecprod[1]) + vecprod
[2]*vecprod[2]) + 1.0E-10));
vecprod[0] *= rtb_Sqrt1;
vecprod[1] *= rtb_Sqrt1;
vecprod[2] *= rtb_Sqrt1;


//Producto escalar
dotprod = (desired_vector[0]*origin_vector[0] + desired_vector[1]*origin_vector[
1]) + desired_vector[2]*origin_vector[2];

raizdotprod = sqrt((1.0 - dotprod) / 2.0);

//Obtencion cuaternio qt
qt[0] = sqrt((1.0 + dotprod) / 2.0);
qt[1] = raizdotprod * vecprod[0];
qt[2] = raizdotprod * vecprod[1];
qt[3] = raizdotprod * vecprod[2];
```

```
//Obtener current vector

quat_current_vector[0]=(qros[0]*quat_origin_vector[0])-
(qros[1]*quat_origin_vector[1])-(qros[2]*quat_origin_vector[2])-
(qros[3]*quat_origin_vector[3]);
quat_current_vector[1]=(qros[0]*quat_origin_vector[1])+(qros[2]*quat_origin_vect
or[3])+(qros[1]*quat_origin_vector[0])-(qros[3]*quat_origin_vector[2]);
quat_current_vector[2]=(qros[0]*quat_origin_vector[2])+(qros[2]*quat_origin_vect
or[0])+(qros[3]*quat_origin_vector[1])-(qros[1]*quat_origin_vector[3]);
quat_current_vector[3]=(qros[0]*quat_origin_vector[3])+(qros[1]*quat_origin_vect
or[2])-(qros[2]*quat_origin_vector[1])+(qros[3]*quat_origin_vector[0]);

quat_current_vector[0]=(quat_origin_vector[0]*qros[0])-(quat_origin_vector[1]*-
qros[1])-(quat_origin_vector[2]*-qros[2])-(quat_origin_vector[3]*-qros[3]);
quat_current_vector[1]=(quat_origin_vector[0]*-qros[1])+(quat_origin_vector[2]*-
qros[3])+(quat_origin_vector[1]*qros[0])-(quat_origin_vector[3]*-qros[2]);
quat_current_vector[2]=(quat_origin_vector[0]*-
qros[2])+(quat_origin_vector[2]*qros[0])+(quat_origin_vector[3]*-qros[1])-
(quat_origin_vector[1]*-qros[3]);
quat_current_vector[3]=(quat_origin_vector[0]*-qros[3])+(quat_origin_vector[1]*-
qros[2])-(quat_origin_vector[2]*-qros[1])+(quat_origin_vector[3]*qros[0]);

current_vector[0]=quat_current_vector[1];
current_vector[1]=quat_current_vector[2];
current_vector[2]=quat_current_vector[3];



//Obtenemos qreferencia

if (det==1)
{


qe[0]=(qros[0]*qt[0])-(qros[1]*qt[1])-(qros[2]*qt[2])-(qros[3]*qt[3]);
qe[1]=(qros[0]*qt[1])+(qros[2]*qt[3])+(qros[1]*qt[0])-(qros[3]*qt[2]);
qe[2]=(qros[0]*qt[2])+(qros[2]*qt[0])+(qros[3]*qt[1])-(qros[1]*qt[3]);
qe[3]=(qros[0]*qt[3])+(qros[1]*qt[2])-(qros[2]*qt[1])+(qros[3]*qt[0]);


Acos = acos(qe[0]);
enorm = sqrt(qe[1]*qe[1] + qe[2]*qe[2] + qe[3]*qe[3]);

//Multiplicamos el vector componente por componente
if (enorm > 0.0001){
  Prod = Acos*qe[1]*2.0/enorm;
  theta_e[0] = Prod;
  Prod = Acos*qe[2]*2.0/enorm;
  theta_e[1] = Prod;
```

```
  Prod = Acos*qe[3]*2.0/enorm;
  theta_e[2] = Prod;
}
else{
  theta_e[0] = 0;
  theta_e[1] = 0;
  theta_e[2] = 0;
}

setpoint_pitch =  theta_e[1];
setpoint_roll  =  0;
setpoint_yaw   =  theta_e[2];

}

else
{
setpoint_pitch =  0;
setpoint_roll  =  0;
setpoint_yaw   =  0;
}


//Establezco y publico goals
goals.vector.x = setpoint_roll;
goals.vector.y = setpoint_pitch;
goals.vector.z = setpoint_yaw;


control_goal_pub.publish(goals);


//Salida por pantalla
t++;
if(t%60000==0){
printf("Ejecutando...\n");
}

}

void changeDet(const ros::TimerEvent &event) {
  det=0;
  printf("salta timer");
}

void getAnglesCallback(geometry_msgs::Vector3Stamped message_angles)
{
  roll_act = message_angles.vector.x;
  pitch_act = message_angles.vector.y;
  yaw_act = message_angles.vector.z;
}
```

```
void getGoalsCallback(geometry_msgs::Vector3Stamped message_goals)
{
  vector_goals[0] = message_goals.vector.x;
  vector_goals[1] = message_goals.vector.y;
  vector_goals[2] = message_goals.vector.z;
  det=1;
  puntero_timer->stop();
  puntero_timer->start();
}

void getImuCallback(sensor_msgs::Imu message_imu)
{
  omega_real[0] = message_imu.angular_velocity.y;
  omega_real[1] = -message_imu.angular_velocity.x;
  omega_real[2] = message_imu.angular_velocity.z;
}

void getOrientationCallback(geometry_msgs::Quaternion message_orientation)
{
  qros[0] = message_orientation.w;
  qros[1] = message_orientation.x;
  qros[2] = -message_orientation.y;
  qros[3] = message_orientation.z;
}


void getSincroCallback(std_msgs::Bool message_sincro)
{
archivo << qr[0] << endl;
archivo << qr[1] << endl;
archivo << qr[2] << endl;
archivo << qr[3] << endl;
archivo << qe[0] << endl;
archivo << qe[1] << endl;
archivo << qe[2] << endl;
archivo << qe[3] << endl;
archivo << qros[0] << endl;
archivo << qros[1] << endl;
archivo << qros[2] << endl;
archivo << qros[3] << endl;
archivo << theta_e[0] << endl;
archivo << theta_e[1] << endl;
archivo << theta_e[2] << endl;
archivo << desired_vector[0] << endl;
archivo << desired_vector[1] << endl;
archivo << desired_vector[2] << endl;
archivo << current_vector[0] << endl;
archivo << current_vector[1] << endl;
```

```
archivo << current_vector[2] << endl;
archivo << omega_real[0] << endl;
archivo << omega_real[1] << endl;
archivo << omega_real[2] << endl;
archivo << pitch_act << endl;
archivo << roll_act << endl;
archivo << yaw_act << endl;
printf("comprobando rate");
}

int main(int argc, char **argv)
{

archivo.open("/home/xavier1/Documentos/datoscontrol.txt",ios::out);
    if(archivo.fail()){
        cout<<"No se pudo abrir el archivo";
        exit(1);
    }
ros::init(argc, argv, "control_gremsy");
ros::NodeHandle pnh;
ros::NodeHandle nh;


ros:: Subscriber control_orientation_sub = pnh.subscribe("ros_gremsy/mount_orien
tation_local_yaw", 1, getOrientationCallback);
ros:: Subscriber control_vector_sub = pnh.subscribe("vector_goals", 1, getGoalsC
allback);
ros:: Subscriber angles_sub = pnh.subscribe("ros_gremsy/angulos_actual", 1, getA
nglesCallback);
ros:: Subscriber sincro_sub = pnh.subscribe("sincro",1, getSincroCallback);

control_goal_pub = pnh.advertise <geometry_msgs::Vector3Stamped>("ros_gremsy/goa
ls", 10);

ros::Timer timer =
       nh.createTimer(ros::Duration(1 / 100), publishStateTimerCallback);

ros::Timer timer_det =
       nh.createTimer(ros::Duration(0.5), changeDet);

puntero_timer = &timer_det;


ros::spin();

archivo.close();
return 0;
}
```