Trabajo Fin de Grado

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Modelado en CoppeliaSim de Sistema Aéreo Autónomo para el Seguimiento Visual de Múltiples Objetivos

Autor: Francisco Javier Román Cortés

Tutor: Manuel Vargas Villanueva

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

Sevilla, 2023

Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Modelado en CoppeliaSim de Sistema Aéreo Autónomo para el Seguimiento Visual de Múltiples Objetivos

Autor:

Francisco Javier Román Cortés

Tutor:

Manuel Vargas Villanueva

Profesor titular de Universidad

Dpto. de Ingenería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2023

Trabajo de Fin de Grado: Modelado en CoppeliaSim de Sistema Aéreo Autónomo para el Seguimiento Visual de Múltiples Objetivos

Autor: Francisco Javier Román Cortés

Tutor: Manuel Vargas Villanueva

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

A mis padres

A mis compañeros

A mis profesores

Agradecimientos

Con este trabajo termina una intensa e importante etapa en mi vida, la cual me ha permitido evolucionar personalmente, tanto en cuanto a habilidades y conocimientos como en términos de madurez personal. Es conveniente mencionar a aquellas personas que me han acompañado durante este proceso.

Indudablemente, agradecer el apoyo de mi familia, especialmente el apoyo incondicional de mis padres que me han permitido superar diversos obstáculos que he ido encontrando durante esta etapa.

También mencionar a mis amigos, especialmente a los nuevos conocidos durante estos años, los cuales me han hecho creer en las cualidades del trabajo en grupo y me han enseñado que la ingeniería no debe ser un mundo aíslado sino colaborativo y donde, aunque pueda parecer obvio, es importante apoyarse en los conocimientos de los demás para sacar lo mejor de un proyecto.

Por otra parte, agradecer ampliamente al equipo docente, el cual no se valora suficientemente, la calidad de los conocimientos transmitidos y su voluntad por transmitir sus experiencias personales en sus respectivos campos de especialización. Han permitido que se termine este grado con unos conocimientos y habilidades repartidos en varios campos muy candentes de la ingeniería los cuales confio que serán útiles en mi futuro cercano.

Finalmente, mencionar especialmente a Manuel Vargas, agradeciendo su paciencia y confianza en mi persona para este proyecto, así como por su trato y atención extraordinarios. Me ha ayudado a sobrepasar las diferentes incertidumbres que surgían y a cerrar esta etapa con un proyecto de una temática muy interesante y prometedora.

Francisco Javier Román Cortés

Sevilla, 2023

Resumen

En este proyecto se busca diseñar un sistema consistente en un **UAV dotado de múltiples cámaras** que permita realizar el seguimiento de un número variable de objetivos móviles. Acompañando el diseño teórico del sistema es importante respaldarlo con simulación, la cual se realiza en el software CoppeliaSim, sentando las bases para una posible implementación física.

Este sistema multicámara puede ser innovador en diferentes tareas como son: **videovigilancia, monitoreo del tráfico, agricultura,** entre otras, permitiendo automatizar procesos y aprovechar las ventajas de un sistema autónomo en zonas más inaccesibles.

Se plantean y analizan diferentes técnicas de **visión por computador**, indispensables para poder identificar a los objetivos a seguir, así cómo se estudia la geometría relativa entre el sistema UAV y la escena para comprender cómo conseguir un apuntamiento adecuado de los objetivos mediante sus cámaras orientables.

Una vez identificados, se presentan algoritmos para **interaccionar con los objetivos**, tales como agrupación de estos en diferentes *clusters* (grupos), interacción entre la cámara gran angular y las orientables, entre otros.

Se incorpora también una parte de **modelado** de elementos del UAV para emular su comportamiento físico, y que permite desarrollar **estrategias de control** para los mismos.

Finalmente, tras presentar el análisis teórico del problema, será implementado en un **prototipo software** planteado mediante programas basados en **Python** que se comunican con el entorno modelado y permiten verificar el funcionamiento de las técnicas y algoritmos desarrollados.

Abstract

In this project, we aim to design a system consisting of a **UAV equipped with multiple cameras** that allows tracking a variable number of moving targets. Along with the theoretical design of the system, simulation is important to support it, which is carried out in the CoppeliaSim software, laying the groundwork for a possible physical implementation.

This multicamera system can be innovative in various tasks such as **video surveillance, traffic monitoring, agriculture**, among others, enabling automation of processes and taking advantage of the benefits of an autonomous system in more innacessible areas.

Different **computer vision techniques** are proposed and analyzed, which are essential to identify the targets to be tracked. The relative geometry between the UAV system and the scene is also studied to understand how to achieve proper targeting of the objectives through their orientable cameras.

Once the targets are identified, algorithms are presented **to interact with them**, such as grouping them into different clusters, interaction between the wide-angle camera and the orientable ones, among others.

A segment of **modeling** the UAV elements is also incorporated to emulate their physical behavior, allowing the development of **control strategies** for them.

Finally, after presenting the theoretical analysis of the problem, it will be implemented in a **software prototype** designed using **Python** programs that communicate with the modeled environment and allow verification of the functionality of the developed techniques and algorithms.

Índice

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

1 INTRODUCCIÓN

El propósito de este primer apartado es la de presentar la idea del proyecto, describiendo el sistema a plantear, así como los motivos que incitan a su desarrollo. Es interesante además revisar el estado del arte acerca de los diferentes sistemas y aplicaciones basados en el uso de UAV (*Unmanned Aerial Vehicle ó vehículo aéreo no tripulado*) y detallar la estructura y objetivos a alcanzar con este proyecto.

1.1 Motivación

El sistema propuesto consiste en un UAV dotado de una cámara situada en la zona central del mismo de tipo gran angular acompañada de un número definido de cámaras orientables que disponen además de zoom óptico.

Estas cámaras estarán acopladas a unos gimbals, los cuales compensarán las oscilaciones propias del movimiento de vuelo del UAV, permitiendo fijar una orientación controlada de las cámaras independientemente de la del UAV, desacoplando así el problema y solucionando un más que posible problema de *motion blur* en las imágenes obtenidas sin este elemento estabilizador.

Las denominadas cámaras orientables requieren de ir montadas en gimbals de 3 grados de libertad, ya que es necesario poder tener control de la orientación sobre los 3 ejes de estos para lograr un correcto apuntamiento hacia sus objetivos asignados. Por su parte, la idea de la cámara de gran angular es ser estable (evitar oscilaciones que afecten a la cámara) y apuntar constantemente hacia abajo respecto al dron independientemente de su inclinación. Por tanto, para dicha cámara es suficiente con montarla en un gimbal de 2 grados de libertad que garantice ambas condiciones.

El objetivo de la cámara gran angular es abarcar un campo de visión lo más amplio posible, logrando así identificar la mayor cantidad de objetivos posibles de la escena, si no todos ellos. A partir de estos objetivos captados con dicha cámara, se realiza una asignación de objetivos, ya sea individuales o grupos de ellos, para ser seguidos por cada una de las cámaras orientables.

Teniendo en cuenta esto, es necesario que el objetivo asignado a cierta cámara sea también seguido por la cámara Gran Angular, ya que es a través del objetivo en su imagen y ciertas técnicas que se consigue apuntar al objetivo asignado con la cámara orientable correspondiente.

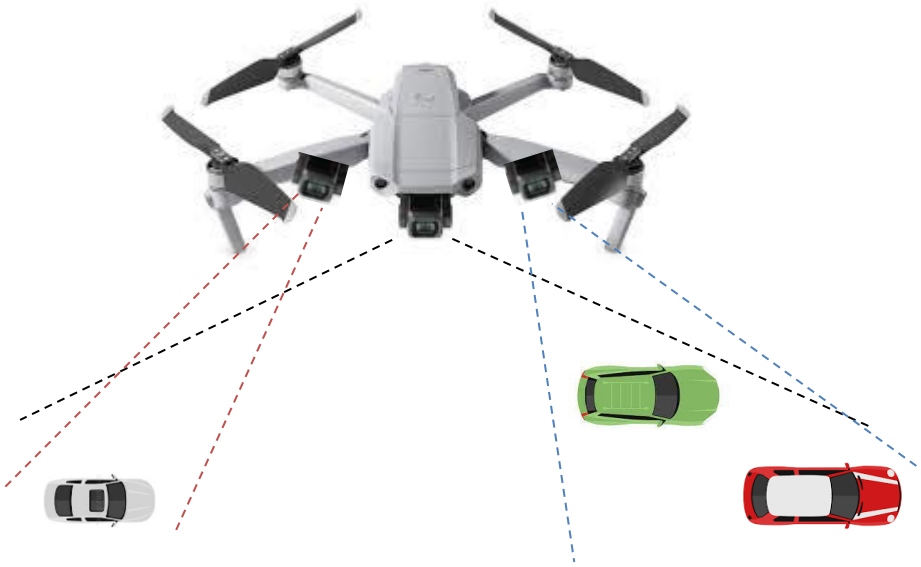


Figura 1-1. Esquema ejemplificativo del proyecto con 2 cámaras orientables.

Es importante recalcar que las particularidades y problemas que surgen de este planteamiento se han enfocado de manera genérica, es decir, partiendo de la indispensable cámara gran angular, se puede parametrizar el número de cámaras orientables de las que dispone el sistema, lo que facilitaría su adaptación a un sistema físico similar al planteado con un número de cámaras orientables dependientes de su aplicación.

El número de cámaras orientables a usar dependerá fuertemente de las características de la aplicación, como por ejemplo el número de objetivos, las limitaciones físicas del UAV (típicamente el *payload* [1]) o los objetivos concretos de la aplicación a realizar. En la **Figura 1-1** se ha planteado un sistema de 2 cámaras orientables para un posible seguimiento de vehículos en lo que podría ejemplificar la supervisión del tráfico de manera autonóma.

Las ventajas de este sistema respecto a un típico sistema UAV con una cámara única son notables. Principalmente, ante la existencia de múltiples objetivos dispuestos de manera dispersa en la escena, se habilita un enfoque más adecuado con las cámaras orientables que proporcionarán imágenes más detalladas de los objetivos, permitiendo la identificación de rasgos claves de los objetivos en tareas de supervisión y vigilancia. Además, se puede lograr el seguimiento simúltaneo de varios objetivos con gran detalle, lo cual sería imposible con un sistema que sólo disponga de una cámara.

1.2 Contexto

En la actualidad, acompañado por los innovadores avances en términos de inteligencia artificial en múltiples ámbitos, incluida la visión por computador, permiten plantear nuevas y cada vez más complejas aplicaciones para sistemas basados en UAVs dotados de visión (mediante una o varias cámaras). Otro factor que favorece el desarrollo de estas aplicaciones son los avances en las velocidades de transmisión de datos, con el asentamiento de las redes 4G y 5G incluso en áreas más remotas se habilita la recepción de imágenes o streaming de vídeo con una latencia cada vez menor incluso para resoluciones de imagen cada vez mayores. Ambos factores principalmente favorecen que las tareas clásicamente asignadas a estos sistemas, como son apoyo en misiones de búsqueda y rescate, supervisión en el ámbito de seguridad o bien monitoreo e inspección de instalaciones o terrenos, sean realizadas cada vez con mayor eficiencia y precisión.

El uso de UAV incorpora la ventaja de poder realizar estas tareas desde cierta altura, pudiendo llegar a cubrir zonas extensas. Asimismo, la versatilidad en su movimiento (destacando en este aspecto los UAV basados en sistema multirotor) los hace muy convenientes a la hora de desplazarse en terrenos irregulares y de difícil acceso.

La amplia comercialización de estos drones multirotor en los últimos años ha propulsado una reducción en los costes de fabricación, motivando avances en escenarios cada vez más diversos tanto en el ámbito de investigación como en el comercial y recreacional.

Tras esta introducción de sus implicaciones actuales, es conveniente revisar su contexto histórico desde sus orígenes hasta nuestros días [2].

Aunque cuando se habla de UAVs, los globos aerostáticos no suelen formar parte de la discusión, desde un punto de vista tecnológico, fueron la primera aeronave que no requerían un piloto humano. [3] Fue así como en 1783, los hermanos Joseph-Michel y Jacques-Étienne Montgolfier fueron los anfitriones de la primera demonstración de una aeronave no tripulada en la localidad de Annonay, Francia.

El globo medía unos 18 m de alto y 13 m de ancho y pesaba unos 400 kg. Ante la familia real y una multitud expectante, el globo cuyos pasajeros eran una oveja, un pato y un gallo, ascendió hasta 600 m de altitud. Sin embargo, una pequeña rotura en la tela provocó que descendiese lentamente durante 8 minutos, tras viajar una distancia de 3.5 km, con sus pasajeros animales intactos.

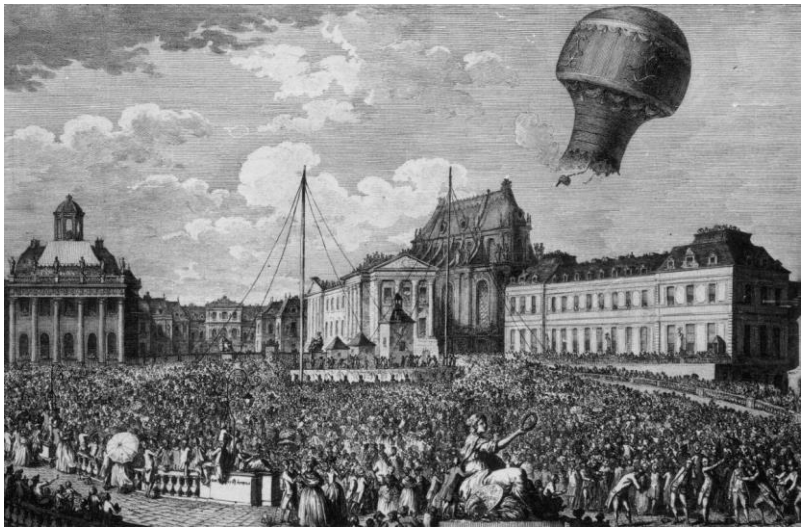


Figura 1-2. Globo aerostático sin tripulación humana de los hermanos Montgolfier en Francia, 1783 [3].

Con el paso de los años y el desarrollo del concepto de cámara similar a cómo la conocemos actualmente en torno a la segunda década del siglo XIX, sería a finales de este siglo, en 1896 de manos de Alfred Nobel, quien llevaría a cabo el lanzamiento de un cohete con una cámara en él. [4] La idea era obtener "mapas fotográficos". Para ello, cada cohete lanzado sería capaz de tomar una fotografía, justo durante su descenso en paracaídas hacia el suelo. Aunque murió y no llegó a realizar un prototipo, se cree que su grupo de investigadores lo realizó y tomó un par de fotos aéreas de la ciudad de Karslkoga en Suecia. Sin embargo, aunque hay controversia si las fotos fueron tomadas realmente mediante ese método, la idea fue bien recibida y puede ser considerado un muy primitivo precedente de los UAVs con visión actuales.

Figura 1-3. Supuesta foto aérea tomada mediante el método de cámara en cohete de Alfred Nobel en 1896 [4].

Sólo unos 20 años después, en 1917, [5] Charles Kettering inventó en Ohio la aeronave no tripulada "*Kettering Aerial Torpedo*", apodada como "*the Bug"*. Disponía de un sistema pre-configurado de controles neumáticos y elctricos que estabilizaban y guíaban la aeronave hacía un objetivo. Tras un período determinado, se cerraba un circuito eléctrico que apagaba el motor y se desplegaban las alas, provocando que la aeronave se precipitase hacia el suelo, donde las 180 libras de explosivos que cargaba detonarían. Aunque se produjeron varias decenas de ellos, nunca vieron combate y se convertiría en una pieza de museo.

Figura 1-4. Aeronave "Kettering Aerial Torpedo" en el museo de las fuerzas aéreas de Estados Unidos [5].

Sería ya a partir de la década de 1960 cuando los UAV comienzan a desarrollarse como los conocemos. Un ejemplo de este salto tecnólogico se presenta en 1973, [6] cuando Israel desarrolló UAVs para vigilancia y exploración, lo que habilitó a los comandantes militares mejorar su comprensión de las situaciones militares. Su gran impacto en este ámbito fomentó un crecimiento en interés por parte de múltiples potencias en esta tecnología.

Figura 1-5. "Tadiran Mastiff" primer UAV militar israelí, que vio acción en la guerra Yom Kippur en 1973 [6].

Este impulso llevó un par de décadas después a la aparición de uno de los UAV con más impacto en el campo militar de manos de Abraham Karem en Estados Unidos [6] y que revolucionaría de nuevo el concepto de esta tecnología. Sus avances incorporaron mejores en múltiples aspectos: eficiencia, reducción de ruido, rango, *payload*, etc. El resultado de su trabajo fue el MQ-1 Predator [7], desarrollado en 1996 y que se convertiría poco después en una pieza clave de las Fuerzas Aéreas estadounidenses en 2001.

Figura 1-6. UAV "MQ-1 Predator" pieza clave de las fuerzas aéreas de EEUU a principios del siglo XXI [7].

Es en estos años cuando los UAV captan interés también en tareas más alejadas del ámbito bélico, donde gracias a los nuevos materiales más ligeros y nuevas tecnologías, como las mejoras en los sistemas de almacenamiento de energía, entre otros, han permitido que tomen partido en el ámbito civil en sectores tales como la agricultura, topología, inspección y mantenimiento de infraestructura industrial, aplicaciones en el sector audiovisual (facilitando filmar escenas desde altura con mayor accesibilidad) etc.

Es también en la década de 2010 cuando los UAV basados en sistema multirotor toman protagonismo frente a los previamente establecidos basados en ala fija. Esto se debe a sus mejores características como menor peso, mayor versatilidad de movimiento, así como su menor coste en cuanto a su producción.

Es así como empresas como *DJI, Parrot o Potensic*, entre otras, que comercializan soluciones cerradas de modelos de quadrotors, tanto destinados para fines lúdicos como los de su gama baja, como sus modelos más avanzados que podrían desempeñar las tareas anteriormente mencionadas.

Figura 1-7. Dron comercial "DJI Phantom 4 RTK" [https://www.raynastech.com/product-page/DJI-Phantom-4-Pro-RTK].

Es en este ámbito donde tendría lugar el sistema planteado a principios de esta introducción. El sistema multicámara a diseñar innovaría en estas nuevas tareas que los UAV están desempeñando siendo posible automatizar el proceso en este caso de la tarea que encaja de manera más natural en nuestro sistema, la vigilancia o supervisión de objetivos.

Se comentaba que las cámaras debían ir montadas en sistemas gimbal para lograr que la imagen no se vea afectada negativamente por las vibraciones propias del vuelo del UAV, así como de su orientación derivada de la inclinación propia que toma al desplazarse. Luego es más que interesante mencionar el origen de los mismos y sus principios de funcionamiento.

Un gimbal [8] es un dispositivo mecánico que se utiliza para estabilizar la cámara o cualquier otro objeto en un movimiento. Consiste en una serie de anillos interconectados que permiten que la cámara permanezca estable, independientemente de los movimientos del operador o del medio ambiente. En la actualidad, los gimbals se utilizan en diversas aplicaciones, como en la fotografía y el cine para obtener tomas estables y fluidas. También se utilizan en la industria de los drones para mantener estables las cámaras y en la industria de los videojuegos para mejorar la experiencia de juego.

Los gimbals modernos funcionan mediante el uso de giroscopios y acelerómetros, que miden la inclinación y la rotación de la cámara. Estos datos se utilizan para controlar los motores en los anillos del gimbal y mantener la cámara estable.

Figura 1-8. Modelo comercial de Gimbal de 3 ejes para montar en un UAV [https://www.4smapper.com/sensors?lightbox=dataItem-l886107r1].

La principal característica de los gimbal son sus grados de libertad y es que, aunque un gimbal de 3 grados de libertad es el tipo más común de gimbal utilizado en la industria, ya que permite una estabilización de la cámara eficaz y suave en tres direcciones diferentes, también hay gimbals de 2 y 4 grados de libertad, pero son menos comunes.

Para concluir este apartado de contextualización es interesante mencionar las líneas de investigación actuales en torno a los UAVs. Un ejemplo es el siguiente artículo [9], donde se realiza un análisis exhaustivo de las aplicaciones de los enjambres de UAVs en una variedad de campos, incluyendo agricultura, cartografía, seguridad, comunicaciones, y vigilancia. Los autores discuten las ventajas y desventajas de usar enjambres de UAVs en lugar de un solo UAV, y describen cómo se pueden coordinar los enjambres de UAVs para realizar tareas complejas. También se discuten los desafíos técnicos y regulatorios que enfrentan los enjambres de UAVs, y se presentan soluciones para abordar estos desafíos.

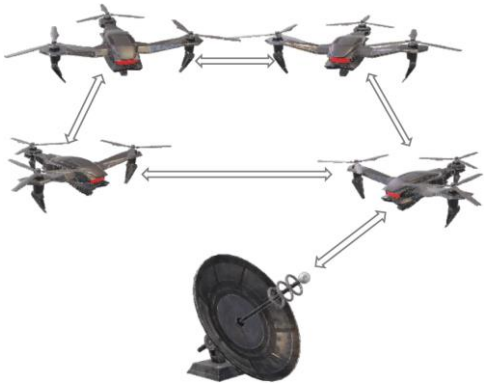


Figura 1-9. Esquema de comunicaciones de un enjambre de UAVs [9].

Aunque el análisis de estas nuevas funcionalidades y aplicaciones de los UAVs presenta gran interés queda fuera del ámbito de estudio de este proyecto, en el cual nos centraremos en describir y plantear soluciones para el sistema descrito.

Con esto queda contextualizado históricamente el problema, con lo que se puede hablar ahora de este sistema propio que se plantea.

1.3 Objetivos

El proyecto se centrará en lograr satisfacer los siguientes objetivos:

- Plantear y resolver de manera teórica los problemas ópticos y geométricos que aparecerían al implementar el concepto de este proyecto propuesto con las funcionalidades deseadas:
    - Apuntar objetivos.
    - Agrupar objetivos.
    - Asignación e intercambio de objetivos.
    - Disponer de un control automático sobre el zoom óptico para encuadrar los objetivos asignados.
- Modelar las dinámicas de los diferentes elementos que conforman el sistema.
- Diseñar estrategias de control de acuerdo con los modelos previamente definidos.
- Simular el sistema diseñado en un entorno 3D.
- Desarrollar un esquema software el cual, trabajando con las imágenes de las múltiples cámaras, sea capaz de realizar el control basado en visión que satisfaga los requerimientos propuestos.
- Realizar simulaciones de diferentes experimentos para verificar el funcionamiento esperado del sistema.

A lo largo de los diferentes apartados de esta memoria se tratará de explicar y resolver los problemas que aparecen al tratar de completar los objetivos ya comentados.

1.4 Herramientas software empleadas

Para facilitar la comprensión del desarrollo de los diferentes apartados, es interesante mencionar las herramientas y programas utilizados en este proyecto:

- *Matlab R2023a.* Utilizado para modelar los sistemas y diseñar sus controladores.
- *Visual Studio Code* en *Ubuntu 20.04,* para programar en *Python 3.8.10* los scripts capaces de resolver los diferentes problemas geométricos y ópticos de forma abstracta, facilitando su implementación posterior.
- Librería *OpenCV 4.7.0*, para hacer uso del conjunto de funcionalidades necesarias que ofrece en cuanto a técnicas de visión por computador.
- *CoppeliaSim V4.5.1 rev2 Edu* ejecutándose en *Ubuntu 20.04.* Constituye el software de simulación en el que se modela el sistema descrito y se verifican los experimentos realizados.

2 INTRODUCCIÓN A COPPELIASIM

La idea del proyecto descrito es adaptar la implementación original realizada en el TFG de David Tejero [10] en ROS + Gazebo al entorno CoppeliaSim, proporcionando así varias alternativas del sistema planteado en dos de los entornos de simulación de robótica más populares en la industria.

Como tal vez el software CoppeliaSim puede no ser tan conocido, es interesante dedicarle un breve capítulo a explicar algunos de los aspectos más importantes de su funcionamiento.

2.1 Instalación y uso

Este software se puede instalar tanto en Windows como en Ubuntu pero varía la forma en que se lanza el software. Destacar que a la hora de usar este software con un "programa cliente" que controle la simulación, es más que recomendable usar Ubuntu, ya que esta forma de uso parece estar más optimizada para este sistema operativo y la latencia de comunicaciones entre el script de Python es mucho menor que si se trata de hacer lo mismo en Windows. Por tanto, tras probarlo en ambos sistemas operativos, se ha terminado utilizando la versión en Ubuntu por su mejor rendimiento.

Al ir a la página de descargas: https://www.coppeliarobotics.com/downloads, esta detecta automáticamente el sistema operativo desde el que se accede y ofrece varias opciones de descarga:

- **Versión *Player:*** La cual es gratis para todo el mundo y permite su uso comercial, aunque carece de ciertas funcionalidades de edición.
- **Versión *Edu:*** Destinada a estudiantes, profesores, universidades u otras entidades educativas. Proporciona acceso a todas las funcionalidades, pero no permite su uso comercial.
- **Versión *Pro:*** Sin ninguna restricción de uso, pero no es gratuita.

2.1.1 Windows

Una vez descargado el ejecutable de la versión deseada, se instala de manera similar a cualquier otro ejecutable (.exe). Tras instalarlo, ejecutar el programa es tan simple como hacer doble click en el icono que aparecerá en el escritorio.

Figura 2-1. Icono de CoppeliaSim en Windows.

2.1.2 Ubuntu 20.04

En este caso, se descarga una carpeta comprimida (.tar.xz) la cual se debe extraer. Una vez extraída, se puede mover la carpeta resultante al directorio de trabajo del proyecto y se puede acceder al programa mediante varios comandos en un terminal (varían según versiones y path donde se sitúen los archivos):

- ***cd*** *~/path_to_extracted_Coppelia_folder/CoppeliaSim_Edu_V4_5_1_rev2_Ubuntu20_04/*
- ***./coppeliaSim.sh*** *~/path_to_desired_scene/scene.ttt*

Notar que los archivos con extensión ".ttt" corresponden a escenas guardadas de CoppeliaSim. Tras estos pasos se abrirá el programa en cuestión con la escena cargada. Si se proporciona una escena no existente, simplemente abrirá el programa con la escena vacía por defecto, la cual tiene el mismo aspecto tanto en Windows como en Ubuntu.

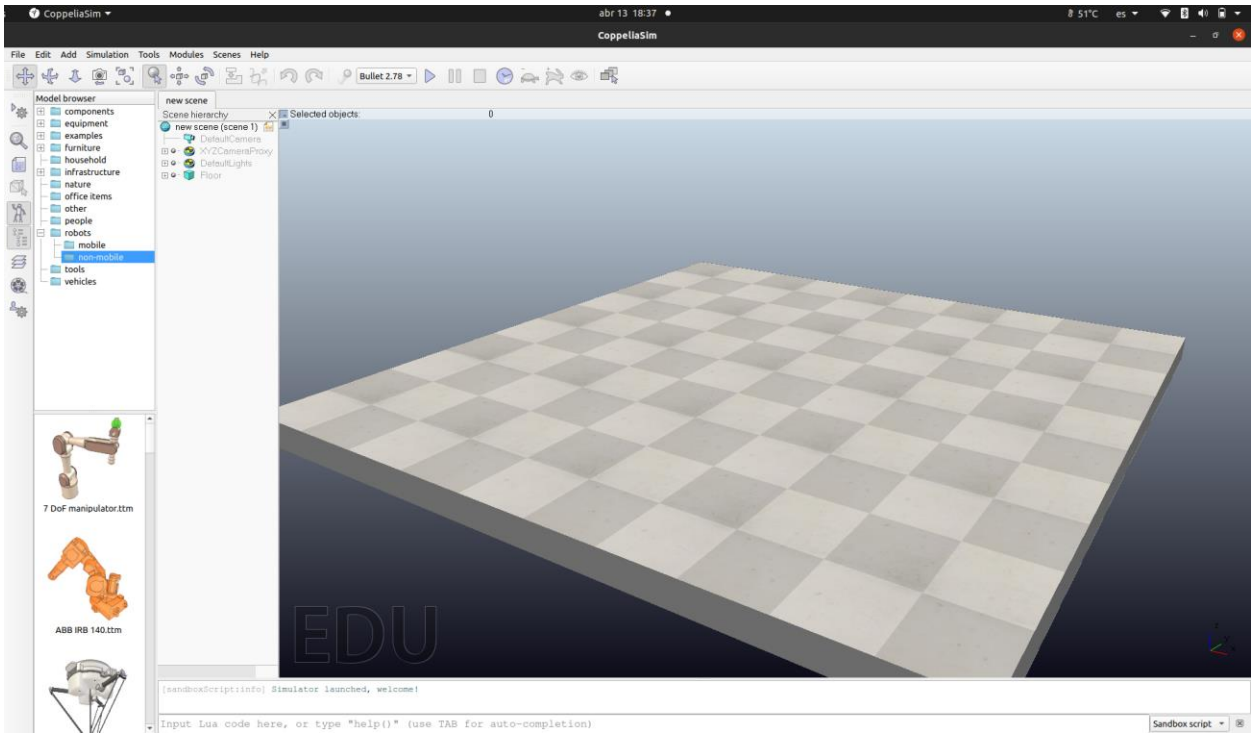


Figura 2-2. Apariencia por defecto al abrir CoppeliaSim (similar en Windows).

2.2 Revisión de los principales menús y opciones

Obviando las herramientas de la barra horizontal superior, que permiten lanzar la simulación, encuadrar la imagen según ciertos objetos, rotar o desplazar la vista, mover o rotar entidades de la escena, entre otros. Se puede pasar a describir brevemente los menús fundamentales que ofrece:

2.2.1 Modelos predefinidos de robots y otros elementos

En el siguiente menú que por defecto aparece visible, se pueden encontrar multitud de modelos de robots y otros elementos que podrían ser útiles para elaborar todo tipo de entornos. Destacar que el propio de quadrotor está disponible en la sección "robots/mobile"de este menú. Para añadirlos a la escena basta con arrastrar el modelo deseado a la misma.

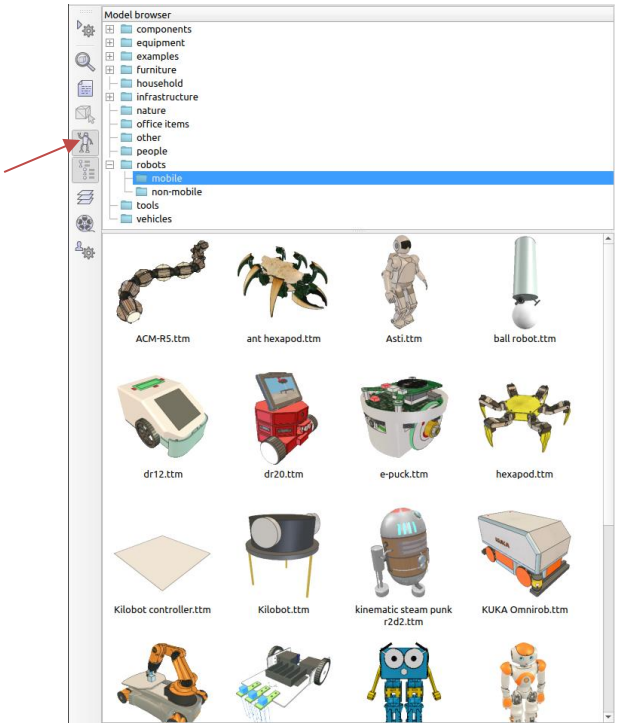


Figura 2-3. Submenú que contiene modelos predefinidos agregables a la escena.

2.2.2 Arbol jerárquico de los elementos de la escena

Este menú también aparece por defecto visible y permite identificar los objetos presentes en la escena, establecer relaciones de dependencia jerárquicas entre elementos y realizar diversas operaciones con ellos.

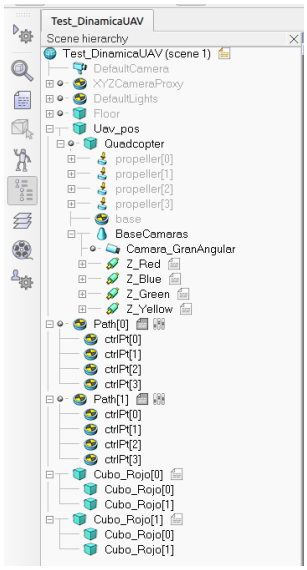


Figura 2-4. Submenú que contiene los objetos presentes en la escena y sus relaciones jerárquicas.

2.2.3 Menú “Add”

Este menú permite agregar diferentes elementos típicos en simuladores de robótica como son:

- Figuras primitivas (Esferas, Cuboides, Cilindros, Discos, etc.).
- Articulaciones (de revolución, prismáticas y esféricas).
- Nubes de puntos.
- Octrees.
- Sensores de proximidad.
- Cámaras (de tipo ortogonal y proyectiva).
- Sensor de fuerza.
- Trayectorias configurables mediante “waypoints”.
- Scripts (en lenguaje Lua) asociados para los objetos de la escena que permiten controlar su comportamiento de manera programática.

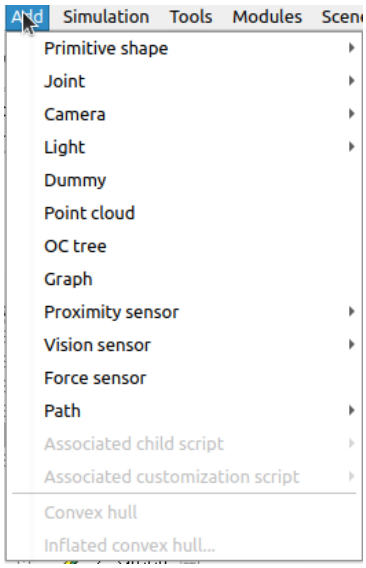


Figura 2-5. Menú “Add” con sus diferentes opciones.

2.2.4 Menú “Tools”

La principal utilidad de este menú es la de modificar las propiedades y configuración de elementos de la escena. Por ejemplo, aplicado a una cámara permite modificar diferentes parámetros del modelo de la cámara.

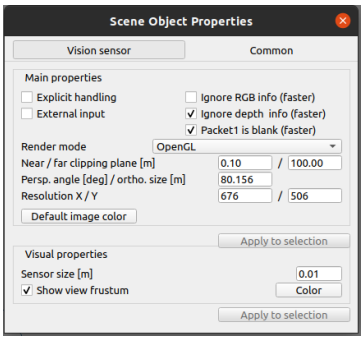


Figura 2-6. Menú “Tools” aplicado a una cámara de la escena.

Aunque hay más opciones y funcionalidades que no se han mencionado, con las anteriores se obtiene una idea introductoria de las posibilidades que ofrece el software y cómo realizar tareas básicas en el mismo.

Luego con esto pasamos a mencionar la última parte fundamental que permite desarrollar un cliente que se comunique con CoppeliaSim.

2.3 API de programación remota para CoppeliaSim y manuales

Para esta tarea, se ofrecen múltiples opciones:

- **Legacy remote API:** Se trata de la manera original de crear un cliente, aunque está desaconsejada ya que carece de soporte y manetinimiento.
- **ZeroMQ remote API:** Consiste en la opción recomendada de crear un cliente para lenguajes como Python, C++, Matlab, Octave, Java, Lua y Rust.
- **WebSocket remote API:** Opción recomendada para clientes escritos en JavaScript.

Como en este proyecto se ha usado Python, se ha optado por *ZeroMQ*, el cual es además la opción más versátil al permitir multitud de lenguajes.

https://www.coppeliarobotics.com/helpFiles/en/zmqRemoteApiOverview.htm

Su uso en un script no es del todo trivial y hay que tener en cuenta las siguientes consideraciones.

El módulo *ZeroMQ* está contenido dentro de la carpeta mencionado en el primer apartado de este capítulo y tanto en Windows como en Ubuntu, es necesario desde el propio script de Python añadirlo a las variables de entorno para poder importarlo adecuadamente. Para ello basta con añadir al comienzo de cada script encargado de actuar como cliente de CoppeliaSim las siguientes órdenes:

Código 2.1 Configuración de variables de entorno para hacer uso del plugin de ZMQ en CoppeliaSim.

```
import sys
sys.path.insert(0,'~/path_to_extracted_Coppelia_folder/
CoppeliaSim_Edu_V4_5_1_rev2_Ubuntu20_04/programming/zmqRemoteApi/clients/python')
from zmqRemoteApi import RemoteAPIClient
```

Una vez hecho esto, la documentación recomienda incluir lo siguiente para configurar y comenzar la simulación desde nuestro propio cliente (sólo en uno de ellos si hay varios clientes):

Código 2.2 Configuración de parámetros e inicio remoto de la simulación desde cierto cliente.

```
client = RemoteAPIClient()
sim = client.getObject('sim')
defaultIdleFps = sim.getInt32Param(sim.intparam_idle_fps)
sim.setInt32Param(sim.intparam_idle_fps, 0)
client.setStepping(False)

sim.startSimulation()
```

Finalmente, ese objeto ***"sim"*** es el que nos habilitará a usar la enorme multitud de funciones recogidas en el siguiente manual para interactuar con la escena de CoppeliaSim desde nuestro cliente remoto:

https://www.coppeliarobotics.com/helpFiles/en/apiFunctions.htm

2.4 Comunicaciones entre Publisher y Subscriber con ZMQ

Por la compleja naturaleza del problema y la necesidad de realizar cálculos y diferentes tareas en paralelo, se hace necesario utilizar un esquema similar al de ROS en cuanto al uso de nodos y publishers/subscribers. Para ello se puede hacer las funciones genéricas del propio módulo de zmq (no es específico de Coppelia se podría usar para otras tareas que requieran de comunicación entre "nodos").

Para ello se requiere tener instalado el módulo ***pyzmq*** el cual se instala con la herramienta ***pip*** de manera simple:

Código 2.3 Instalación de librería pyZMQ.

```
pip install pyzmq
```

2.4.1 Publisher

Una vez se dispone de este módulo, el publisher consiste en un socket de tipo PUB que enviará mensajes a los subscriptores usando el topic especificado y con la dirección asociada especificada. Los pasos en código para configurarlo son:

Código 2.4 Creación y configuración de un Publisher de la librería ZeroMQ.

```
import zmq
# Crear un objeto Context de ZeroMQ:
context = zmq.Context()
# Crear un socket de tipo PUB:
pub_socket = context.socket(zmq.PUB)
# Asociar el socket a una dirección y un puerto (* para todas las direcciones, para
IP específica sustituir # * por la IP)
pub_socket.bind("tcp://*:5555")
```

Una vez creado y configurado, debido a que los sockets de ZeroMQ sólo pueden enviar y recibir "byte strings", habría que codificar el mensaje para que sea de este tipo, ya sea lo que queramos enviar una lista, un entero u otros tipos de datos. Es importante especificar también el topic, el cual también debe codificarse de la misma forma, para que el socket receptor que espera datos en dicho topic pueda recibirlos. Se expone en el siguiente fragmento un ejemplo de envío de datos a través del Publisher creado en el **código 2.4:**

Código 2.5 Ejemplo de envío de datos a un cierto topic con un Publisher de ZeroMQ.

```
topic = "topic_name"
test_list = [1,2,3,4,5]
test_int = 10
# La manera de enviar es creando una lista con el primer elemento el topic
codificado y el segundo los datos codificados, para enviar los datos se codifican
de igual manera:
message = [topic.encode(), str(test_list).encode()]
message = [topic.encode(), str(test_int).encode()]
pub_socket.send_multipart(message)
```

Lógicamente, el topic definido en el suscriptor deberá coincidir con este.

2.4.2 Subscriber

Por su parte, el subscriber, que estará en otro "nodo" deberá configurarse para que reciba los datos del publisher que le corresponda. En estos fragmentos, para servir de ejemplo, se va a plantear un subscriber que reciba los datos del publisher creado en los **códigos 2.5 y 2.6**. Para ello se hace lo siguiente:

Código 2.6 Creación y configuración de un Subscriber de la librería ZeroMQ.

```
import zmq
# Crear un objeto Context de ZeroMQ:
context = zmq.Context()
# Crear un socket de tipo SUB:
sub_socket = context.socket(zmq.SUB)
# Conectar el socket suscriptor a la misma dirección y puerto usados por el
Publisher correspondiente:
sub_socket.connect("tcp://localhost:5555")
# Suscribirse al topic en cuestión:

sub_socket.setsockopt(zmq.SUBSCRIBE, b"topic_name")
```

Una vez creado y configurado el suscriptor, basta con recibir y decodificar el mensaje. Destacar que esta espera de datos se puede hacer de manera bloqueante y no bloqueante, aunque si se hace de forma no bloqueante habría que tratar la excepción de que no haya mensaje con un bloque **try-except** para evitar que el código se interrumpa por este motivo. Se ejemplifica esto a continuación.

Código 2.7 Espera y decodificación de mensaje de manera bloqueante.

```
# Recibir de manera bloqueante (el programa espera en esta línea hasta recibir el
mensaje)
message_rcv = sub_socket.recv_multipart()
# Según el tipo de dato que se vaya a recibir se decodifica de forma diferente:
received_list = eval(message_rcv[1].decode())    # Para listas
received_int = int(message_rcv[1].decode())      # Para ints
```

Código 2.8 Espera y decodificación de mensaje de manera no bloqueante.

```
# Recibir de manera no bloqueante (el programa continúa si no recibe mensaje)
while True:
    try:
        message_rcv = sub_socket.recv_multipart(zmq.NOBLOCK)
        # Si no ha llegado será None
        if message_rcv is not None:
            received_list = eval(message_rcv[1].decode())    # Para listas
            received_int = int(message_rcv[1].decode())      # Para ints

    except zmq.error.Again:
        # Tratar el error

        pass
```

Con esta breve explicación y la documentación adjunta, se sientan las bases para la creación de un cliente remoto que comience la simulación o se conecte a ella (si ya ha comenzado) y se comunique en "tiempo real" con CoppeliaSim actuando sobre dicha simulación.

2.5 Recomendaciones para resolver problemas complejos en CoppeliaSim

En los apartados anteriores se ha explicado como instalar CoppeliaSim, sus menús básicos y cómo realizar operaciones con clientes remotos escritos en *Python*.

Sin embargo, a la hora de enfrentarse a la tarea de modelar y programar escenas y sistemas relacionados con el campo de la robótica, uno puede sentirse desorientado ante falta de referencias en las que estudiar cómo resolver esas situaciones más complejas.

Dicha situación ha sucedido al tratar de resolver el problema planteado en este proyecto y su modelado en un primer contacto con CoppeliaSim. Por tanto, tras encontrar varias fuentes con las que entender mejor el programa, es conveniente recogerlas aquí como posibles alternativas para lograr modelar y programar lo que se pretenda con mayor facilidad.

2.5.1 Canal de Youtube de Leopoldo Armesto, Profesor de la UPV

Aunque el profesor Leopoldo Armesto de la Universidad Politécnica de Valencia presenta en su canal multitud de videos relacionados con sistemas robóticos tanto en el ámbito teórico como práctico, interesa en este caso especialmente la siguiente lista de reproducción:

https://www.youtube.com/watch?v=l32IiRkCwxg&list=PLjzuoBhdtaXOYfcZOPS98uDTf4aAoDSRR

En ella se recogen un total de 87 vídeos en los que trata por una parte los principios básicos del programa, como los menús, jerarquía entre objetos, articulaciones y explicación de los diferentes sensores disponibles.

También trata exhaustivamente el proceso de modelado de un robot móvil particular, lo cual puede dar ideas o resolver aspectos necesarios para el modelado de otros sistemas.

Finalmente, recoge y explica ejemplos de sistemas programados que hacen tareas relativamente complejas como por ejemplo son:

- Programación de robot siguelíneas con evitación de obstáculos.
- Implementación de odometría.
- Resolución de laberinto con robot móvil mediante algoritmo de seguir la pared de un único lado.
- Implementación de campos potenciales para robot móvil.
- Implementación de creación de mapas de ocupación usando plugins.
- Control de brazos robóticos de 6 grados de libertad.

También trata muy ligeramente los sensores visuales o cámara, proporcionando al menos una idea básica de cómo trabajar con ellos.

Como vemos, se aprecia que, gracias a su contribución, hay multitud de tareas típicas de robótica que ya están relativamente resueltas por él y se podrían aplicar con relativa facilidad a otros robots móviles particulares u otros escenarios.

Sin embargo, hay una rama que no recoge, la robótica aérea en CoppeliaSim. Para la cual se dejarán otras alternativas con las que buscar facilitar el trabajo a posibles proyectos de este tipo en este software.

De nuevo, para tareas relacionadas con robots móviles terrestres ó robots manipuladores en CoppeliaSim, recomiendo fervientemente acudir a esta lista de vídeos, ya que posiblemente alguno de ellos proporcione una base valiosa desde la que partir.

2.5.2 Ejemplos de escenas predeterminadas que vienen instaladas con CoppeliaSim

Aunque no es fácil de encontrar y hacer uso de ellas, dentro del software instalado hay unos directorios con escenas muy interesantes que pueden resolver multitud de dudas. Se va a explicar como acceder a ellas en la instalación de *Ubuntu 20.04*, aunque en Windows bastaría con ir a la carpeta del programa instalado y buscarlas manualmente con el explorador de archivos.

Sin embargo, en Ubuntu, una vez se descarga el programa y se descomprime, tendremos una carpeta con un nombre similar a este (según la versión y SO): *CoppeliaSim_Edu_V4_5_1_rev2_Ubuntu20_04*.

Abriendo dicha carpeta, aparecen gran cantidad de librerías y plugins, pero los ejemplos que interesan se encuentran en la carpeta *scenes*.

En ella aparecen multitud de escenas ya programadas, eso sí apenas hay ejemplos usando clientes externos, pero para ello se recogía el planteamiento explicado en el *Apartado 2.2*. Estas escenas plantean problemas bastante interesantes de diferentes ámbitos.

Es interesante recoger algunos de los más interesantes para ver qué posibilidades ofrecen las escenas de ejemplo. Notar que ofrecen una simulación funcional y totalmente completa a partir de las cuales se podrían adquirir conocimientos o procedimientos útiles.

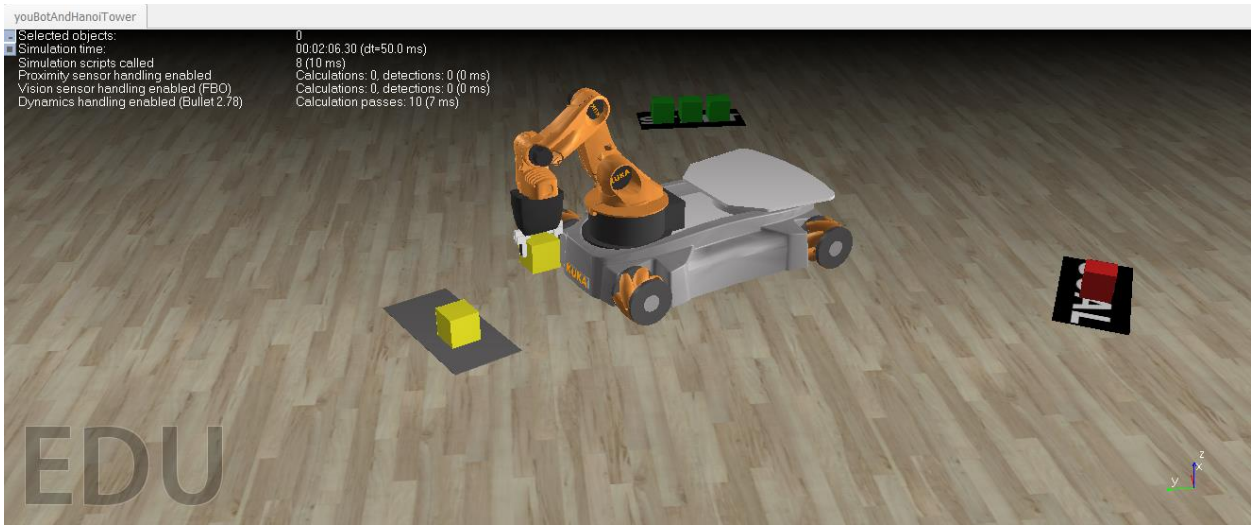


Figura 2-7. Robot móvil con brazo manipulador resolviendo el problema de la torre de Hanoi.

Vemos como este ejemplo resuelve una tarea no tan común, el control de un robot móvil con un brazo robótico incorporado, lo cual ya alberga bastante complejidad.

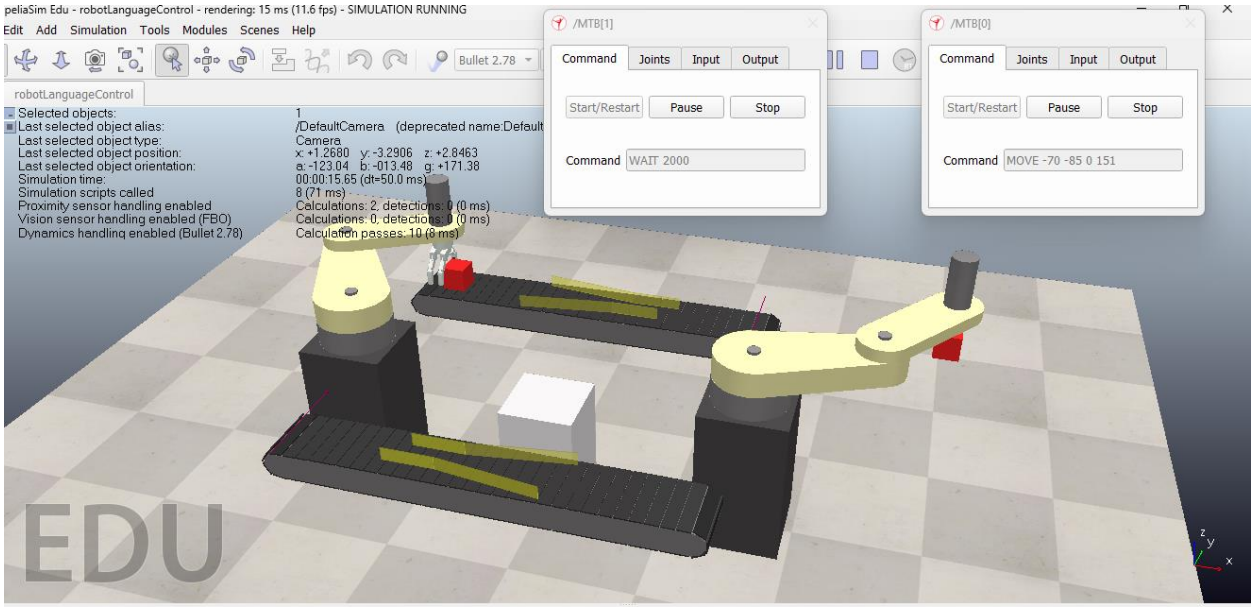


Figura 2-8. Pareja de robots manipuladores intercambiando objetos mediante las cintas transportadoras de manera sincronizada.

La cooperación entre brazos robóticos sí es algo más común en robótica y este ejemplo ofrece una buena base desde la que partir si tuviésemos que realizar una tarea similar. Notar que cada brazo está brograma con comandos en lenguaje RAPID o equivalente, con comandos del tipo “MOVE”, “WAIT”, etc.

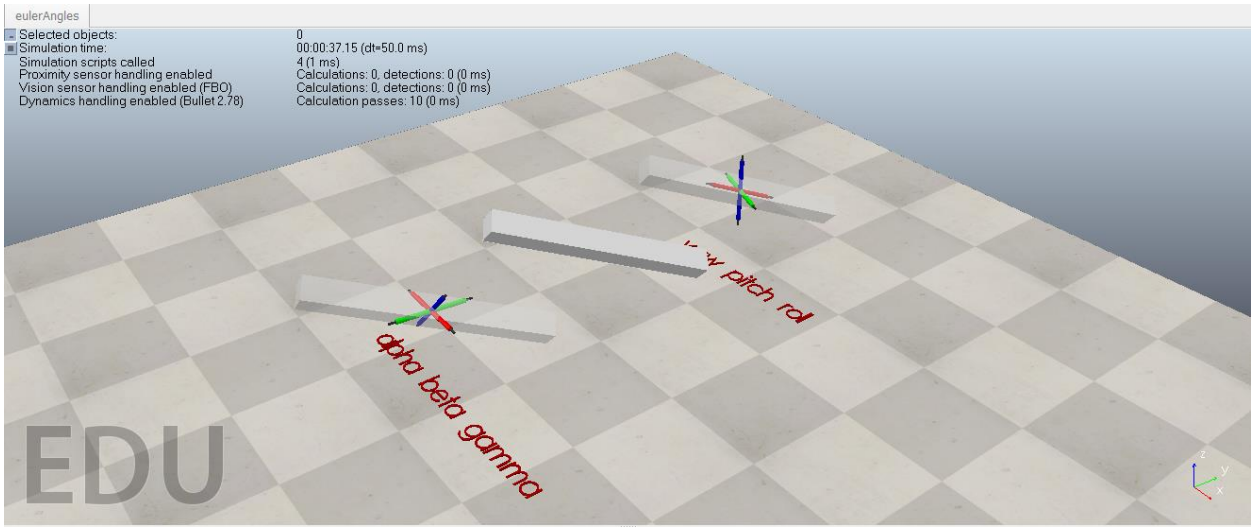


Figura 2-9. Ejemplo de conversión de ángulos típicos de CoppeliaSim (alfa, beta, gamma) a Ángulos de Euler (yaw, pitch, roll).

Este ejemplo ha sido particularmente útil ya que me ofreció la solución a la difícil tarea de modelar el gimbal de 3 grados de libertad que modificase su orientación de acuerdo a la convención de los ángulos de Euler.

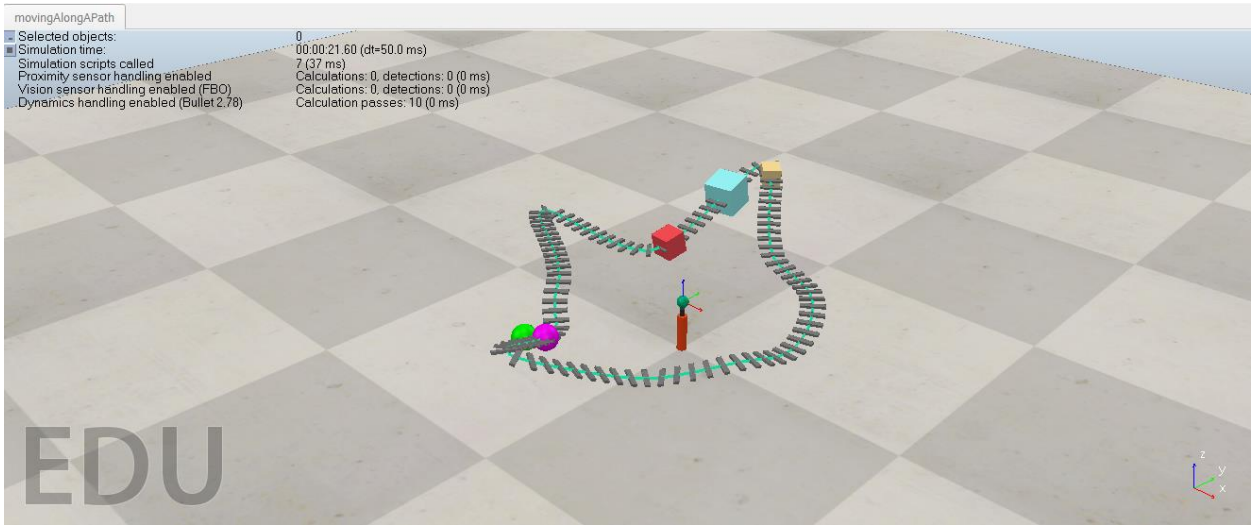


Figura 2-10. Diferentes objetos recorriendo una trayectoria con diferentes configuraciones de movimiento.

Este ejemplo, aunque es más simple, es fundamental para entender cómo generar una trayectoria que defina el movimiento que deseemos y cómo hacer que un cierto objeto o entidad se desplace a lo largo de dicha trayectoria. Ha sido útil para generar trayectorias elípticas en los objetivos en alguna de las escenas que se usan para los propios experimentos de este proyecto.

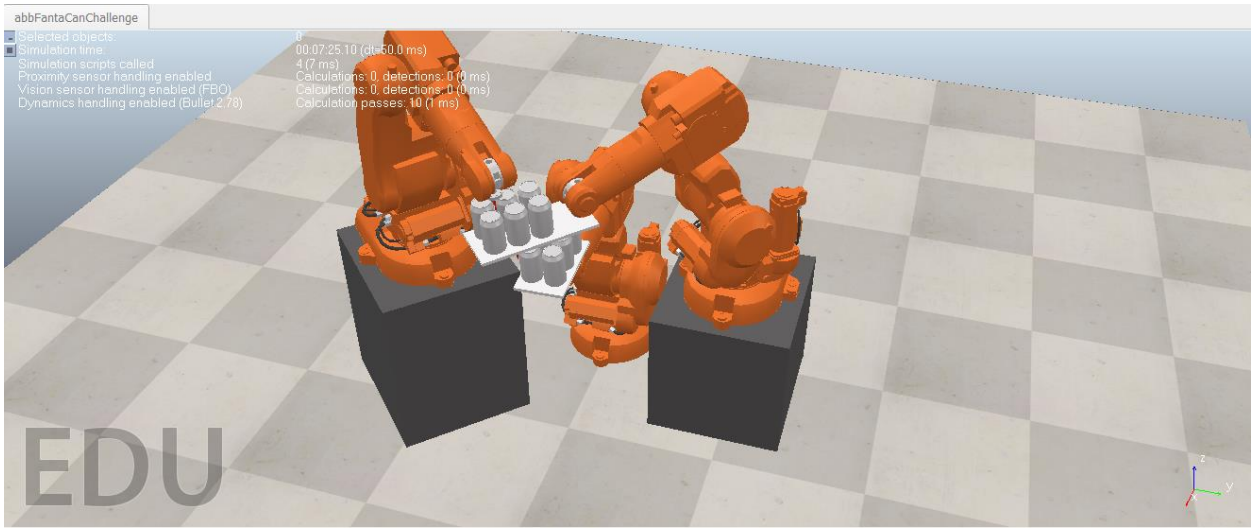


Figura 2-11. Programación y modelado del ABB fanta can challenge (https://www.youtube.com/watch?v=SOESSCXGhFo) con robots ABB IRB 140.

Este ejemplo demuestra una programación de un popular reto realizado por ABB con algunos de sus brazos robóticos para demostrar su gran control y precisión de movimiento moviendo con un robot una bandeja con latas y con el otro simultáneamente desplazando su efector final entre los huecos de las latas mientras el otro las mueve desplazando su bandeja.

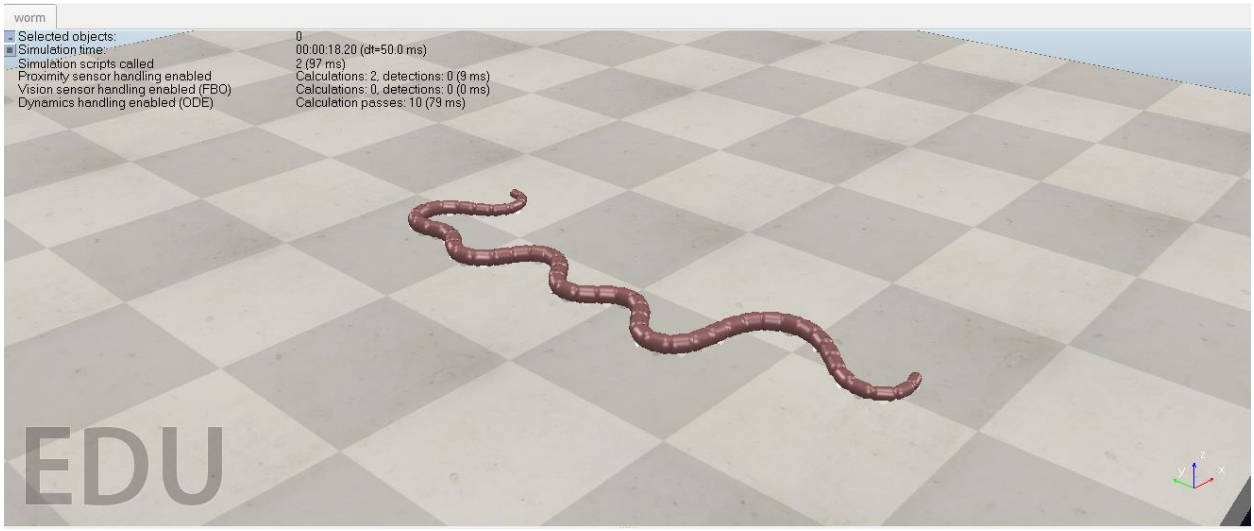


Figura 2-12. Robot inspirado en la fisionomía de un gusano emulando su movimiento.

También se adentra ligeramente en la robótica bioinspirada, programando el modelo de un robot de tipo gusano que además emula con cierta precisión sus movimientos característicos.

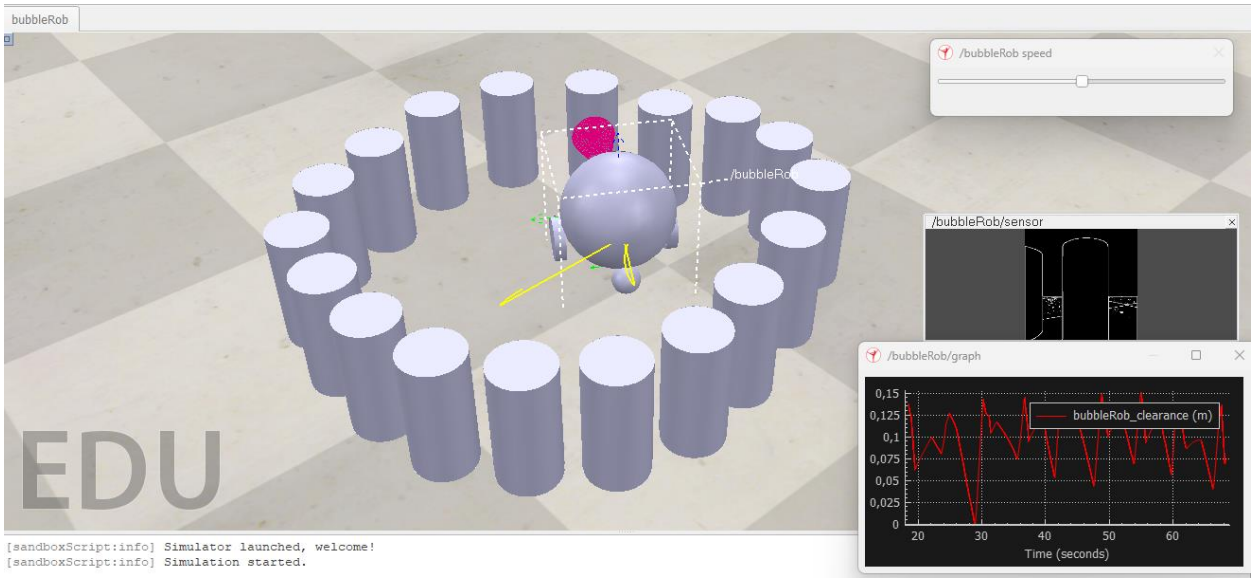


Figura 2-13. bubbleRob dotado con sensores de proximidad evitando colisiones.

De nuevo, tenemos otra de las tareas típica en robótica, la evitación de colisiones, en este caso usando sensores de proximidad. Es un ejemplo básico pero permite entender cómo usar y programar sensores de proximidad en CoppeliaSim.

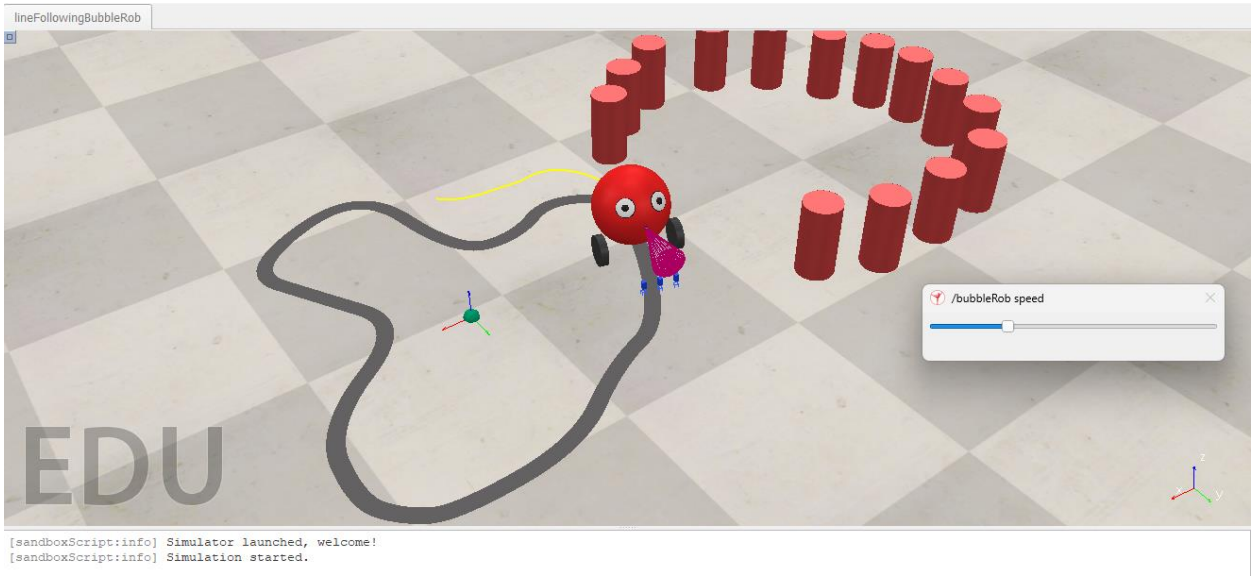


Figura 2-14. bubbleRob dotado con trío de sensores infrarrojos siguiendo línea.

Finalmente, se muestra este último ejemplo en el que el modelo bubbleRob realiza el seguimiento de un camino definido por una línea y con los sensores infrarrojos y algoritmos clásicos para resolver este problema sigue adecuadamente la trayectoria definida.

Aunque hay también ejemplos de conexiones remotas y uso de plugins de ROS y conexión con propio ROS, se indica que existen y se recomienda acceder a ellos si se requiere ese tipo de funcionalidad.

Como vemos, estos ejemplos mostrados, junto al resto de los no mencionados existentes, permiten resolver u ofrecer un planteamiento inicial a multitud de problemas típicos y no tan comunes que puede ser muy útil para tomarlo como referencia para ciertos problemas en lugar de tratar de resolverlo desde cero.

2.5.3 Foro para usuarios de CoppeliaSim

Según las características o complejidad del problema a resolver, puede ocurrir que las dos maneras anteriores no sean suficiente para obtener ideas o técnicas para afrontar su resolución.

En dicho caso, recomiendo acudir al foro de CoppeliaSim, donde, aunque hay que escribir en inglés, si planteas tu duda de manera comprensible, empleados expertos en CoppeliaSim tratarán de responderte de la manera que ellos vean conveniente para resolver tu problema o pregunta relacionado con este software.

Como las cuestiones de otros usuarios son visibles, lo natural es buscar en primer lugar si ha habido en el pasado alguien con la misma duda o situación en dicho foro, y ya si no es así, formular uno mismo la nueva pregunta.

Para acceder al foro, se puede hacer según el siguiente enlace:

https://forum.coppeliarobotics.com/index.php

Lógicamente, hay que crearse una cuenta para poder escribir en él. Con ese enlace podremos ver en *General Questions* las preguntas de otros usuarios.

Una vez ahí, para formular una pregunta, basta con hacer click en *New Topic*, como se muestra en la **Figura 2-15:**

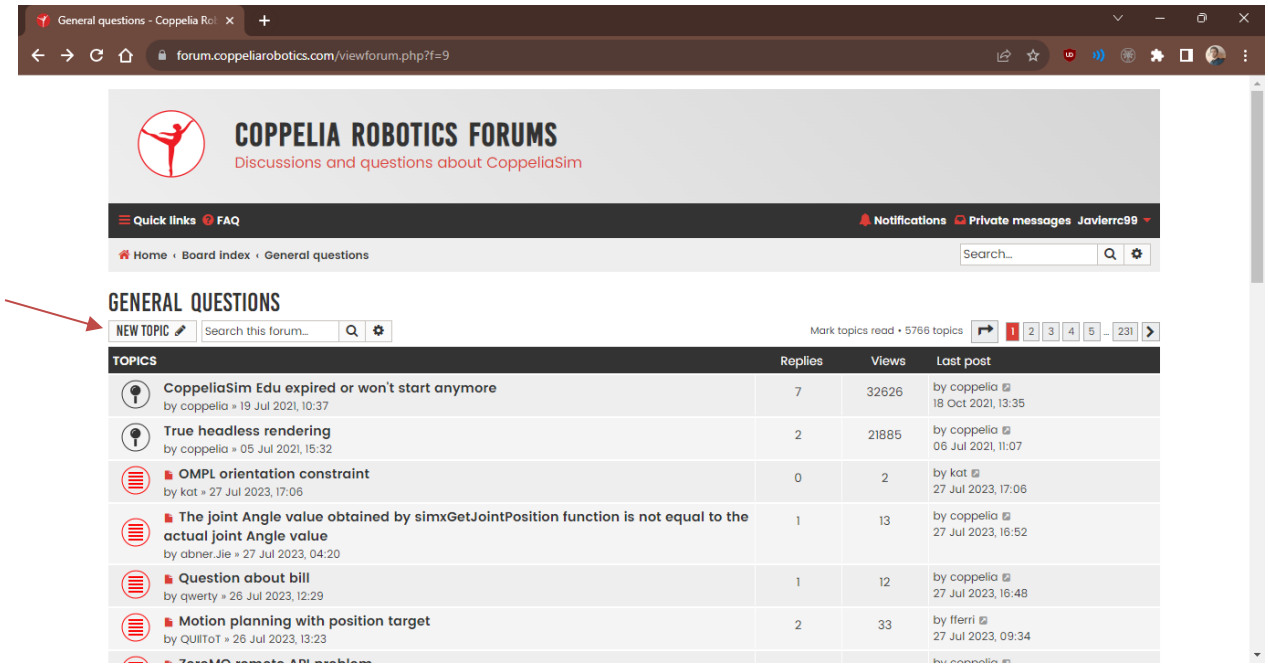


Figura 2-15. Procedimiento para formular una pregunta en el foro de CoppeliaSim.

Tras ello, basta con escribir el "Asunto" y la descripción de la pregunta o problema y darle a *Submit* y la pregunta ya estará subida al foro para ser revisada por empleados de CoppeliaSim que ya han resuelto infinidad de cuestiones y normalmente ofrecen respuestas útiles.

Como ejemplo de la utilidad del foro, dejo el conjunto de mis discusiones a lo largo del desarrollo del proyecto

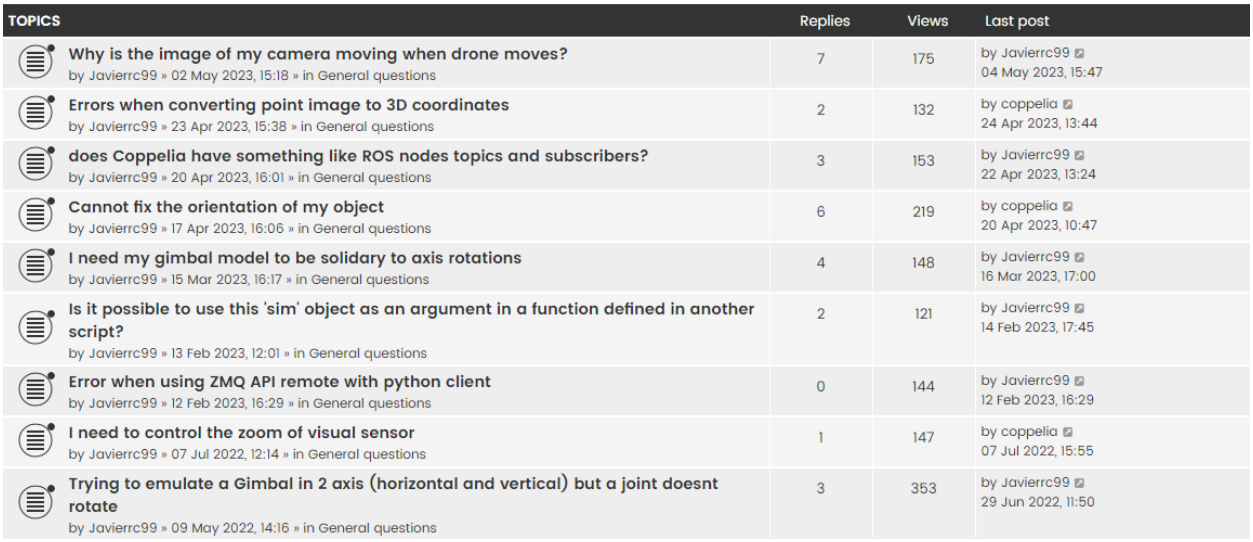
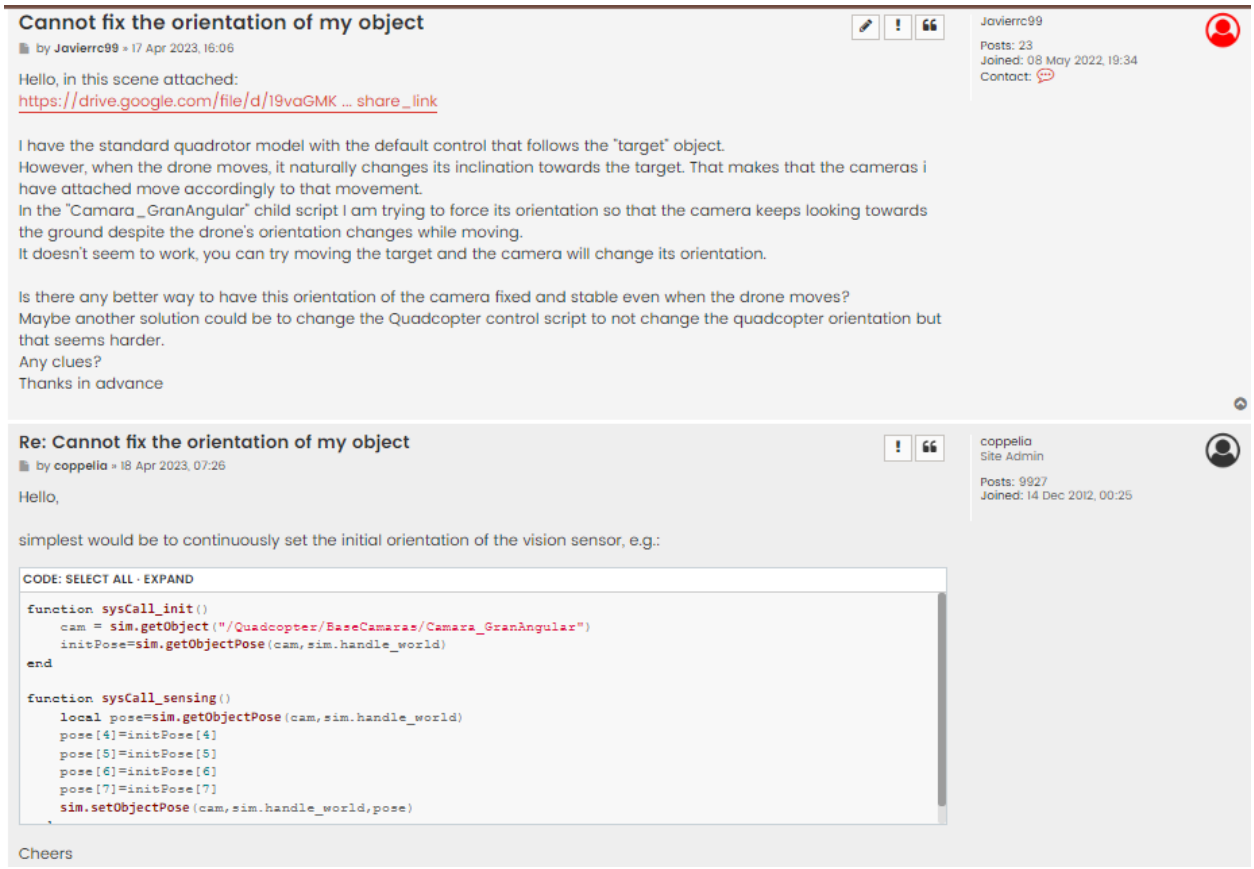


Figura 2-16. Conjunto de preguntas formuladas en el foro a lo largo del desarrollo del proyecto.

Se deja también un fragmento de discusión tratando de encontrar la solución a alguno de estos problemas, donde se aprecia el tipo de respuestas que ofrecen y cómo posiblemente resuelven tu duda.



Cannot fix the orientation of my object

by **Javierrc99** » 17 Apr 2023, 16:06

Hello, in this scene attached:
https://drive.google.com/file/d/19vaGMK ... share_link

I have the standard quadrotor model with the default control that follows the "target" object.
However, when the drone moves, it naturally changes its inclination towards the target. That makes that the cameras i have attached move accordingly to that movement.
In the "Camara_GranAngular" child script I am trying to force its orientation so that the camera keeps looking towards the ground despite the drone's orientation changes while moving.
It doesn't seem to work, you can try moving the target and the camera will change its orientation.

Is there any better way to have this orientation of the camera fixed and stable even when the drone moves?
Maybe another solution could be to change the Quadcopter control script to not change the quadcopter orientation but that seems harder.
Any clues?
Thanks in advance

Javierrc99
Posts: 23
Joined: 08 May 2022, 19:34
Contact:

Re: Cannot fix the orientation of my object

by **coppelia** » 18 Apr 2023, 07:26

Hello,

simplest would be to continuously set the initial orientation of the vision sensor, e.g.:

CODE: SELECT ALL · EXPAND

```
function sysCall_init()
    cam = sim.getObject("/Quadcopter/BaseCamaras/Camara_GranAngular")
    initPose=sim.getObjectPose(cam,sim.handle_world)
end

function sysCall_sensing()
    local pose=sim.getObjectPose(cam,sim.handle_world)
    pose[4]=initPose[4]
    pose[5]=initPose[5]
    pose[6]=initPose[6]
    pose[7]=initPose[7]
    sim.setObjectPose(cam,sim.handle_world,pose)
```

Cheers

coppelia
Site Admin
Posts: 9927
Joined: 14 Dec 2012, 00:25

Figura 2-17. Ejemplo de pregunta formulada y respuesta proporcionada por parte de los administradores.

3 SEGUIMIENTO VISUAL DE OBJETIVOS

Uno de los problemas fundamentales que se debe resolver en cuanto a la videovigilancia automática es la posibilidad de identificar en imágenes consecutivas el mismo objetivo. Este proceso está lejos de ser trivial, ya que se enfrenta a numerosas situaciones que escalarían la dificultad de lograr un sistema robusto, como por ejemplo, cambios de luminosidad, cambios en la posición y orientación del objeto, oclusión parcial o total del objetivo seguido, etc.

Partiendo del detallado análisis de los diferentes algoritmos de *tracking* que ofrece la librería **OpenCV** realizado en el capítulo 2 del TFG [10] para *datasets* basados en secuencias de imágenes propias de las captadas por un UAV, en este capítulo se van a mencionar ciertos fundamentos del seguimiento visual y se hará algo de hincapié en el algoritmo de tracker que resulta más versátil ante multitud de situaciones según este análisis y que se elige para nuestro problema: **KCF (Kernelized Correlation Filter)** para comprender mejor en qué basa su funcionamiento.

Además de este algoritmo, se pueden mencionar el resto de ellos disponibles, por conocer de su existencia y origen a lo largo del tiempo junto a los artículos que los describen:

- *Boosting,* 2006. [11]
- *MIL (Multiple Instance Learning),* 2009. [12]
- *MedianFlow,* 2010. [13]
- *MOSSE (Minimum Output Sum of Squared Error),* 2010. [14]
- *TLD (Tracking Learning Detection),* 2011. [15]
- *KCF (Kernelized Correlation Filter),* 2014. [16]
- *CSRT (Channel and Spatial Reliability Tracker),* 2017. [17]

3.1 Fundamentos del seguimiento visual o *tracking*

Como se describe en [10], el problema de "*Object Tracking"* combina diferentes técnicas de procesamiento: Flujo óptico [18] (consistente en obtener el vector de movimiento de cada punto), filtros de Kalman (se usan para predecir la localización del objetivo en movimiento), *meanshift* y *camshift* [19] (algoritmos que buscan localizar el máximo de una función de densidad), entre otros.

Estas técnicas son complejas y el objetivo no es analizarlas a bajo nivel, pero sí merece la pena mencionarlas ya que principalmente todos los algoritmos de *Tracking* de **OpenCV** se basan en ellas y sientan las bases para resolver este problema.

Es importante realizar una distinción entre seguimiento visual o *tracking* y detección, ya que no tienen exactamente el mismo objetivo y pueden confundirse. Los algoritmos de detección tienen como objetivo detectar en una imagen objetos o entidades que satisfagan una serie de características, como por ejemplo algoritmos de reconocimiento facial, de personas o de cualquier otro elemento, que suelen estar basados en redes neuronales complejas para asegurar una gran precisión. Sin embargo, este tipo de algoritmos suele ser más lento que los de *tracking* lo cual los aleja como posible método para identificar objetivos a lo largo de las imágenes. Aún más determinante para entender por qué surgen los algoritmos de *tracking* frente a estos, es que los de detección no son capaces de mantener una identidad del objetivo, es decir, si se pretende seguir varios objetivos del mismo tipo o clase, un algoritmo de detección no permitiría distinguir cuál es cuál, cosa que sí es posible con los de *tracking.*

En multitud de aplicaciones se hace un uso conjunto de ambos tipos de algoritmo para lograr alta fiabilidad y precisión. Sin embargo, el número de algoritmos de detección es mucho más abundante y muy dependientes de la aplicación que resuelva, luego su revisión o mención queda fuera del ámbito de estudio de este trabajo.

3.2 Fundamentos del algoritmo elegido – KCF (*Kernelized Correlation Filter*)

3.2.1 Conceptos generales

Al ser el algoritmo elegido para este trabajo y uno de los que mejores resultados ofrece en tareas de *tracking,* es interesante, hacer una revisión rápida de las bases de este algoritmo. Debido a su elevada complejidad, se apoya la explicación en los artículos [16] y [20].

Aunque ya se definió anteriormente, el problema de *tracking* se puede considerar alternativamente como la tarea de "encontrar la mínima distancia desde el objeto seguido hasta el subespacio representado por los resultados previos del proceso de *tracking"*. El siguiente diagrama refleja el esquema típico de un algoritmo de *tracking visual*:

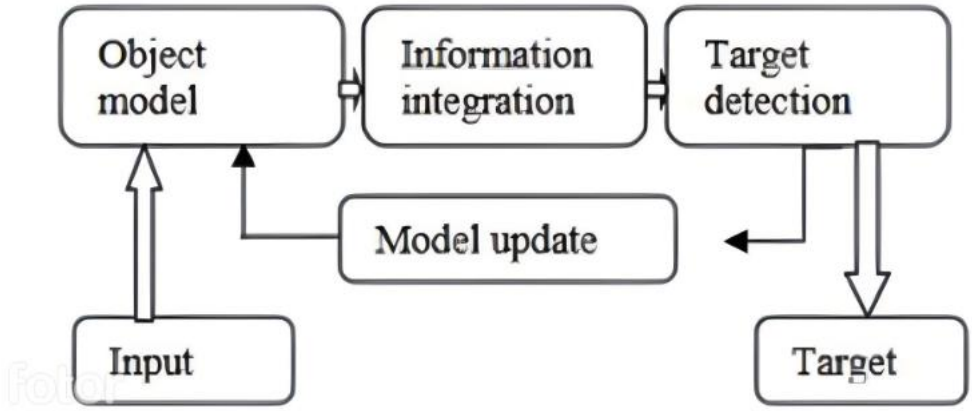


Figura 3-1. Diagrama de bloques genérico de un algoritmo de *tracking visual* [20].

Típicamente, los filtros basados en correlación requieren de un gran número de muestras, consituyendo gran carga computacional y creando conflicto con los requerimientos de tiempo real. KCF toma ventaja de la estructura circulante del formato matricial de los datos para expandirlo usando conceptos matriciales y lograr un incremento significativo en la velocidad de cómputo del *tracker*.

La gran ventaja viene del hecho de que este tracker puede aprender a partir de un solo *frame* y seguir adaptándose a partir del siguiente. De ahí en adelante, opera en el dominio de Fourier para lograr dicha eficiencia superior en cuanto a rendimiento.

Para no hacerlo muy extenso se recogerán ciertos procesos, elementos y técnicas que están integradas en la implementación de este *tracker*, explicados en mayor detalle en [16] y [20]:

- *Regresión lineal.*
- *Matrices circulantes.*
- *Kernel Trick,* que busca encontrar una función que transforme los puntos de datos en un espacio de características que sea linealmente separable en una dimensión mayor, muy eficiente para aplicaciones como *tracking visual.*

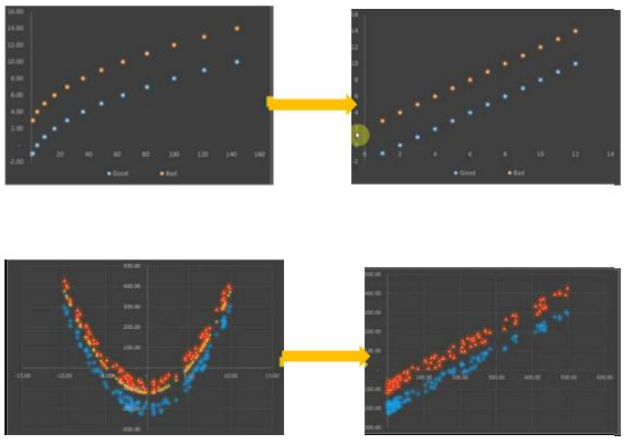


Figura 3-2. Visualización de la idea de *Kernel Trick* para lograr separabilidad lineal en las características [20].

- *Transformación espacial de una imagen.*
- *Transformada de Fourier de una imagen.*
- *Descriptores de características.*
- *Convolución.*
- *Convolución circular.*
- *Filtros de correlación.*

Tras haber mencionado el conjunto de técnicas y conceptos en los que se basa este tracker KCF, podemos pasar a dar una descripción breve con la que arrojar algo de luz sobre el funcionamiento de un algoritmo altamente complejo.

3.2.2 Explicación particular del algoritmo *Kernelized Correlation Filter* (KCF)

Este algoritmo KCF parte de una posición central y un tamaño de ventana del parche (*patch*) de interés, más conocido en este campo como *región de interés,* o ROI (*Region of Interest*), situado en algún lugar de la imagen. Una vez definido este parche, deben extraerse del mismo ciertas características, como pueden ser los píxeles en bruto u otras características derivadas de las anteriores y utilizarlas como datos representativos para localizar al mismo parche en fotogramas inmediatamente posteriores, previsiblemente en una posición diferente.

En una primera versión del algoritmo [21], los autores proponían como características a usar, efectivamente, las intensidades píxel a píxel resultantes de la conversión de la imagen entrante RGB a su correspondiente versión en escala de grises. En la versión mejorada del método [16][22], sin embargo, se proponía utilizar una serie de características más sofisticadas, con el objetivo de captar la forma y textura del contenido de la región. Además, estas características exhiben una cualidad de circularidad que podrá ser explotada apropiadamente por el algoritmo. Éste es el caso de las denominadas características HOG (*Histogram of Oriented Gradients*), de la cual se ofrece una sencilla descripción en el siguiente Subapartado 3.2.3.

En ambas versiones citadas del algoritmo, la elevada dimensionalidad de los datos que se manejan puede convertir el problema en una tarea muy exigente, desde el punto de vista computacional, lo que podría comprometer su rendimiento en sus aplicaciones típicas de tiempo real que requieren un rendimiento temporal altamente exigente y restrictivo. Para solventar este problema, KCF realiza principalmente, entre otras cosas que también influyen, una transformación del espacio de características original a otra representación de menor dimensión, pero manteniendo las propiedades requeridas de la representación original.

A pesar del reto que supone gestionar tan elevado número de datos, el revolucionador éxito de KCF recae en ser capaz de agilizar sustancialmente el proceso de aprendizaje gracias a la exploración e integración de herramientas previamente no tan implicadas de manera conjunta como son el denominado *kernel trick,* las propiedades de las *matrices circulantes* y la *transformada de Fourier* directa e inversa (DFT e IFT, respectivamente).

Sin entrar en muchos de los detalles de la implementación del algoritmo, los cuales son ciertamente complejos, podemos indicar que KCF formula el problema de seguimiento (*tracking*) como una tarea de clasificación binaria definida en línea[1], con el objeto de distinguir el parche objetivo de cualquier otro parche de fondo que pueda componerse de la imagen recibida, utilizando un discriminante lineal, como el *Linear Ridge Regressor* (LRR). Acorde a esto, sólo se considerarán dos clases: una correspondiente al verdadero parche objetivo y otra correspondiente a cualquier otro parche que no lo sea. El conjunto de muestras se compondrá, por tanto, mayoritariamente de muestras "negativas", es decir, muestras diferentes al parche objetivo, generadas automáticamente a partir de éste, por el principio de matrices circulantes, que es una forma de generar variantes del patrón original mediante desplazamientos cíclicos.

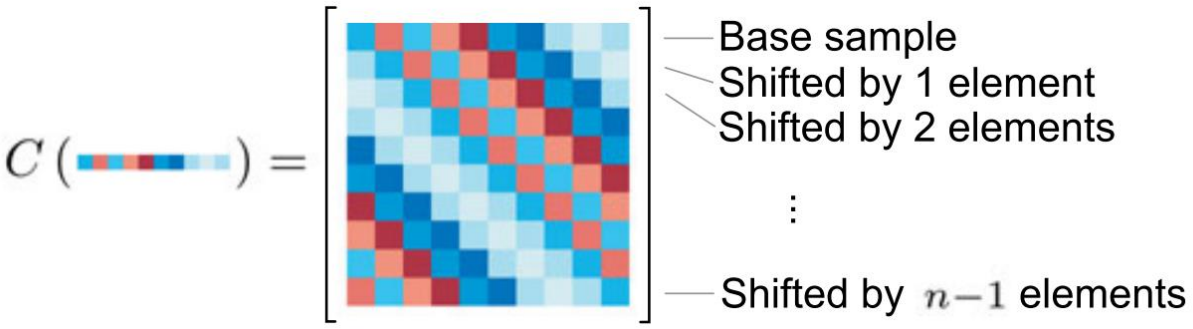


Figura 3-3. Ilustración de una matriz circulante. Las filas corresponden a desplazamientos cíclicos del vector de una imagen, o sus translaciones en 1D. Las mismas propiedades se expanden a matrices circulantes describiendo imágenes 2D [16].

Esas variantes constituirán, por tanto, patrones virtuales "negativos" que serán usados para el entrenamiento del reconocedor.

Como se ha comentado, LRR es un clasificador lineal; tiene la gran ventaja de proporcionar una solución analítica, al tiempo que permite obtener resultados comparables a los de clasificadores más sofisticados, como las *Máquinas de Soporte Vectorial* [2](SVM). En su forma original, LRR se formula como un problema de minimización, del siguiente modo:

$$\min_{\omega} \sum_{i=1}^{n} (\omega^T x_i - y_i)^2 + \lambda \|\omega\|^2 \qquad (3\text{-}1)$$

Donde $x_i \, \epsilon \mathbb{R}^m \; \forall i = 1 \ldots n$ es cada uno de los patrones muestra (en nuestro caso, la representación dada por el conjunto de características, bien del parche objetivo, bien de alguno de los parches "negativos"), con n muestras. Como se indica, cada patrón viene dado por un vector de características de dimensión m.

$\boldsymbol{\omega}$ es el vector de pesos a determinar como resultado de la minimización (aprendizaje). Por su parte, λ es un parámetro de regularización que permite controlar el sobre-aprendizaje (*overfitting*), mientras que y_i es el valor elegido como indicador de la clase a la que pertenece el patrón, por ejemplo $y_i = 1$ si $\boldsymbol{x_i}$ es el parche objetivo, o bien $y_i = -1$ si $\boldsymbol{x_i}$ es parche "negativo".

[1] El aprendizaje automático en línea es una técnica que permite a un modelo de inteligencia artificial aprender continuamente y adaptarse a nuevos datos en tiempo real, en lugar de requerir entrenamiento por lotes en conjuntos de datos estáticos.

[2] Una Máquina de Soporte Vectorial (SVM, por sus siglas en inglés, Support Vector Machine) es un algoritmo de aprendizaje automático supervisado que se utiliza tanto para tareas de clasificación como de regresión. La SVM busca encontrar el hiperplano óptimo en un espacio multidimensional que mejor separa las distintas clases de datos, maximizando el margen entre ellas.

La solución a este problema, sin la intermediación de ningún espacio de características alternativo, sería la siguiente:

$$\boldsymbol{\omega} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I})^{-1}\,\boldsymbol{X}^T\boldsymbol{y} \qquad (3\text{-}2)$$

Siendo **I** la matriz identidad de dimensiones apropiadas, **X** la matriz formada por las muestras e **y** el vector formado por los indicadores de la clase:

$$\boldsymbol{X} = \begin{bmatrix} x_1^T \\ x_2^T \\ \dots \\ x_n^T \end{bmatrix} \quad , \quad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}$$

Típicamente, en un problema de clasificación convencional, se dispone de un número de muestras muy superior al número de características usadas para representar éstas. No es así en este caso. Es por ello que se resuelve el problema de clasificación pasando por un espacio dual que requerirá, en lugar de invertir una matriz de datos *m x m,* como en la expresión (3-2), siendo *m* el número de características, invertir una matriz de datos transformados de dimensión *n x n* siendo *n* el número de muestras.

En general, LRR pretende encontrar una solución $\boldsymbol{\omega}$ que defina un hiperplano que actúe como separador lineal de ambas clases.

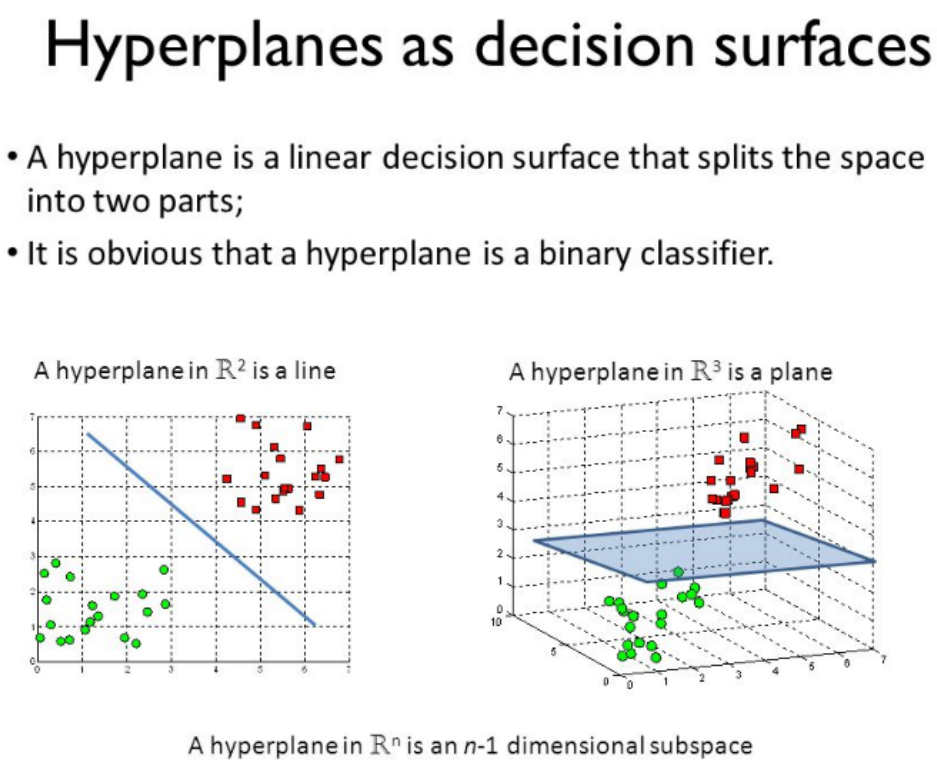


Figura 3-4. Descripción de hiperplano como clasificador binario; Fuente: https://medium.com/@csarchiquerodriguez/maquina-de-soporte-vectorial-svm-92e9f1b1b1ac.

Sin embargo, siendo HOG la representación preferida para éste y muchos otros problemas de reconocimiento, hay que tener en cuenta que es una representación de naturaleza no lineal. La separación lineal se conseguirá traduciendo el espacio de características original a un nuevo espacio de características en el que los datos sí sean linealmente separables, de manera similar a la que se veía en la **Figura 3-2.** Pero ello conllevaría, como efecto colateral, el aumento de la dimensión de los datos. Sin embargo, esto no supondrá un problema en este caso, gracias al denominado *kernel trick.*

La citada transformación se realizará mediante la introducción de la llamada *función kernel*, de la que deriva el nombre del método. El *kernel trick* consiste en que la *función kernel* permitirá evaluar la correlación entre los vectores transformados sin necesidad de realizar en la práctica dicha transformación, evitando de este modo la necesidad de transitar efectivamente por el espacio de dimensionalidad superior. La estructura del *kernel* (máscara de correlación) subyacente para ese fin puede ser, por ejemplo, una máscara gaussiana, la cual ofrece ciertas ventajas matemáticas.

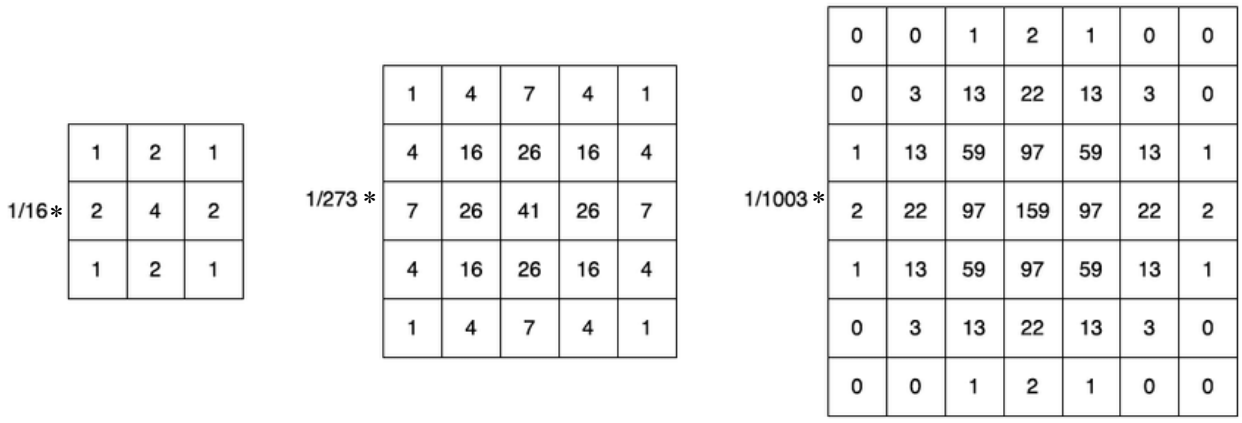


Figura 3-5. Ejemplos de aproximación discretizada para *kernels* gaussianos de 3x3, 5x5 y 7x7; Fuente:https://www.researchgate.net/figure/Discrete-approximation-of-the-Gaussian-kernels-3x3-5x5-7x7_fig2_325768087.

Un hecho interesante del uso de matrices circulantes es que éstas se convierten en diagonales a través de la transformada discreta de Fourier. Esto a su vez, permitirá que la resolución de la ecuación (3-2) se pueda realizar a nivel de componentes individuales, en lugar de realizarse de forma matricial.

Para la implementación del método, se ha recurrido a su correspondiente algoritmo dentro de la API de tracking en la librería ***OpenCV***, recogida en la siguiente documentación tanto en *Python* como en *C++*, así como en otros recursos online:

- Descripción de la función (C++): https://docs.opencv.org/4.x/d2/dff/classcv_1_1TrackerKCF.html.
- Tutorial básico de uso (Python): https://learnopencv.com/object-tracking-using-opencv-cpp-python/.

Como este proyecto ha sido basado en Python, se van a presentar a continuación de manera breve las funciones que inicializan y aplican el algoritmo KCF descrito disponible en la librería mencionada. Aunque no suele surgir este problema, es importante notar que, según la versión de OpenCV instalada, se accede a la función mediante el prefijo "cv2.legacy" o directamente "cv2.", pero el nombre del método en sí no cambia entre versiones.

- ***tracker_object = cv2.legacy.TrackerKCF_create()*** → Crea una instancia de objeto tracker con algoritmo KCF.
- ***tracker_object.init(tracker_frame, tuple(bbox))*** → Inicializa al objeto tracker con una imagen inicial y una *bounding-box* que defina al parche a identificar en sucesivos *frames*. Es importante para mejorar el rendimiento del algoritmo, que dicha *bounding-box* no sea la que encuadra exclusivamente al objetivo, sino que incluya parte de su vecindad para facilitar la posterior identificación del parche.
- ***tracker_object.update(raw_frame)*** → Una vez inicializado el algoritmo, se actualiza con nuevas imágenes, siendo importante que dichas imágenes sean directamente los frames originales proporcionados por la cámara. Si recibiese una imagen con elementos superpuestos por procesamiento, el algoritmo perdería su coherencia y su precisión.

Para profundizar en esta breve descripción del método y sus bases matemáticas, se vuelve a citar los artículos [16][21][22] que lo describen con mayor detalle.

3.2.3 Descripción de características HOG (*Histogram of Oriented Gradients*)

Como se ha mencionado en el Subapartado 3.2.2, el método que utiliza KCF para extraer características es la técnica HOG, por sus siglas en inglés. Al ser una parte fundamental que conforma el esquema de implementación de KCF, es también merecedor de una descripción propia.

Es importante comenzar aclarando que HOG consiste en un *feature descriptor*, que se trata de una representación simplificada de una imagen que contiene únicamente la información más importante acerca de la imagen. Hay que notar que hay otros algoritmos con la misma función pero que generarán representaciones diferentes según la naturaleza de dichos algoritmos, algunos ejemplos de otras técnicas de este tipo son SIFT (*Scale Invariant Feature Transform*), SURF (*Speeded-Up Robust Feature*), ORB (*Oriented Fast and Rotated Brief*), entre otros.

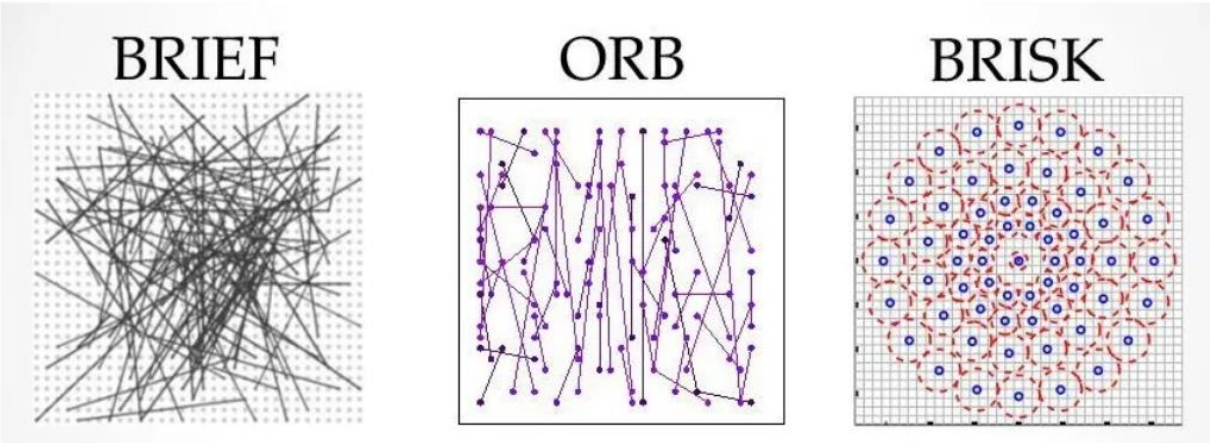


Figura 3-6. Ejemplos de descripción de diferentes algoritmos de *feature descriptors*;
Fuente: https://medium.com/data-breach/introduction-to-feature-detection-and-matching-65e27179885d.

Para concretar, el proceso de descripción realizado por estos algoritmos busca típicamente describir la "apariencia" local en torno al punto de interés o punto característico de tal manera que, idealmente, sea invariante a cambios de iluminación, translación, escala y rotación. Generalmente, estos algoritmos producirán un vector con la descripción para cada punto característico.

La idea, lógicamente, es que sea relativamente fácil emparejar puntos característicos entre dos imágenes sucesivas y así mantener una especie de continuidad en dichos puntos característicos. Esto se aplica de la misma forma para el parche que describíamos en el subapartado anterior junto con otras técnicas.

Volviendo a HOG, se trata de una técnica para generar descripción muy popular en visión por computador. En rasgos generales, analiza la distribución de las orientaciones de los bordes dentro de un objeto para describir su forma y apariencia. Este método involucra calcular la magnitud del gradiente y la orientación para cada píxel en una imagen y luego subdividir la imagen en celdas de pequeño tamaño.

Algunas de las ventajas o particularidades del descriptor HOG que lo hacen preferible en ciertos contextos sobre otros algoritmos son:

- Se centra en la estructura o forma de un objeto. Mientras otros algoritmos sólo determinan por ejemplo si un píxel es borde o no de un objeto, HOG va más allá y describe su dirección también.
- Estas orientaciones se calculan para subregiones definidas de la imagen, abarcando la imagen completa.
- Finalmente, HOG genera un histograma para cada una de estas subregiones de manera independiente.

Aunque con lo ya introducido se podría tener una idea de cómo funciona HOG, se puede plantear un ejemplo de los pasos de procesado que implica en una imagen simple para tener una ilustración visual de cómo "describe" cierta imagen [23].

Consideramos la imagen de la **Figura 3-7**, con dimensiones 180 x 280.

Figura 3-7. Imagen inicial sobre la que se va a aplicar HOG.

En primer lugar, se requiere un preprocesado de la imagen, de manera que se rebaje el ratio ancho-alto a 1:2.

Para simplificar las operaciones, ya que se aplicarán parches de *8 x 8* y *16 x 16* píxeles, se redimensiona la imagen a *64 x 128*.

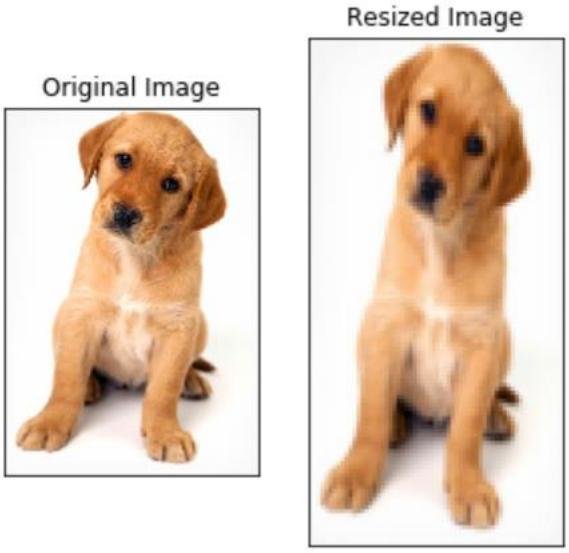


Figura 3-8. Imagen inicial (*180 x 280*) junto a imagen redimensionada (*64 x 128*).

A continuación, se calcula el gradiente para cada píxel en la imagen. Como ya se ha mencionado, se escogen parches de la imagen y se calculan los gradientes para dicho parche.

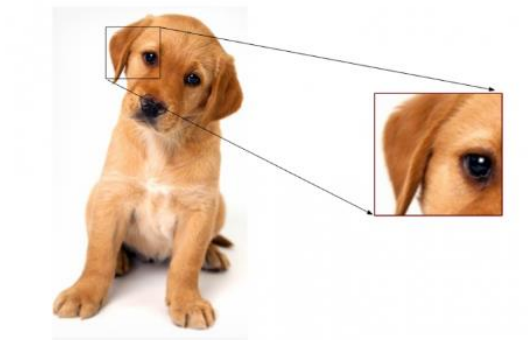


Figura 3-9. Parche extraído de la imagen redimensionada sobre la que calcular el gradiente.

Suponiendo que se extrae una matriz de intensidad de píxeles del parche del siguiente tipo:

121	10	78	96	125
48	152	68	125	111
145	78	85	89	65
154	214	56	200	66
214	87	45	102	45

Figura 3-10. Intensidad de píxeles de un parche ficticio equivalente al que se extraería de cierta imagen.

Notar que para el cálculo del gradiente se necesita tener un valor a la "izquierda" y otro a la "derecha" así como uno "arriba" y "debajo", luego no se podría calcular el gradiente para los píxeles del extremo del parche.

Las expresiones para calcular la magnitud y orientación numéricas del gradiente son las siguientes:

$$G_M = \sqrt{g_x^2 + g_y^2} \quad , \quad G_{dir} = atan2(g_y, g_x) \tag{3-3}$$

Donde, para el ejemplo del píxel destacado en rojo en la **Figura 3-10** tendríamos $g_x = 89 - 78 = 9$ y $g_y = 68 - 56 = 8$.

A la hora de construir los histogramas usando la magnitud y orientación de los gradientes, se va a comentar brevemente cómo se genera el histograma en HOG.

Se genera una tabla con 9 "bins" separados de 20 en 20 y en función del valor de la orientación y la magnitud, se da un peso mayor al bin cuya orientación esté más próximo:

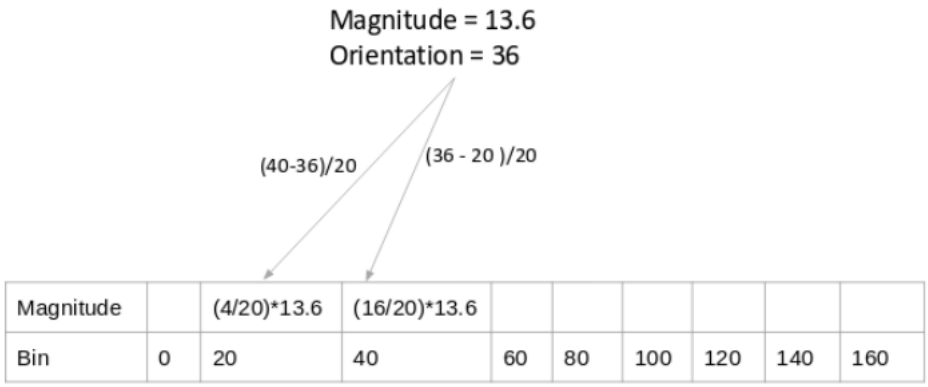


Figura 3-11. Construcción del histograma en HOG.

Sin embargo, los histogramas no se generan directamente para la imagen completa, sino que se divide la misma en celdas de 8x8, 16x16, 32x32, etc.

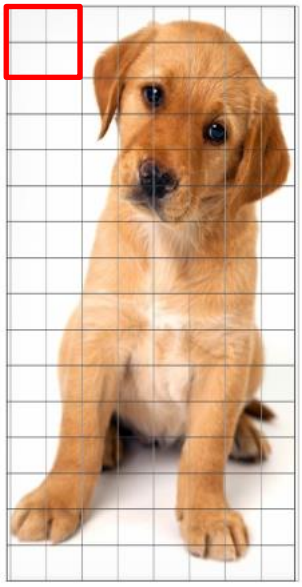


Figura 3-12. Imagen subdividida en celdas de 8x8.

Por último, para reducir el efecto de los cambios de luminosidad entre parches, se normalizan los gradientes tomando, en este caso, bloques de 16x16, que equivaldrían al cuadrado rojo en la **Figura 3-12.**

Cada celda 8x8 tiene una matriz 9x1 (había 9 bins) por histograma, luego para cada bloque 16x16 se tendrían 4 matrices 9x1, o bien una única matriz 36x1. Para normalizar esta última, se divide cada valor por la raíz cuadrada de la suma de todos los valores al cuadrado. Es decir, para cierto vector V:

$$V = [a_0, a_1, a_2, a_3, \dots, a_{35}] \quad (3\text{-}4)$$

Se calcula la raíz cuadrada de sus valores al cuadrado:

$$k = \sqrt{a_0^2 + a_1^2 + a_2^2 + \cdots + a_{35}^2} \quad (3\text{-}5)$$

Y se dividen todos los valores del vector por *k:*

$$V_{normalizado} = [\frac{a_0}{k}, \frac{a_1}{k}, \frac{a_2}{k}, \dots, \frac{a_{35}}{k}] \quad (3\text{-}6)$$

Finalmente, para este caso, tendremos 105 (7x15) bloques de 16x16 píxeles. Cada uno de estos 105 bloques tiene un vector de 36x1 como características. Luego para esta imagen, el número total de características sería 105x36x1 = 3780 características.

Aunque en este caso se han escogido bloques de 16x16, podría interesar para imágenes con mayor resolución tomar bloques de mayor tamaño como 32x32 o 64x64 píxeles, teniendo en cuenta lógicamente que van a cambiar las dimensiones de los vectores, pero proporcionando una descripción relativamente similar. También puede darse el efecto de que al aumentar el tamaño de este parche se capture mejor la información espacial. Según el tamaño de los objetos a describir en la imagen, aumentarlo puede provocar pérdida de información en aquellos objetos de menor tamaño, luego la mejor alternativa sería evaluar diferentes tamaños de parche y ver cuál de ellos da mejores resultados para cierta aplicación.

Se deja para terminar, la siguiente figura para visualizar el resultado de HOG en esta imagen:

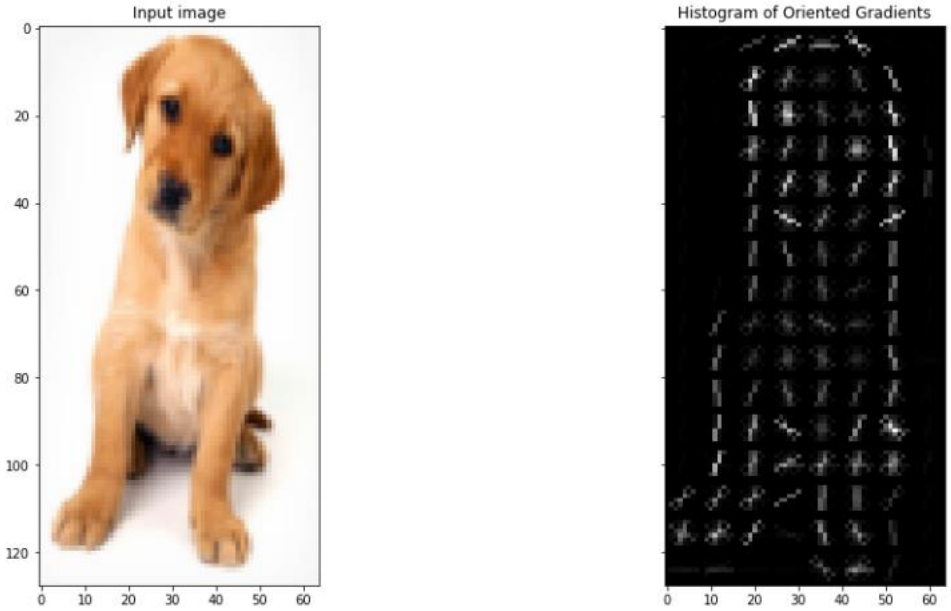


Figura 3-13. Resultado de aplicar HOG a la imagen que se propone.

3.2.4 Conclusiones

Como se concluye en el capítulo 2 de [10], KCF mantiene una buena tasa de FPS, la cual es inferior a MOSSE, pero es más equilibrado que este y responde bien en multitud de situaciones. La detección de la pérdida del objetivo es suficientemente precisa y permitirá identificar cuándo ha habido incapacidad de seguir al objetivo en lugar de apuntar a lugares erróneos como ocurre con otros *trackers*.

Es importante mencionar que el hecho de tener una gran cantidad de trackers ejecutándose en cada *frame* puede suponer una carga computacional elevada.

Esto implica que pueda ser interesante hacer uso del *tracker* MOSSE para ciertas situaciones, agilizando el procesamiento y obteniendo resultados similares. Esto se debe a que MOSSE proporciona una tasa de FPS de en torno al orden de 10 veces superior a KCF.

Por tanto, según la complejidad de la aplicación o las limitaciones de hardware, puede ser más que interesante hacer uso único de MOSSE en lugar de KCF, o bien hacer uso de KCF para las imágenes que enfrenten situaciones complejas como oclusión y MOSSE para las imágenes donde la cámara orientable ya tiene mediante control centrado el objetivo de interés en la imagen, y por tanto, el algoritmo de *tracking* tendrá menos dificultades a la hora de seguirlo.

4 PROBLEMAS VISUALES Y GEOMÉTRICOS

Es necesario realizar un análisis en profundidad de la geometría de la escena y de las necesidades requeridas del análisis de la imagen para entender, resolver e implementar los objetivos descritos en el primer capítulo. Podemos separar los problemas a abordar en 3 categorías:

- Tareas de visión por computador para poder identificar los objetivos a *trackear* respecto del fondo de la imagen.
- Problemas derivados del apuntamiento de la cámara, que involucran la geometría relativa entre los diferentes elementos del sistema, aplicación de los módelos de cámaras y geometría óptica para conocer la posición tridimensional de los objetivos a partir de la imagen proporcionada por la cámara, así como ser capaces de realizar un reajuste fino del apuntamiento inicial y trabajar con el zoom de las cámaras para tener una mejor imagen del objetivo asignado a cierta cámara.
- Problemas relacionados con las técnicas de gestión de objetivos, como por ejemplo dividir el conjunto de objetivos en diferentes agrupaciones.

4.1 Identificación de objetivos

En los pasos a seguir para implementar el sistema planteado, es lógico que la capacidad de identificar a los objetivos en la escena es el primer problema a plantear y resolver. Por una parte, se requiere de una segmentación de la imagen que separe a los objetivos del fondo de la imagen y por otra parte realizar el seguimiento visual de dichos objetivos, siendo capaces de mantener la identidad de los mismos en el tiempo, es decir, que se pueda distinguir el objetivo a lo largo del paso de los *frames* proporcionados por la cámara.

En el capítulo anterior se concluyó eligiendo KCF como el algoritmo de seguimiento visual que nos permitirá identificar y encuadrar a cierto objetivo a lo largo del tiempo. Sin embargo, es importante notar que en esta aplicación se requiere de *multitracking*[3] para la cámara gran angular, por tanto, se podría plantear el uso de múltiples instancias de *trackers* KCF para cada objetivo. Sin embargo, [24] OpenCV proporciona una implentación de multitracker que gestiona esta operación de manera más eficiente, pudiendo usarse cualquiera de los algoritmos de *tracking* individuales mencionados anteriormente.

Con esto se define la solución del segundo problema descrito, sin embargo, los algoritmos de *tracking* requieren de ser inicializados con sus *bounding box* iniciales. Para obtener dichas ventanas iniciales que encuadran los diferentes elementos que aparecen, debemos primero poder reconocer todos los objetivos de la escena respecto del fondo. Tras ser segmentados, los algoritmos de *tracking* se iniciarán en los objetivos que interesen.

Por su parte, hay que plantear la resolución de la segmentación de los objetivos, lo cual, aunque pueda parecer trivial, está lejos de serlo, ya que existen multitud de posibles escenarios con multitud de posibles "definiciones" de objetivos.

Este proceso debe dar como resultado las coordenadas en píxeles de los centros de los objetivos con los que se va a trabajar.

Es importante mencionar que además de servir como inicialización para los algoritmos de *tracking*, esta segmentación será usada a lo largo del tiempo conjuntamente con los algoritmos de seguimiento visual para afinar su resultado. La razón de esto se debe a que la *bounding box* de los *trackers* tiene un tamaño mayor y el centro de esta normalmente no coincidirá exactamente con el centro del objetivo que está siguiendo.

[3] Técnica de seguimiento de objetos que permite detectar y rastrear simultáneamente múltiples objetos en una escena, utilizando algoritmos y técnicas de proce
samiento de imágenes.

Lo que se propone es tomar las coordenadas resultado del centro de la *bounding box* del objetivo *trackeado* y todas las coordenadas de los objetivos etiquetados durante la segmentación. Se elige de estos últimos el centro que esté más cerca del centro de la *bounding box* del algoritmo de *tracking*.

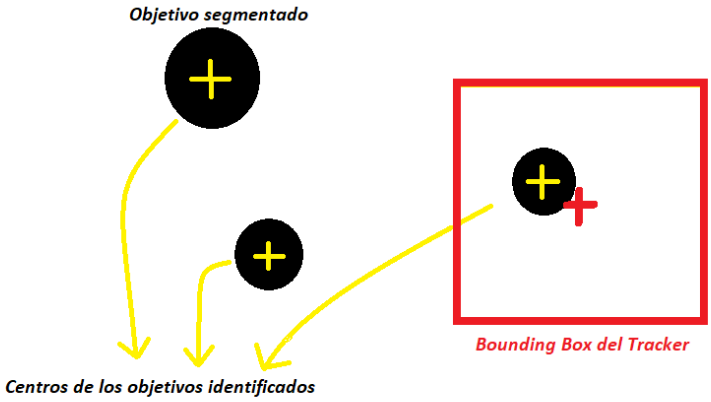


Figura 4-1. Segmentación y seguimiento para el caso de objetivo y *bounding box* del tracker descentrados.

La utilidad de dotar de identidad a cada objetivo que lo haga distinguible del resto e identificable en sucesivos instantes es fundamental. Por tanto, hay que poner en contexto qué objetivos han de ser sometidos a este proceso completo.

En la cámara gran angular deben ser identificados a lo largo del tiempo aquellos objetivos o agrupaciones de objetivos que hayan sido asignados a las cámaras orientables. Esto es necesario para poder realizar el apuntamiento inicial de las cámaras orientables, reinicios en caso de fallo, intercambio de objetivos, etc.

En cuanto a las cámaras orientables, se debe identificar el objetivo o *cluster*[4] que le ha sido asignado a sí misma, siendo capaz de distinguir el objetivo que le corresponde con respecto al resto que puedan aparecer también en la imagen.

El proceso de segmentación es generalmente complejo, ya que los escenarios suelen ser de carácter heterogéneo, es decir, además de los objetivos, puede haber irregularidades en el terreno, árboles, edificios o cualquier otro elemento que conforme un reto para las técnicas de detección a la hora de identificar a lo que se define como objetivo. En este caso se va a realizar una simplificación respecto a esto consistente en considerar un fondo homogéneo que predomine en proporción de píxeles en la imagen sobre los objetivos. Esto podría ser directamente válido para aplicaciones donde el fondo sea relativamente constante y sin muchos elementos o irregularidades debido a la asunción de mundo plano que se toma, como por ejemplo agricultura, monitoreo de infraestructuras, etc.

Se presenta a continuación en la **Figura 4-2** una imagen representativa, obtenida mediante la cámara angular del UAV a cierta altura, de esta tarea con el fondo y objetivos simulados.

En dicha figura, los objetivos simulados corresponden a cuboides estáticos (sin desplazamiento durante la simulación) de diferentes tamaños, tanto en altura, como en largo y ancho, así como colores.

[4] Grupo de elementos similares que están cercanos entre sí y se comportan como una entidad única.

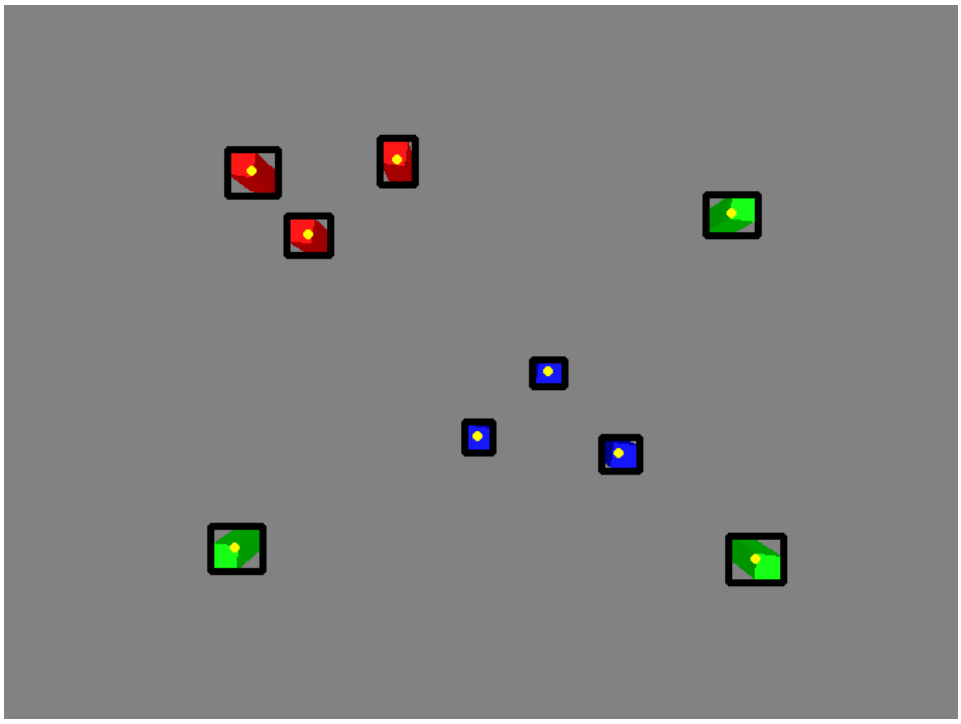


Figura 4-2. Imagen de cámara gran angular con superposición de bounding boxes y centroides de objetivos (las sombras que se aprecian son las caras de los cubos).

Se puede observar como en negro se identifican las bounding boxes de cada objetivo identificado en el proceso de segmentación. Estas serán las que, para los objetivos que interesen y ampliadas en tamaño (cómo se veía en el parche extraído en el capítulo 3) constituirán la inicialización del algoritmo de *tracking* KCF. Por su parte, en cada objetivo se muestra con un círculo amarillo el centroide del objetivo identificado (correspondiente a las crucetas amarillas en la **Figura 4-1**).

Sin embargo, aunque en este proyecto se supone un fondo homogéneo, el sistema sería relativamente fácil aplicarlo a situaciones más complejas, en el sentido de entorno irregular y con otros elementos que podrían asemejarse a los objetivos deseados, dificultando la tarea de discernir entre fondo y otros elementos y objetivos deseados. Para ello habría que recurrir a técnicas más avanzadas o más específicas en el ámbito de la visión por computador.

Aún así, los innovadores avances en el campo de *Deep Learning* [5] permiten obtener resultados sorprendentes en este ámbito de detección de objetivos en imágenes. Hay multitud de redes neuronales capaces de identificar objetos comunes en escenas con diferentes objetos y gran variabilidad. Sin embargo, hay que tener claro que cada problema tiene sus particularidades y es posible que no exista una única técnica que se adecúe, sino más bien la combinación de varias que logre mejores resultados. Es importante además plantear la pregunta de qué define al objetivo, por ejemplo, ¿qué hace distinguible a una motocicleta respecto de otros vehículos en una autopista?

Los modelos que están actualmente en el estado del arte en detección de objetos en imágenes tienen una precisión de alrededor de un 60% mAP (*mean Average Precision*[6]), por lo cual, es posible que eso no sea suficiente para la situación en cuestión o que no haya modelos bien entrenados para el objetivo que queremos identificar. En este caso, se podría recurrir a una inicialización manual del usuario para el algoritmo de *tracking*, aunque en caso de que este algoritmo falle, el reinicio del mismo volvería a requerir del usuario, o bien desarrollar otras técnicas para recuperar al objetivo perdido.

[5] Técnica de aprendizaje automático que utiliza redes neuronales artificiales para aprender a partir de grandes cantidades de datos y realizar tareas complejas.

[6] Métrica de para estudiar la precisión de modelos de detección de objetos. Calcula la precisión media entre diferentes niveles de límites *Intersection-over-Union* (IoU) y entre múltiples categorías de objetos. Se estudian en datasets como COCO (Common Objects in Context).

Para las simplificaciones dadas, se va a plantear un algoritmo de segmentación que se explica a continuación.

4.1.1 Algoritmo de segmentación

El algoritmo de segmentación trabaja convirtiendo la imagen RGB recibida de la cámara al formato HSV [25] (*Hue, Saturation, Value*), teniendo una mayor separabilidad de las diferentes tonalidades de la imagen.

Para la imagen obtenida de la cámara gran angular, que es en la que se debe realizar esta identificación de objetivos, calcularemos el histograma de cada canal (H, S y V), donde debido al fondo homogéneo, habrá un máximo global fácilmente identificable. Para la imagen de la **Figura 4-2** donde los objetivos aparecen con poco tamaño, se obtienen los siguientes histogramas donde sus picos son prácticamente inapreciables respecto del pico del fondo.

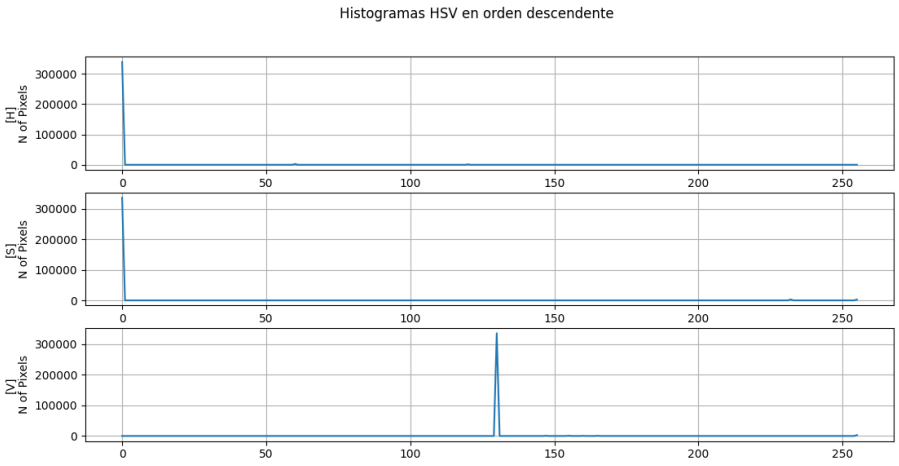


Figura 4-3. Histogramas HSV correspondientes a la imagen de la **figura 4-2**.

A diferencia de los cuboides de la **Figura 4-2**, sin mover la cámara, se incrementa considerablemente el tamaño y altura de los cuboides, así como se añaden algunos más.

Si consideramos ahora la siguiente **Figura 4-4** donde por la distribución descrita anteriormente se dará la situación de que ocupan mayor espacio en la imagen. El objetivo de esto es comprobar si el algoritmo es efectivo realizando la segmentación respecto del fondo homogéneo.

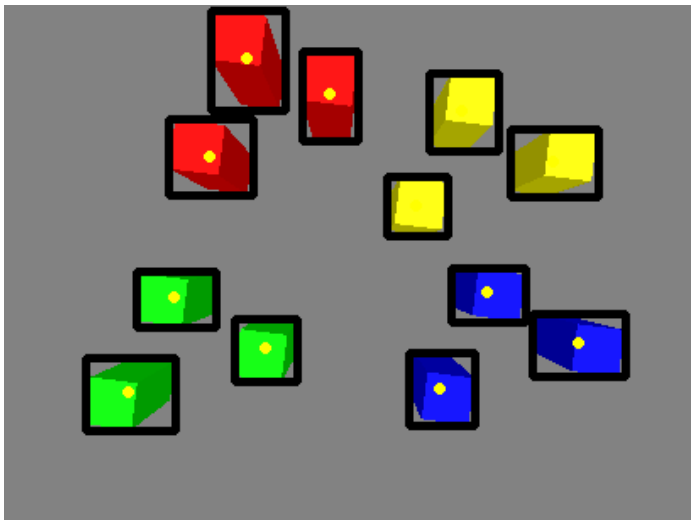


Figura 4-4. Imagen de cámara gran angular con mayor número de objetivos y más cercanos.

Analizando ahora sus histogramas HSV, vemos como claramente el fondo sigue siendo predominante, pero si serían apreciables los picos correspondientes a los objetivos de diferentes colores.

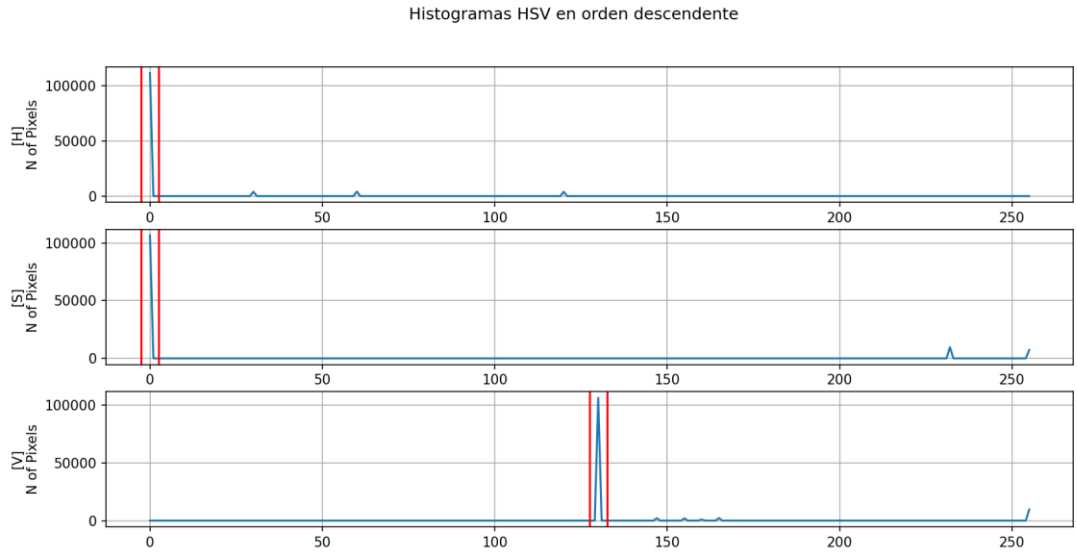


Figura 4-5. Histogramas HSV correspondientes a la imagen de la **figura 4-4**.

En cualquier caso, vemos cómo el máximo global es claramente correspondiente al fondo de la imagen, lo que ayudará a separarlo de los objetivos.

Para realizar esta separación, se define un ancho de supresión, que debe ajustarse al ancho del pico del fondo. Este ancho crecerá cuanto menos homogéneo sea el fondo, o bien cuanto más objetivos haya y más cercanos estén como se ha visto anteriormente. Una vez definido este ancho, se define un umbral a la derecha e izquierda del valor máximo del pico en el histograma de cada canal H, S y V, logrando aislar dicho pico (ver bandas rojas verticales en **Figura 4-5**).

De esta manera se puede generar una máscara binaria con los píxeles fuera de este pico del fondo en los 3 canales, logrando esa segmentación de los objetivos respecto del fondo de la imagen.

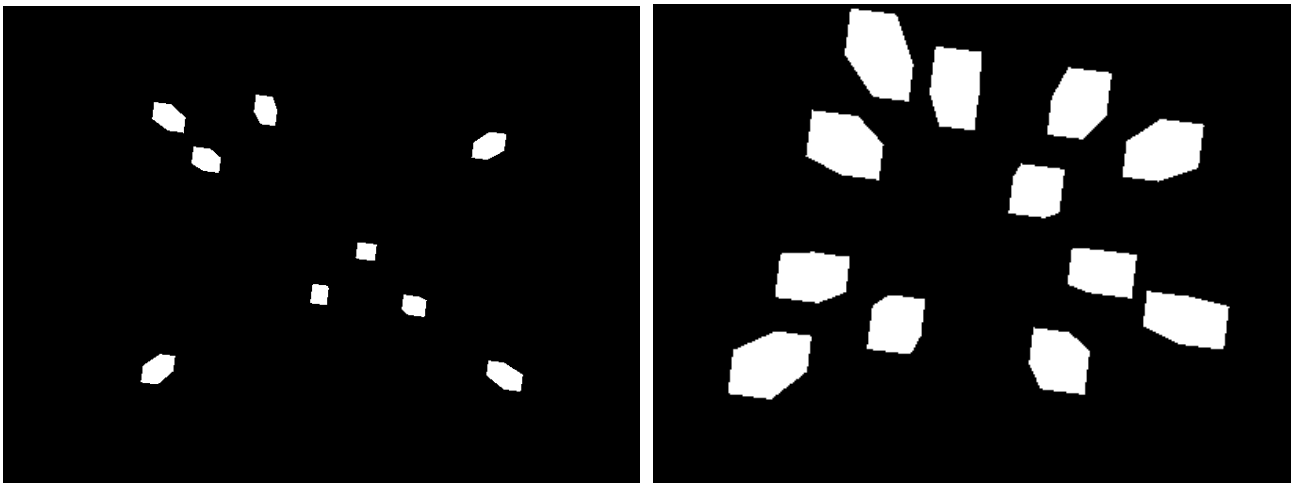
Podemos reafirmar esta idea viendo el **Código 4-1**, que implementa esta tarea.

Código 4.1 Segmentación de objetivos respecto del fondo.

```
# Ancho de supresión:
supress_width = 5

# Pasar a HSV
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

# Cálculos de Histogramas
hist_h = cv2.calcHist([img_hsv], [0], None, [256], [0, 256])
hist_s = cv2.calcHist([img_hsv], [1], None, [256], [0, 256])
hist_v = cv2.calcHist([img_hsv], [2], None, [256], [0, 256])

# Valores máximos
maxh = int(np.argmax(hist_h))
maxs = int(np.argmax(hist_s))
maxv = int(np.argmax(hist_v))

# Umbrales
hsv_min = (maxh-supress_width/2, maxs-supress_width/2, maxv-supress_width/2)
hsv_max = (maxh+supress_width/2, maxs+supress_width/2, maxv+supress_width/2)

# Máscara
mask = cv2.inRange(img_hsv, hsv_min, hsv_max)
mask = cv2.bitwise_not(mask)
```

Con esto se obtiene la máscara binaria buscada, que ya permite obtener los *bounding boxes* y centroides de cada objetivo de manera fácil, el cómo se verá más adelante.

Se muestran a continuación las máscaras resultantes para la imagen de la **Figura 4-2 y 4-4.**



a) b)

Figura 4-6. a) Máscara binaria de la imagen en la Figura 4-2**.** b) Máscara binaria de la imagen en la Figura 4-4

4.2 Apuntamiento de objetivos

Partimos ahora de que, tras aplicar el análisis del apartado anterior, tenemos las coordenadas en píxeles de los centroides de los objetivos.

El sistema tendrá una serie de sistemas de referencia representados en el esquema de la **Figura 4-7**, para el caso de disponer de dos cámaras orientables.

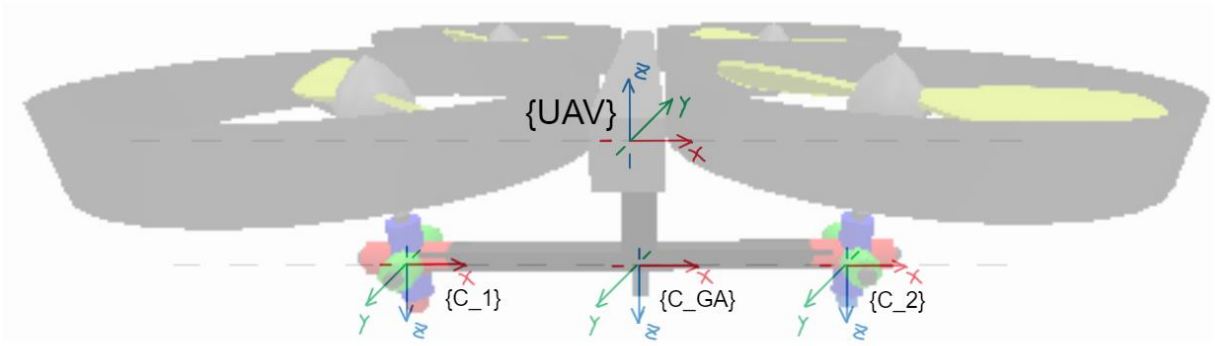


Figura 4-7. Esquema representativo de los sistemas de referencia del sistema con dos cámaras orientables.

Se va a plantear el apuntamiento y gestión de objetivos individuales, pero será equivalentemente aplicable a un punto que represente la agrupación de ellos, como sería el centroide del *cluster*. Los ángulos de los apartados sucesivos serán referencias para los controladores de los capítulos posteriores.

4.2.1 Ángulos a girar por los ejes del gimbal a partir de la imagen de la cámara Gran Angular

Para obtener estos ángulos, el primer paso es obtener la posición tridimensional de los puntos de interés. Se parte de los centros en píxeles de los objetivos vistos en la imagen de la cámara gran angular y se realiza la proyección inversa.

Se considera conocida la altura del dron, ya que es estimable con un altímetro, cuya medida suele tener un cierto error [26] que será incluido en la implementación.

Debemos tener las coordenadas en píxeles de cierto centro en cuestión *[u, v]* y la matriz de parámetros intrínsecos, que tiene la siguiente forma y es calculable a partir de los parámetros reales de la cámara:

$$K = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Donde:

- $f_x = f/\rho_x$: longitud focal efectiva horizontal [pix]
- $f_y = f/\rho_y$: longitud focal efectiva vertical [pix]
  - $\rho_x = \frac{w\ (ancho\ sensor\ [m])}{N(Nº\ píxeles\ eje\ horizontal)}$
  - $\rho_y = \frac{h\ (alto\ sensor\ [m])}{M(Nº\ píxeles\ eje\ vertical)}$
  - $f = distancia\ focal\ de\ la\ cámara$
- (u_0, v_0): coordenadas del punto principal [pix]

Disponiendo de ambos, por geometría, conocemos la recta a la que debe pertenecer el punto tridimensional P_j.

$$P_j \propto K^{-1} \cdot \tilde{p}_j \quad ; \quad \tilde{p}_j = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{4-1}$$

Introduciendo la siguiente notación, queda definido y resuelto el problema de proyección inversa, ya que Z_j es la distancia de la cámara al objetivo, que se considera conocido gracias al altímetro previamente mencionado.

$$\tilde{m}_j{}' = K^{-1} \cdot \tilde{p}_j \tag{4-2}$$

$$\tilde{m}_j = \begin{bmatrix} \tilde{m}_{jx}{}' / \tilde{m}_{jz}{}' \\ \tilde{m}_{jy}{}' / \tilde{m}_{jz}{}' \\ \tilde{m}_{jz}{}' / \tilde{m}_{jz}{}' \end{bmatrix} \tag{4-3}$$

$$P_j = \tilde{m}_j \cdot Z_j \tag{4-4}$$

Particularizando para la cámara gran angular, tras aplicar estas ecuaciones, se obtienen las coordenadas tridimensionales referidas a los ejes $\{C_{GA}\}$, es decir, ${}^{C\_GA}P_j$. Estas coordenadas deben transformarse al marco de referencia de la cámara que queremos que apunte a dicho objetivo, $\{C_i\}$, siendo "*i*" la cámara que queremos que haga esto:

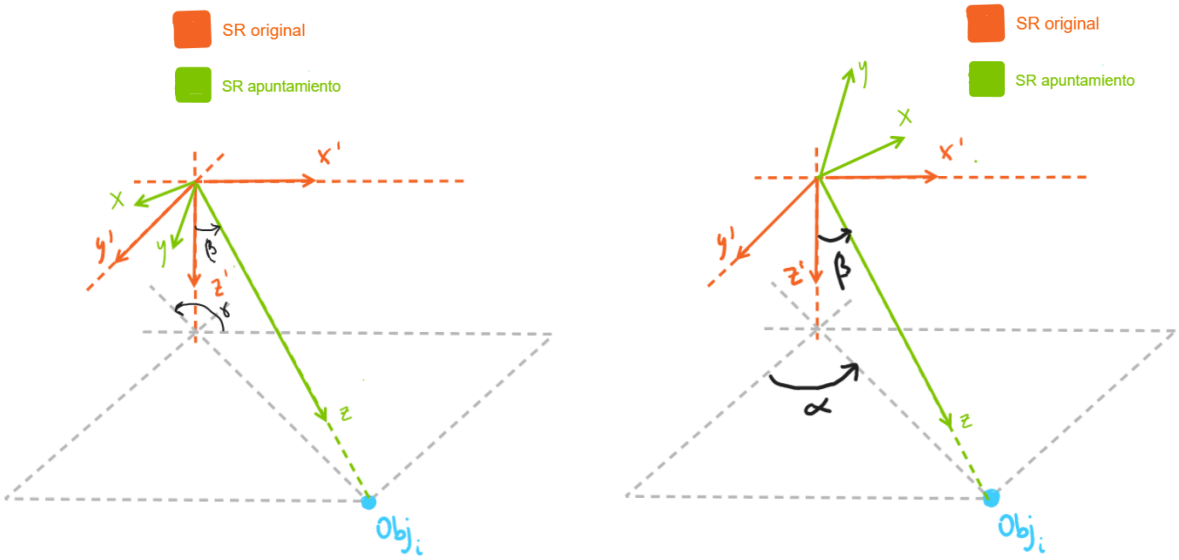
$$ {}^{Ci}P_j = {}^{Ci}P_{C\_GA} \cdot {}^{C\_GA}P_j \qquad (4\text{-}5) $$

Es importante notar que la matriz de transformación ${}^{Ci}P_{C\_GA}$ es conocida, constante y dependiente de la geometría del sistema (depende de la posición relativa entre la cámara C_i y la C_GA).

Problema de apuntamiento

Consiste en hacer que el Eje Z de la cámara (hacia donde apunta y captará la imagen) este contenido en la recta que une el origen del sistema de referencia de dicha cámara y el objeto (referido al SR de la cámara en cuestión). Sin embargo, esto deja un grado de libertad en cuanto a la solución, ya que una vez se ha conseguido incluir ese eje z en la recta, se podría girar en torno a él cualquier ángulo y la solución seguiría siendo válida.

Podemos tomar así dos soluciones como estándar, una en la que el Eje Y de la cámara apunte hacia el suelo (solución "cielo arriba") y otra en la que apunte hacia arriba (solución "cielo abajo").



Solución "Cielo Arriba" Solución "Cielo Abajo"

Figura 4-8. Posibles soluciones para el problema de apuntamiento.

La orientación se expresará mediante los ángulos de Euler, consistente en concatenar rotaciones sobre los ejes móviles rotando en el siguiente orden de manera sucesiva alrededor de eje Z, eje Y, y eje X. Esta representación de la orientación es típica en el ámbito de la robótica, así como en multitud de simuladores de robótica. Estos giros conocidos más comúnmente como *Yaw (en torno al eje Z), Pitch (en torno al eje Y) y Roll (en torno al eje X).*

De esta manera, para definir la orientación hay que obtener qué ángulos hay que girar para pasar de {Ci'} a {Ci} en las rotaciones respecto a "Z", "Y" y "X", que se **concatenarán respecto a los ejes móviles.**

Luego estos ángulos se calculan según la solución escogida como:

- ***Solución "cielo arriba":***

$$giro_z = \gamma = atan2(x_j, -y_j) \quad (4\text{-}6)$$

$$giro_y = 0 \quad (4\text{-}7)$$

$$giro_x = \beta = atan2\left(\sqrt{x_j^2 + y_j^2}, z_j\right) \quad (4\text{-}8)$$

- ***Solución "cielo abajo":***

$$giro_z = \alpha = atan2(-x_j, y_j) \quad (4\text{-}9)$$

$$giro_y = 0 \quad (4\text{-}10)$$

$$giro_x = \beta = atan2\left(-\sqrt{x_j^2 + y_j^2}, z_j\right) \quad (4\text{-}11)$$

Con respecto a estas ecuaciones hay que tener en cuenta lo siguiente:

- La tripleta conformada por (x_j, y_j, z_j) corresponden a las coordenadas del punto objetivo referidas al sistema de referencia de la cámara que lo apunta, {Ci'}.
- *atan2(b, a)* devuelve el ángulo formado por el eje x positivo y la recta que une el origen con el punto (*a, b*), es decir evita la compensación de 180º del ángulo que devolvería la arcotangente habitual en ciertos casos.

Podemos ver cómo un ángulo del primer cuadrante tendría el mismo resultado con ambas funciones

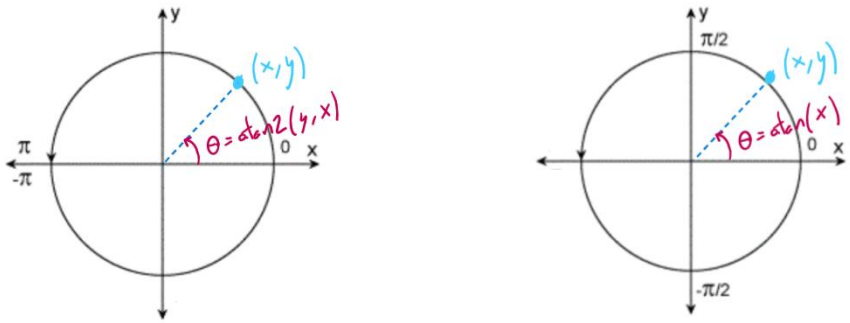


Figura 4-9. Rango y argumentos de las funciones *atan* y *atan2*.

Mientras que atan2 devuelve ángulos en el rango de [-π, π], atan los devuelve en el rango [-π/2, π/2], lo cual implica que, para ángulos pertenecientes al 2º, 3º o 4º cuadrante, habría que aplicar una corrección al ángulo obtenido mediante *atan* para obtener el ángulo adecuado.

Se puede ver la diferencia entre ambas para un caso en cada cuadrante y ver por qué es más conveniente y cómodo usar *atan2*:

- Primer cuadrante, punto en (x = 1, y = 1):
  - *atan(x)* = 0.7854 rad = 45º
  - *atan2(y, x)* = 0.7854 rad = 45º
- Segundo cuadrante, punto en (x = -1, y = 1):
  - *atan(x)* = -0.7854 rad = -45º (representaría ángulo en el 4º cuadrante)
  - *atan2(y, x)* = 2.3562 rad = 135º
- Tercer cuadrante, punto en (x = -1, y = -1):
  - *atan(x)* = -0.7854 rad = -45º (representaría ángulo en el 4º cuadrante)
  - *atan2(y, x)* = -2.3562 rad = -135º
- Cuarto cuadrante, punto en (x = 1, y = -1):
  - *atan(x)* = 0.7854 rad = 45º (representaría ángulo en el 1º cuadrante)
  - *atan2(y, x)* = -0.7854 rad = -45º

Vemos como para puntos en los cuatro cuadrantes, *atan2* devuelve correctamente el ángulo real al cubrir todo el rango de la circunferencia, mientras que *atan* requerería de correcciones que dificultarían obtener una representación cómoda del ángulo.

Dadas estas dos soluciones, se pueden plantear dos modos de funcionamiento del sistema:

- Manual: Se selecciona una de las dos soluciones descritas y se mantiene a lo largo del tiempo. Esto puede provocar cambios bruscos de referencia para los ejes del gimbal de 180º cuando se produzca un cambio de cuadrante.
- Continuista: En cada iteración se calculan ambas soluciones y se elige aquella que minimiza la diferencia angular en el giro respecto al Eje Z (*Yaw*) con el instante anterior, permitiendo que las referencias del gimbal no tomen saltos bruscos.

En este último modo, hay que tener en cuenta que en el caso de que se escoja (por el criterio descrito) la solución “cielo abajo”, habría que invertir la imagen por software para mostrar constantemente el mundo al derecho al usuario.

Además, este modo requiere de disponer el valor del giro del eje Z del instante anterior, por lo cual, para la inicialización, se deberá iniciar el algoritmo con una de las dos opciones para obtener un primer valor de este giro.

4.2.2 Ángulos a girar para recentrar la cámara tras primer apuntamiento

Las referencias de giro obtenidas en el apartado anterior son susceptibles a cierto error por una serie de razones como son: errores en la estimación de la altitud, errores de calibración de la cámara, resolución de la imagen, etc. Esto hace necesario un segundo proceso de apuntamiento de la cámara basada en lo que “ve” en su imagen.

Es imprescindible que el objetivo deseado ya aparezca en la imagen, luego para llevar a cabo este procedimiento, se requiere un primer apuntamiento con el método anterior que asegure la visualización del objetivo.

La idea aquí es calcular los ángulos **incrementales** que debemos girar la cámara para llevar unas coordenadas en la imagen *(u, v)* al centro de la imagen, logrando así un apuntamiento fino del objetivo deseado.

Tras un primer apuntamiento, debido a las fuentes de error mencionadas, es altamente probable obtener un desalineamiento del eje Z con la recta que une el origen del sistema de referencia de la cámara con la posición del objetivo visto desde dicho marco de referencia. Esto puede observarse mejor en la **Figura 4-10:**

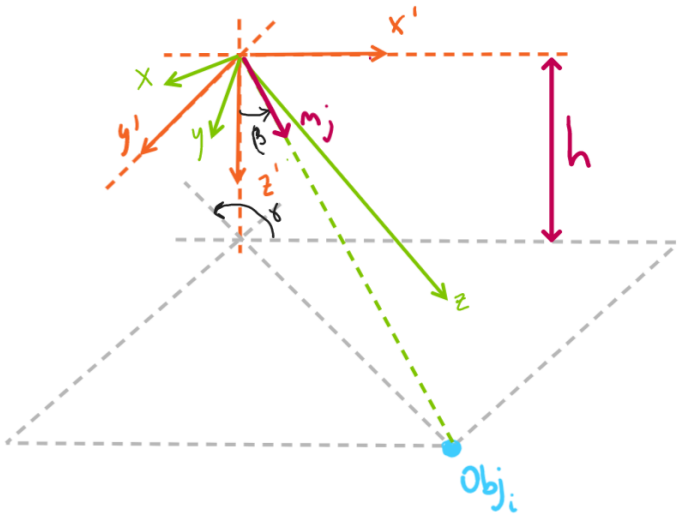


Figura 4-10. Desalineamiento de los ejes respecto al apuntamiento esperado.

Considerando el caso simplificado en el que el centro óptico coincide con el centro de giro, que es cómo se supone en la implementación, no se necesita una estimación de la altitud para este reajuste de los ángulos. Los nuevos ángulos incrementales a girar se pueden obtener a través de trigonometría básica.

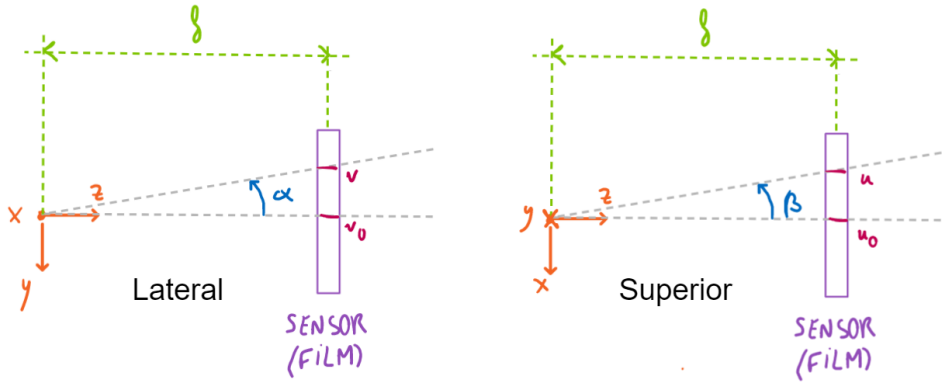


Figura 4-11. Vista lateral y superior del error en el apuntamiento.

Figura 4-12. Triángulos rectángulos extraídos del esquema de la **Figura 4-11**.

Observando la **Figura 4-12,** es prácticamente inmediato obtener las expresiones que proporcionan los nuevos ángulos a girar:

$$giro_z = 0 \tag{4-12}$$

$$giro_y = \beta = atan2((u - u_0), f_x) \tag{4-13}$$

$$giro_x = \alpha = atan2(-(v - v_0), f_y) \tag{4-14}$$

Por otra parte, esta resolución introduciría cierto error si el centro de giro no coincide con el centro óptico. Se puede apreciar esto en la **Figura 4-13,** donde se obtendría con el planteamiento anterior un ángulo de giro β, siendo el correcto β', siendo esto análogo para la vista lateral con su respectivo ángulo α.

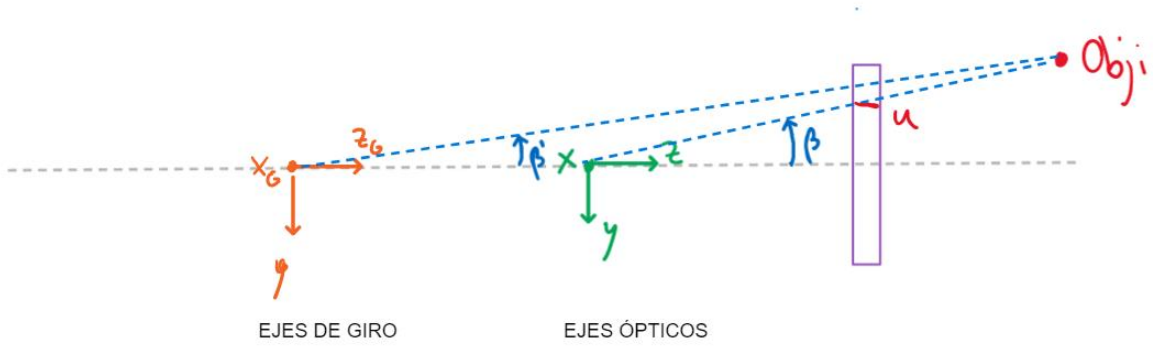


Figura 4-13. Error en el ángulo de giro si centro de giro y centro óptico no coinciden.

En este planteamiento, el objetivo sigue siendo, al igual que se veía en la **Figura 4-10**, lograr que el eje óptico de la cámara tenga la misma dirección que m_j. El primer paso es hallar la posición tridimensional del objetivo referido al marco de referencia de la cámara que queremos reorientar a partir de su propia imagen.

Se calcula el vector $\tilde{m}_j{}'$ a través de las ecuaciones (4-1) y (4-2), y se normaliza, para obtener el vector unitario o director m_j visto desde los ejes *{Ci}*.

$${}^{ci}m_j = \frac{\tilde{m}_j{}'}{\| \tilde{m}_j{}' \|} \tag{4-15}$$

Partiendo de que se conoce la orientación actual de la cámara, se conoce ${}^{ci'}R_{Ci}$, debiendo cumplirse la siguiente expresión:

$${}^{ci'}R_{Ci} \cdot {}^{ci}m_j \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \cos\beta \tag{4-16}$$

Con β el ángulo representado en la **Figura 4-10.** Por su parte a partir de las relaciones trigonométricas se obtiene:

$$\| {}^{ci}P_j \| \cdot \cos\beta = h \tag{4-17}$$

Sustituyendo, queda definida la posición tridimensional del punto como:

$$ {}^{Ci}P_j = \left\| {}^{Ci}P_j \right\| \cdot {}^{Ci}m_j = \frac{h}{\cos\beta} \cdot {}^{Ci}m_j = \frac{h}{{}^{Ci'}R_{Ci} \cdot {}^{Ci}m_j \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}} \cdot {}^{Ci}m_j \qquad (4\text{-}18) $$

Tras conocer la posición tridimensional del punto, se expresa respecto de los ejes *{Ci'}* y se opera como en el **apartado 4.2.1**, obteniéndose las dos soluciones mencionadas de nuevo.

Idealmente, este planteamiento puede suponer una solución más exacta y genérica del problema. Sin embargo, considerando la aproximación de que el centro óptico y el centro de giro no están muy distantes, se puede proceder directamente con la primera solución de este apartado, que además no depende de la estimación de la altura.

Finalmente, es destacable que los ángulos obtenidos en este apartado, tanto para la primera forma como para esta última, serían referencias para un controlador. Por tanto, el error que se introduzca mediante el primer método de este apartado podría ser corregido por la estrategia de control en bucle cerrado.

4.2.3 Asignación de objetivos

En los dos apartados anteriores se ha discutido la existencia de una serie de objetivos que serán avistados desde la cámara gran angular *{C_GA}* y como cierta cámara *{Ci}* deberá orientarse para centrar a uno de ellos (Obj_i).

Sin embargo, falta por determinar qué cámara apuntará a qué objetivo, es decir, concretar el problema de asignación de objetivos.

Una forma razonable de apuntar a los objetivos es tratando de maximizar la verticalidad en el apuntamiento de las cámaras orientables. Es decir, entre todos los objetivos, se escogerán aquellos que hagan que las cámaras comiencen con una verticalidad global máxima.

Esta primera asignación se realiza en base a lo que se observe desde la cámara gran angular *{C_GA}*. Para ello, se calcularán cada una de las orientaciones resultantes de cada una de las cámaras apuntando a cada uno de los objetivos. Es decir, para cada cámara y para cada objetivo se resolverán las ecuaciones (4-6) a (4-8) o de la (4-9) a (4-11), dependiendo de la configuración deseada.

Considerando la orientación representada mediante los ángulos de Euler ZY'X'', la verticalidad está intrínsecamente relacionada con el giro respecto al eje X.

Se puede explicar este concepto con la primera tripleta de ecuaciones, aunque el razonamiento es análogo para la segunda configuración.

Tras girar el ángulo dado por (4-6) o (4-9) respecto del eje Z, estaremos apuntando al suelo completamente, es decir, en un estado de máxima verticalidad. Como se ve en las ecuaciones, no se realiza ningún giro respecto del eje Y, por tanto, es el giro respecto del eje X el que aleja a nuestra cámara de la máxima verticalidad conseguida tras el giro respecto de Z. Esto se puede apreciar en la **Figura 4-14**, donde en lugar de apuntar a los objetivos verdes, la idea es asignar aquellos que produzcan un menor giro respecto al eje X (objetivos rojo y azul respectivamente)

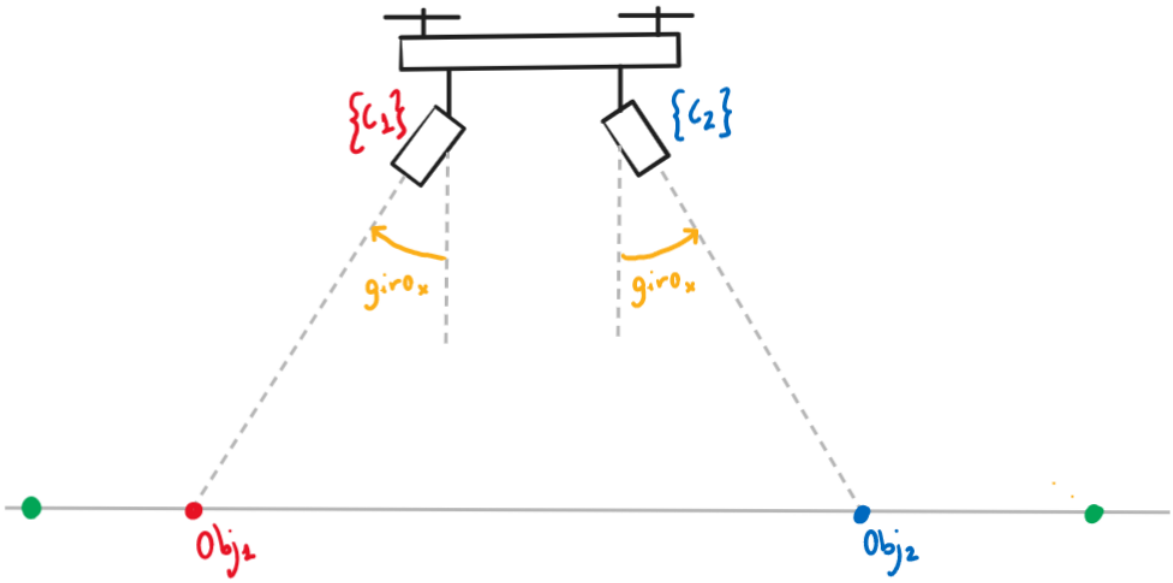


Figura 4-14. Asignación de objetivos para dos cámaras orientables según verticalidad.

Por tanto, la tarea de maximizar la verticalidad consiste en asignar los objetivos de tal modo que el valor absoluto del ángulo a girar respecto del eje X dado por (4-8) o (4-11) sea mínimo.

La idea es escoger la combinación de objetivos para la cual la suma de los ángulos de giro respecto al eje X de las orientaciones resultantes sea la menor posible, teniendo en cuenta que un objetivo no puede ser apuntado por más de una cámara.

Este **algoritmo de asignación** parte de asumir que tenemos "N" cámaras y necesariamente mayor número de objetivos que de cámaras orientables y sus pasos son los siguientes:

- Ordenar en una lista los objetivos de mayor a menor verticalidad en base a la métrica descrita para cada cámara orientable.
- Asignar a cada cámara el objetivo que tiene mayor verticalidad (el primero de su lista).
- Se debe comprobar si un mismo objetivo ha sido asignado a varias cámaras, lo cual no es deseable. Si no ocurre, se ha terminado, si en cambio sí ocurre, se deberá elegir otra combinación que no produzca conflictos.
- Sumar los valores de verticalidad para todas las posibles combinaciones de las "N" cámaras con sus primeros "N" objetivos de sus listas, habiendo eliminado previamente aquellas en las que se asigne a varias cámaras el mismo objetivo.
- Esto no resulta en un número tan descabellado de posibilidades ya que la mayoría de las combinaciones se eliminan por repetir el mismo objetivo. Además, este análisis sólo se ejecuta en la asignación inicial o bien en la reasignación cuando haya conflictos.
- Entre todas las posibles combinaciones restantes, se elige aquella que maximiza la verticalidad del conjunto, es decir, aquella para la que la suma del valor absoluto de los ángulos de giro respecto del eje X sea menor.

Tras resolver este problema, se tiene una asignación inicial de los objetivos. El enfoque ideal es mantener la asignación adjudicada inicialmente durante todo el tiempo, consiguiendo que cada cámara siga a su objetivo de forma precisa e independiente.

4.3 Análisis de grupos de objetivos (*Clusters*)

En el conjunto de apartados anteriores se ha discutido diferentes problemas para el caso de objetivos individuales, pero todo ello es adaptable a agrupaciones de ellos. En este apartado se discute precisamente el cómo pasar de tratar objetivos individuales a tratar conjuntos de ellos.

En cuanto a la implementación, se establecerán por tanto dos modos de funcionamiento:

- Modo de asignación y seguimiento de objetivos individuales.
- Modo de identificación, asignación y seguimiento de agrupaciones de objetivos.

Por otra parte, sin importar el modo que se escoja, el análisis de los objetivos como un grupo global será necesario en todo momento, ya que uno de los apartados de control tratará de llevar el centro del conjunto de todos los objetivos avistados en la cámara gran angular al centro de su imagen. Esto implica tratar de mantener a todos los objetivos en la imagen, logrando así mayor continuidad en su seguimiento.

4.3.1 Centro representativo del grupo

Una buena manera de definir el centro del grupo de objetivos es mediante el centroide del polígono conformado por el *convex-hull* [28] de dichos objetivos.

Este *convex-hull* o envolvente convexa, particularizado para el caso plano (correspondiente al caso de conjunto de centroides de objetivos en una imagen), consiste en obtener el polígono convexo cuyos vértices pertenezcan a los objetivos y que el área del mismo englobe a todos los demás objetivos.

Además, el polígono sólo se considera convexo si ninguno de sus ángulos interiores toma valores superiores a 180º.

Podemos ver en la **Figura 4-16** un ejemplo similar a lo que se realiza para los objetivos (15 en este caso) en una imagen:

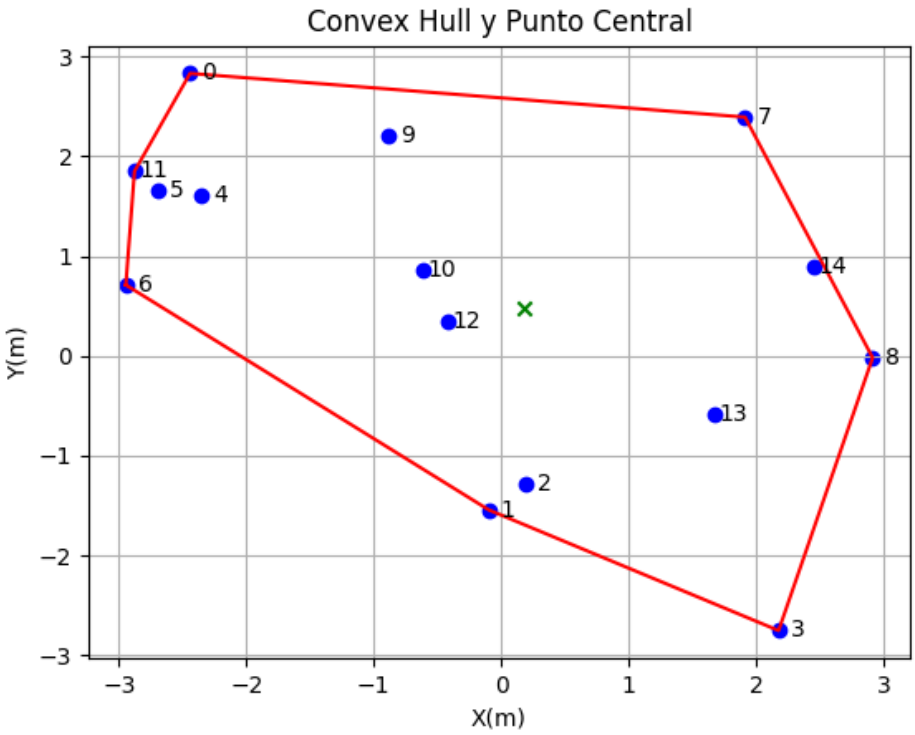


Figura 4-15. Distribución aleatoria de objetivos y cálculo de *convex-hull* y centroide.

El algoritmo utilizado para obtener el *convex-hull* está basado en el conocido como algoritmo *Jarvis March* o *gift wrapping algorithm* [28].

Conceptualmente, el algoritmo se basa en imaginar que se tiene papel de envoltura alrededor del conjunto de puntos y que gradualmente se va "tensando" dicho papel hasta que encierra los puntos adecuadamente, conformando así el *convex-hull.*

Importante mencionar que la complejidad temporal de este algoritmo es O(nh) donde "n" es el número de puntos de entrada al algoritmo (todos los objetivos que haya) y "h" es el número de puntos que conforman el *convex-hull*. En el peor de los casos, el número de puntos del *convex-hull* puede ser igual al número de puntos de entrada, resultando en una complejidad temporal de O(n^2).

Aunque pueda parecer computacionalmente costoso, es difícil pensar en alguna aplicación en la que el número de objetivos identificados sea superior a órdenes de 100, 200 o más objetivos. Por tanto, para este rango de número de objetivos, el algoritmo es suficientemente rápido, pudiendo verse su tiempo de cómputo para diferentes números de puntos de entrada en la **Figura 4-16.** Mencionar que, para cada punto, se ha hecho la media del tiempo de cómputo del proceso con 15000 iteraciones**:**

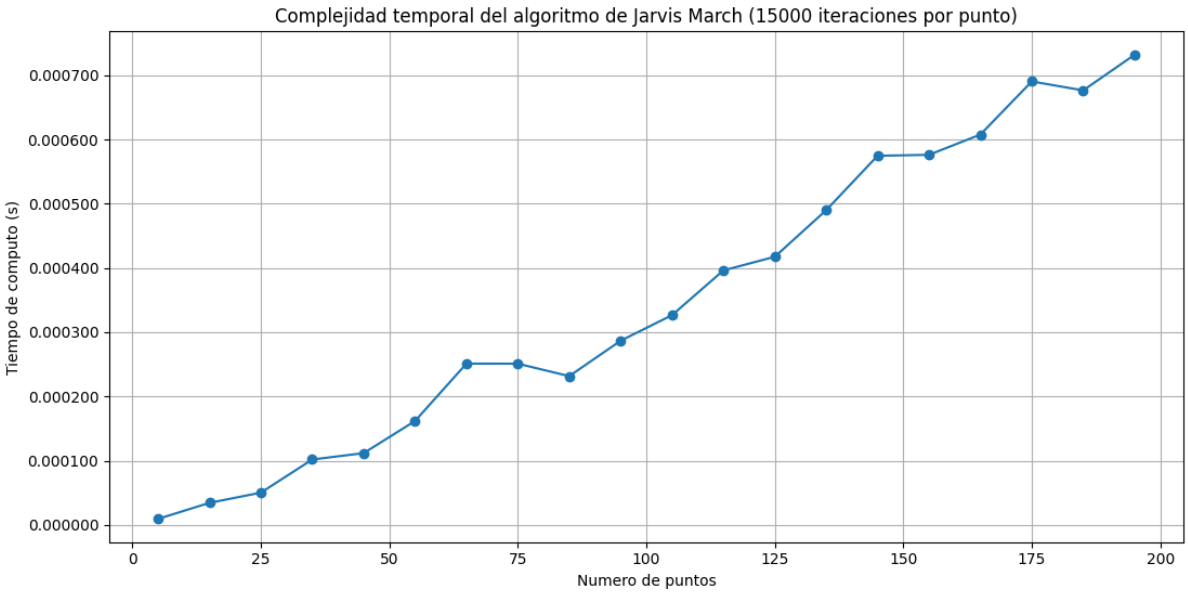


Figura 4-16. Tiempo de cómputo de *convex-hull* según número de puntos usando Jarvis March.

Aunque pueda parecer un algoritmo que podría tardar algo de tiempo, se ve como para el peor caso (casi 200 objetivos, cosa poco probable en una aplicación real), tardaría menos de 1 ms en realizar dicho cálculo.

Los pasos que toma este algoritmo son:

- Si el número total de objetivos es uno, dos o tres, la solución es inmediata, siendo los puntos que forman el *convex-hull* el conjunto de estos objetivos
- Si se tienen cuatro o más objetivos (o puntos en el plano), la solución se obtiene mediante el siguiente procedimiento:
    1. Encontrar el punto más a la izquierda del conjunto de puntos de entrada. Básicamente se compara las coordenadas en el eje X de todos los puntos y se escoge aquel cuyo valor sea mínimo. Este punto será el punto inicial del *convex-hull*.
    2. Inicializar una lista vacía que albergará los puntos del *convex-hull* y añadir el punto inicial a la misma.
    3. Seleccionar el siguiente punto en la dirección contraria a las agujas del reloj a partir del punto actual. Para hacer esto, se itera sobre el resto de los puntos y se encuentra el punto cuya dirección contraria a las agujas del reloj sea la menor respecto del punto actual.

- Para cada punto restante y el actual, calcular ese valor de dirección usando el concepto de orientación entre 3 puntos. Donde, en cada paso, se irán tomando tríos de puntos para hallar esa orientación, consistiendo en:
  - **Punto actual:** Se refiere al punto considerado parte del *convex-hull*. Inicialmente es el punto más a la izquierda y conforme se vayan añadiendo puntos a la lista, será el último punto añadido.
  - **Punto candidato:** Hace referencia al punto bajo estudio para determinar si puede ser el siguiente punto del *convex-hull* (notar que debe ser uno de los restantes no añadidos a la lista). El algoritmo itera sobre todos los restantes para encontrar el mejor candidato.
  - **Punto referencia:** Punto que se usa para determinar la dirección entre el punto actual y el candidato. Ayuda a la hora de realizar la comparación de la orientación de múltiples puntos relativo al punto actual. Se fija como cualquier punto diferente del punto actual y se va actualizando conforme el algoritmo progresa.
- Nombrando los puntos anteriores de la siguiente forma: **Punto actual = og, punto candidato = p1** y **punto referencia = p2**, se define el cálculo de la orientación del punto candidato como:

$$orientacion = ((p2_x - og_x) \cdot (p1_y - og_y)) - ((p1_x - og_x) \cdot (p2_y - og_y)) \quad (4\text{-}19)$$

- Una vez calculada esta orientación, se pueden tener varios casos, siendo el valor numérico fundamental para encontrar el candidato más adecuado:
  - *Clockwise* (orientación < 0)
  - *Counterclockwise* (orientación > 0)
  - *Collinear* (orientación = 0)

  Como se mencionaba, la idea es buscar de entre todos los puntos restantes aquel cuyo valor de orientación sea positivo (*counterclockwise*) y mínimo valor entre todos los candidatos.

4. Añadir el mejor candidato a la lista de puntos del *convex-hull*.
5. Repetir los pasos 3 y 4 hasta que nos encontremos con el punto inicial de nuevo, en cuyo caso el *convex-hull* ya conforma un polígono cerrado y se ha terminado el cálculo.
6. Devolver la lista de puntos que conforman los vértices de este polígono.

Finalmente, para terminar de aclararlo, se adjunta el siguiente enlace a un vídeo explicativo:

https://www.youtube.com/watch?v=nBvCZi34F_o

Así como el siguiente diagrama que muestra el algoritmo durante el proceso de cómputo, donde se aprecia como se van escogiendo aquellos candidatos con mejor valor de orientación *counterclockwise:*

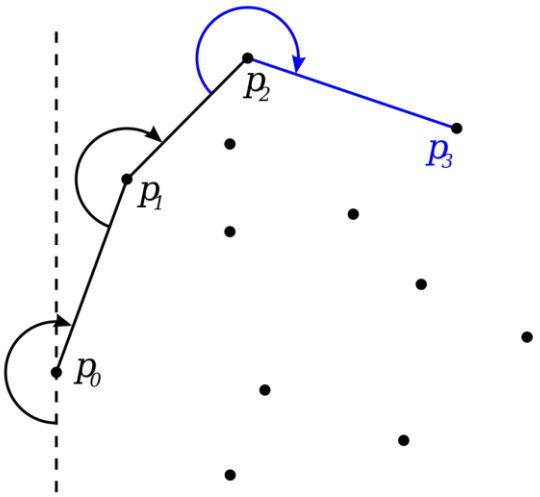


Figura 4-17. Proceso de obtención de puntos del *convex-hull* mediante Jarvis March [28].

Es de esta manera por tanto como se obtiene el polígono que se considera representativo del grupo, sin embargo, se necesita un punto al cuál debemos dirigir el UAV, que corresponderá al centroide o centro geométrico [29] de este polígono convexo.

Aunque la idea de usar el "centro de masas" del polígono utilizada en [10] es adecuada, se ha optado por variar el planteamiento de este cálculo para evitar ciertos problemas que se originan cuando en ciertas iteraciones el polígono se vuelve no convexo o degenerado (área del polígono = 0).

Por tanto, el algoritmo resultante será robusto ante cualquier tipo de polígono a la hora de obtener su centroide representativo. Consiste en lo siguiente:

- Como entrada, el algoritmo debe recibir tanto los vértices (coordenadas) que conforman el *convex-hull* como el número de ellos.
- Si el *convex-hull* sólo está compuesto por 1 vértice, el centroide será dicho vértice.
- Si el *convex-hull* está compuesto por 2 vértices, el centroide será constituido por la media aritmética de sus coordenadas X e Y.
- Si el *convex-hull* está compuesto por 3 ó más vértices, entra en juego el algoritmo en cuestión el cual tiene los siguientes pasos:
    - Se inicializa una lista de triángulos vacía.
    - Se generan triángulos seleccionando un vértice fijo e iterando sobre el resto de vértices del *convex-hull.* Para 'n' vértices, se generarán 'n-2' triángulos.
    - Para cada uno de los triángulos construidos, considerando los vértices como **v1 = (x1, y1), v2 = (x2, y2)** y **v3 = (x3, y3)**, se calcula su área mediante la fórmula de *Shoelace* [30] adaptada para el caso de un triángulo:

$$Área = \frac{|x1 \cdot (y2 - y3) + x2 \cdot (y3 - y1) + x3 \cdot (y1 - y2)|}{2} \quad (4\text{-}20)$$

    - Si la suma de las áreas de todos los triángulos no es nula, se calcula el centroide de cada triángulo dados sus vértices mediante la media aritmética de las coordenadas de sus vértices:

$$C_x = \frac{1}{3} \cdot (x_1 + x_2 + x_3) \; ; \quad C_x = \frac{1}{3} \cdot (y_1 + y_2 + y_3) \;\; \rightarrow \;\; C = (C_x, C_y) \qquad (4\text{-}21)$$

- Finalmente se obtiene el centroide del *convex-hull* como la **media ponderada (cuyos pesos son las áreas de los triángulos) de los centroides de los triángulos.** Es decir, cuanto mayor sea el área del triángulo, mayor será la influencia de su centroide en el cálculo de esa media ponderada.

- Si, por el contrario, el *convex-hull* resultase degenerado (con área nula) el cálculo del centroide del mismo se obtiene como la media aritmética de las coordenadas de todos los vértices que lo conforman. Destacar que este caso es muy particular y raramente se dará el caso en que el cálculo requiera de este método, pero se deja como solución simple a un caso excepcional.

Para mejor comprensión de este algoritmo, se presenta un diagrama en el que se aprecia la obtención de los diferentes triángulos y sus centroides a la hora de obtener el centroide del *convex-hull* con el algoritmo previamente mencionado en la **Figura 4-17**:

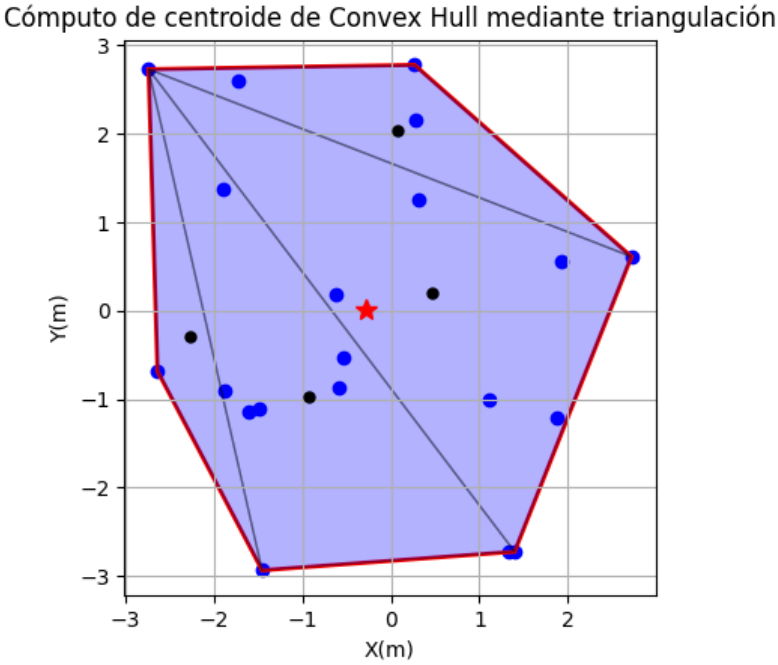


Figura 4-18. Cómputo de centroide de *Convex-hull* mediante triangulación.

Analizando esta figura, se puede observar cómo para el conjunto de puntos azules se ha calculado su *convex-hull* con el procedimiento anteriormente explicado. Una vez se tienen los vértices que conforman este polígono, se obtiene el centroide del mismo (marcado por la estrella roja) a partir de los centroides (marcados por cruces negras) y áreas de los triángulos como se explicaba.

También puede ser interesante como para el algoritmo anterior, hacer un breve estudio de su complejidad temporal. Al ser un algoritmo extraído de diferentes conceptos geométricos, su complejidad temporal no es inmediata de conocer, pero haciendo un breve estudio de los pasos que conlleva se puede obtener:

- **Generación de triángulos:** Se generan triángulos seleccionando un vértice fijo e iterando sobre el resto de vértices del *convex-hull.* Para 'n' vértices, se generarán 'n-2' triángulos. Este paso tiene una complejidad de O(n).
- **Cálculo de áreas:** Implica operaciones aritméticas simples. Al haber 'n-2' triángulos este paso

también tiene una complejidad de O(n).

- **Cálculo de centroides:** También implica operaciones aritméticas simples. De nuevo, al haber 'n-2' triángulos este paso también tiene una complejidad de O(n).

En general considerando todos los pasos, el algoritmo para calcular el centroide del *convex-hull* tiene una complejidad de O(n), donde 'n' es el número de vértices en el polígono.

Al igual que se realizó en la **Figura 4-16,** se puede repetir el estudio de tiempo de cómputo de este algoritmo para calcular el *convex-hull* en función del número de puntos de entrada (lógicamente considerando sólo el tiempo de cómputo del cálculo del centroide del polígono). Mencionar que, de nuevo, para cada punto, se ha hecho la media del tiempo de cómputo del proceso con 15000 iteraciones, para lograr una medida fiable.

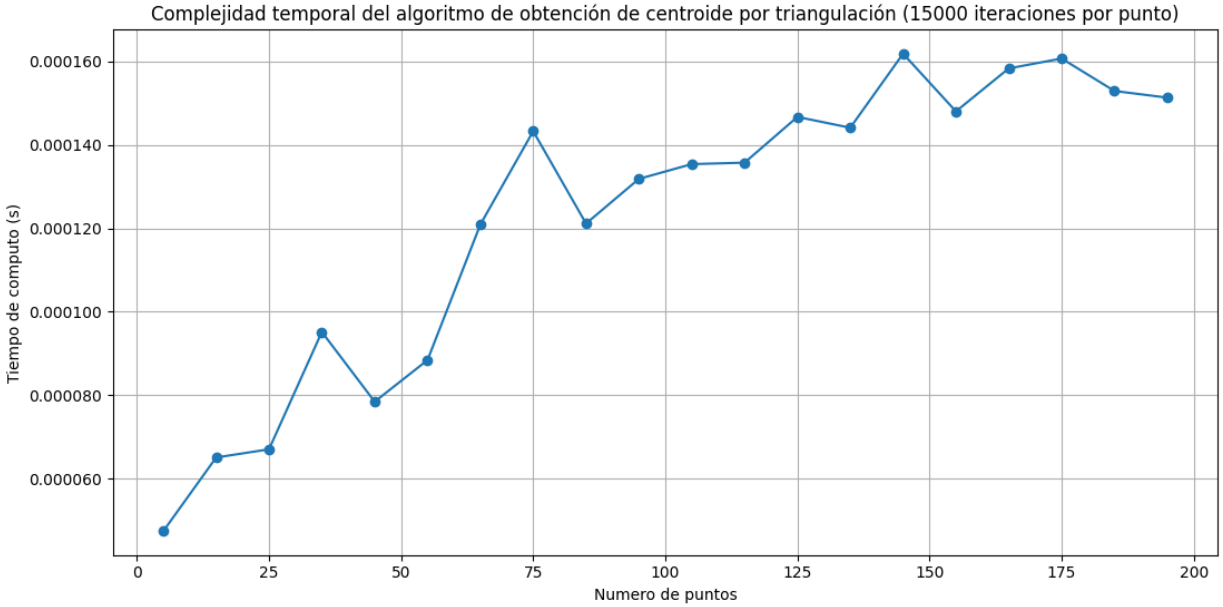


Figura 4-19. Tiempo de cómputo de centroide de *convex-hull* según número de puntos usando triangulación.

Este proceso es más rápido que el cálculo del propio *convex-hull* (alrededor de 0.1 ms para entorno a 200 puntos, que es un rango bastante superior al que podría ser un rango normal de objetivos identificados) y además bastante robusto frente a multitud de situaciones.

Por tanto, considerando, un número de objetivos razonable de 25, el tiempo que tardarían en ejecutarse ambos algoritmos sería del orden de 0.2 ms, siendo importante esta gran eficiencia temporal para contrarrestar otras tareas que sí son mucho más costosas temporalmente.

4.3.2 Técnica de agrupamiento de objetivos escogida

Llegados aquí, se conoce cómo representar un grupo de objetivos y su punto representativo, pero no el cómo escoger los objetivos que formen un grupo.

Como se dejó entrever en el apartado anterior, en primer lugar se considerará un cluster a un nivel superior formado por todos los objetivos avistados. Será el centroide del mismo sobre el que se quiere centrar el UAV para maximizar la capacidad de avistamiento de objetivos y mantenerlos visibles a lo largo del movimiento de los mismos.

Sin embargo, según el modo de funcionamiento, si se desea apuntar con una cámara orientable a un grupo de objetivos, se debe poder subdividir este grupo de nivel superior en subgrupos, y asignarlos a aquella cámara que mejor le convenga.

Si se tienen 'n' cámaras orientables, la idea es dividir el número total de objetivos avistados en 'n' grupos, de tal forma que cada cámara realice el seguimiento del grupo que le sea asignado. Se logra así obtener imágenes más cercanas y con mayor resolución de todos los objetivos repartiendo el trabajo entre las cámaras orientables.

Una vez planteado el problema, es momento de ver cómo realizar esta subdivisión. Por ser un algoritmo popular y efectivo para esta tarea, se opta por hacer uso de *K-Means Clustering* [31], el cual, permite dividir un conjunto de puntos (los objetivos en nuestro caso) en número elegible de agrupaciones de puntos con identidad propia.

En este caso, se escoge una solución parametrizable ofrecida por la librería *OpenCV*. Aún así resulta interesante explicar el funcionamiento del algoritmo. K-means consiste en un algoritmo iterativo y busca dividir 'n' puntos en 'k' grupos. Para ello lleva acabo los siguientes pasos:

- **Inicialización:** Se escogen 'k' puntos del conjunto de manera aleatoria como centroides. Estos centroides representan el punto central inicial de cada cluster.
- **Asignación:** Asignar cada punto del conjunto al centroide (del paso anterior) más cercano basado en una métrica de distancia, típicamente la distancia Euclidea. Es decir, cada punto pertenecerá al cluster a cuyo centroide esté más cerca.
- **Actualización:** Recalcular los centroides de cada cluster tomando la media aritmética de todos los puntos asignados a ese cluster. De esta manera, se calcula en este paso los nuevos puntos centrales de cada cluster.
- **Repetición:** Se iteran los pasos de asignación y actualización hasta que haya convergencia o bien se cumpla el criterio de parada. El algoritmo converge cuando los centroides no cambian significativamente o cuando se alcanza un número máximo de iteraciones.
- **Terminación:** El algoritmo termina y se obtienen los clusters asignados y sus centroides.

Cabe destacar que típicamente se suele ejecutar el proceso anterior un cierto número de veces utilizando puntos iniciales aleatorios diferentes. Con esto, podemos elegir el mejor resultado de *clustering* basado en cierto criterio, como puede ser minimizar la varianza total dentro de cada cluster (también conocida como la suma de las distancias al cuadrado entre los puntos y sus respectivos centroides).

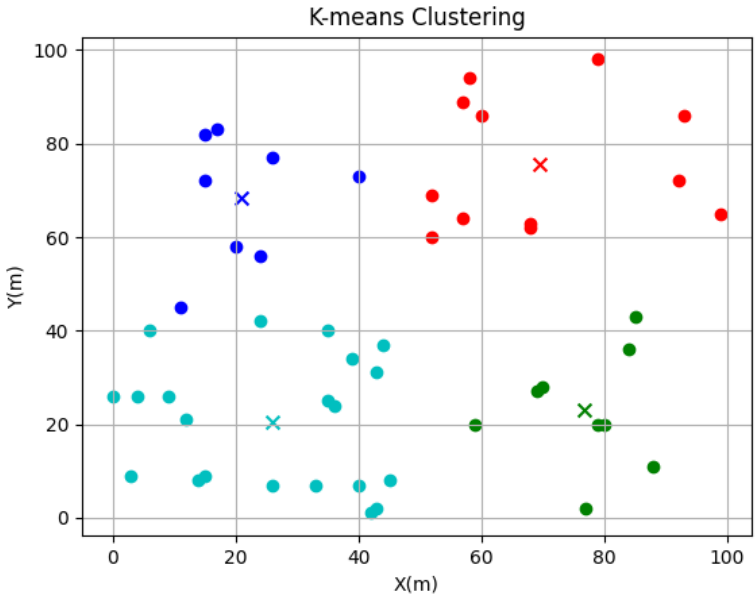


Figura 4-20. Resultado de *K-means Clustering* para 50 puntos y 4 clusters (inicialización aleatoria).

Vemos en la **Figura 4-20** como para 50 puntos distribuidos aleatoriamente se clasifica a través del algoritmo de K-means de *OpenCV* en 4 agrupaciones y con los siguientes parámetros:

- *Número de clusters*: 4.
- *Forma de escoger los centroides iniciales*: aleatoria.
- *Número máximo de iteraciones para que converja cada solución*: 100.
- *Número de intentos con instantes iniciales diferentes*: 10.

Puede ser interesante ver para la misma distribución de puntos, cómo resultaría la clasificación si se modifica la forma de escoger los centroides iniciales al flag 'cv2.KMEANS_PP_CENTERS', lo cual implica que se estaría usando la conocida como inicialización "K-means++" [32]. Esta idealmente implica una mejora sobre la inicialización aleatoria, ya que selecciona los centroides iniciales de tal manera que fomenta una mejor convergencia y puede resultar en clusters más precisos y estables.

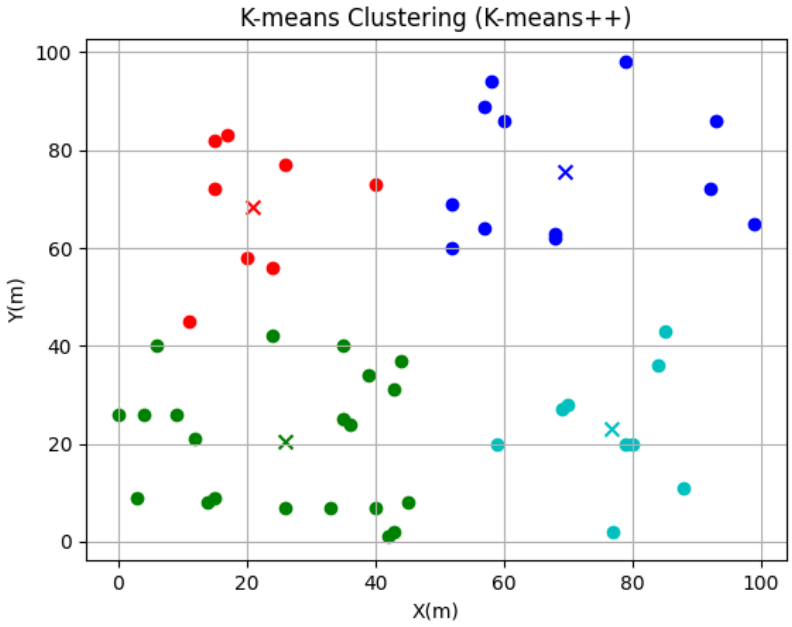


Figura 4-21. Resultado de *K-means Clustering* para 50 puntos y 4 clusters (inicialización k-means++).

Vemos ahora en la **Figura 4-21** como para la misma distribución el resultado es idéntico, esto se explica porque los puntos generados son fácilmente separables y no presentan una situación compleja. Los parámetros del algoritmo en este caso han sido:

- *Número de clusters*: 4.
- *Forma de escoger los centroides iniciales*: k-means ++.
- *Número máximo de iteraciones para que converja cada solución*: 100.
- *Número de intentos con instantes iniciales diferentes*: 10.

Visto el resultado de ambas sobre una distribución no muy compleja, parece interesante hacer un breve estudio sobre los resultados de ambas frente a una distribución más exigente en cuanto a su separabilidad.

Para dificultar la separabilidad, se va a generar una nube de 1000 puntos bastante condensada en torno a un centro que se quiere separar en 7 agrupaciones. Con esta distribución se puede evaluar si hay una diferencia real en cuanto a ambas formas de inicializar el algoritmo. Notar que se ha modificado el número máximo de iteraciones a 10 en este caso.
Vemos en la **Figura 4-22** ambos resultados mano a mano para facilitar la comparativa.

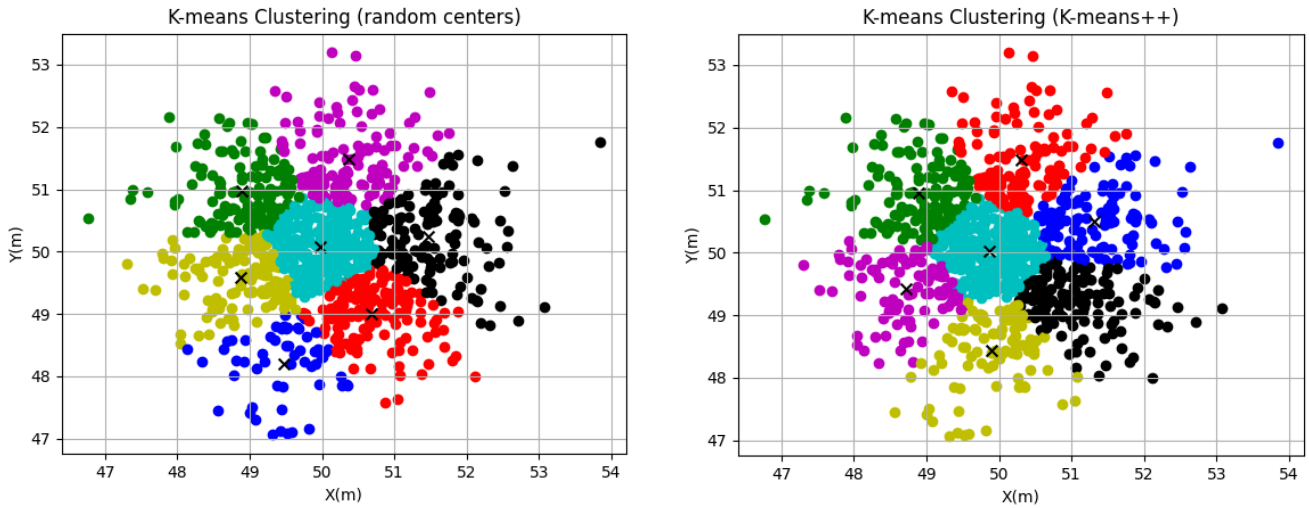


Figura 4-22 Comparativa de *Kmeans Clustering* con sus dos métodos de inicialización.

Los parámetros del algoritmo para este caso han sido:

- *Número de clusters*: 7.
- *Forma de escoger los centroides iniciales*: random centers (izquierda); k-means ++ (derecha).
- *Número máximo de iteraciones para que converja cada solución*: 10.
- *Número de intentos con instantes iniciales diferentes*: 10.

Se observa en este caso una ligera diferencia en los clusters obtenidos, aunque es difícil evaluar que separación ha sido más precisa, se observa una compactación ligeramente mayor en los clusters de *Kmeans++*.
En cualquier caso, para nuestra aplicación cualquiera de los dos métodos podría ser utilizado ya que típicamente los objetivos de la imagen se corresponderán con distribuciones similares a las de la **Figura 4-20 y Figura 4-21**, donde como se veía el resultado era similar con ambos.

Tras este pequeño análisis, es importante tener en cuenta que a cada grupo resultante de este proceso se le aplicará el procesado de los apartados anteriores, es decir, se calculará su *convex-hull* y el centroide del mismo.
Los centroides mostrados en la **Figura 4-20, Figura 4-21 y Figura 4-22** no coinciden con el punto al que queremos apuntar la cámara.
Es decir, los centroides que llevan a cabo esta clasificación son resultado de la media aritmética de los puntos pertenecientes a su grupo, el cuál típicamente no coincidirá con el centro del *convex-hull* que sí es el punto representativo del grupo.

El motivo de esto es que al tomar la media aritmética para hallar el centro, este puede verse desplazado hacia donde haya una mayor densidad de puntos, lo cual para el caso del apuntamiento mediante una cámara no es deseable.

Por su parte, como ya vimos en su apartado, el centroide del *convex-hull* no sufre de este efecto y por ello es un mejor punto representativo del grupo.

Al igual que para los algoritmos previos, también puede resultar interesante hacer un breve estudio de la complejidad temporal de *K-means Clustering*, aunque en este caso, no es tan importante que sea rápido en cuanto a cómputo ya que sólo se aplica durante la asignación inicial, o bien en caso de reinicio del sistema en caso de pérdida o error.

En este caso el análisis para hallar la complejidad temporal de este algoritmo se puede ver analizando sus pasos descritos previamente:

- **Inicialización:** En la versión de inicialización aleatoria de centros la complejidad temporal es O(K) con K el número de clusters, mientras que en su versión de inicialización con k-means++ es O(KN) con N el número de puntos totales.
- **Asignación:** En este paso cada punto se asignaba al centroide más cercano, luego la complejidad de este paso es O(KN).
- **Actualización:** En este paso se actualizan los centroides según los puntos asignados al cluster, luego la complejidad de este paso es O(KNd), donde N es el número de puntos, K es el número de clusters y d es el número de dimensiones de los puntos (2 si es un plano, 3 si es espacial, etc.).
- **Convergencia:** En este paso se repiten los dos pasos anteriores hasta alcanzar un número de iteraciones o cierto criterio. En cualquier caso, tomará un número 'i' de iteraciones.

Por tanto, en base al estudio de los pasos, típicamente la complejidad temporal del algoritmo de *K-means* se expresa típicamente como O(NKdi), con N el número de puntos, K es el número de clusters, d es el número de dimensiones de los datos e i es el número de iteraciones llevadas a cabo para converger.

Podemos ver en la **Figura 4-23**, para ciertos números de puntos y para diferentes clusters en los que separar el tiempo de cómputo de este algoritmo:

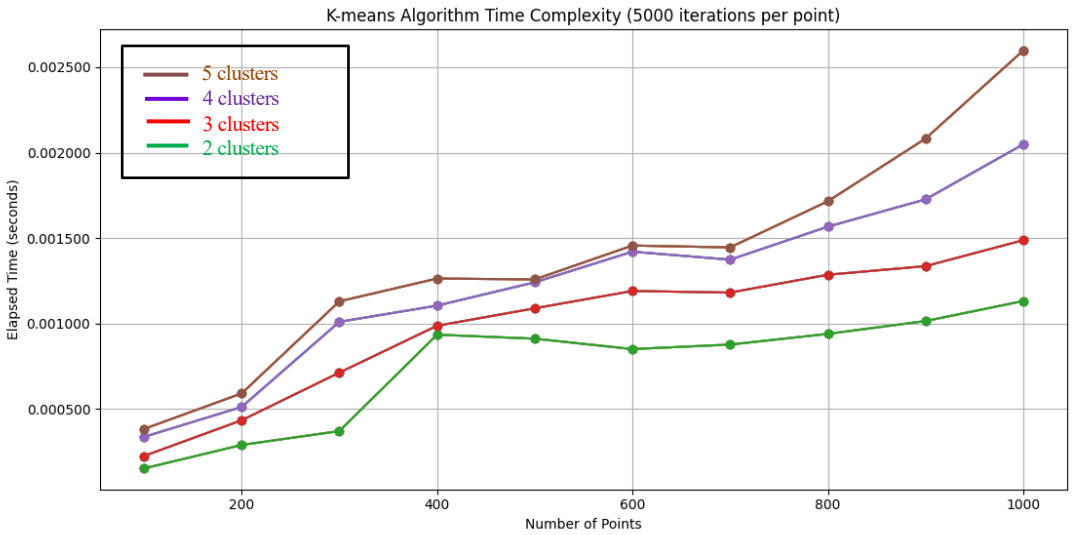


Figura 4-23. Tiempo de cómputo de clusters usando algoritmo *K-means* según número de puntos.

Vemos como vuelve a ser un algoritmo bastante rápido, del orden de < 1 ms para el rango de objetivos típico de nuestra aplicación.

4.3.3 Zoom óptico en modo de agrupaciones

Suponemos que se tiene un grupo identificado mediante los apartados anteriores de manera que le ha sido asignado a cierta cámara. Partiendo de que la cámara apuntará correctamente al centroide de su *convex-hull*, falta incluir una última funcionalidad: se quiere ajustar el zoom óptico de la cámara orientable de manera que el grupo de objetivos esté encuadrado.

Este zoom óptico de la cámara es directamente dependiente de la distancia focal de la lente, por lo que la cuestión a resolver consiste en obtener la distancia focal *f* para ajustar la dispersión del grupo en la imagen.

La funcionalidad del ajuste del zoom óptico se reserva exclusivamente a las cámaras orientables cuando están trabajando en modo de seguimiento de agrupaciones. Hay que destacar que estas cámaras orientables, antes de tener asignados sus correspondientes grupos de objetivos, estarán configuradas con la distancia focal al mínimo para facilitar la identificación del cluster asignado en su propia imagen, tras la reorientación de cada cámara.

Una vez que la cámara esté apuntando al centroide del grupo y el grupo haya sido identificado en su propia imagen, lo primero es calcular el radio de la circunferencia **en píxeles** centrada en el centroide del *convex-hull* que inscriba a todos los puntos de la agrupación. Por tanto, este radio irá desde dicho centro hasta el vértice más alejado del mismo. Esto se ve mejor en el esquema de la **Figura 4-24**:

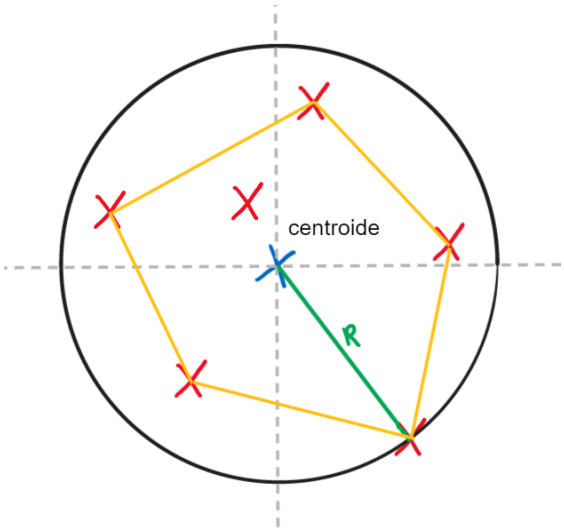


Figura 4-24. Obtención del radio de la circunferencia escogida que inscribe los objetivos.

Para llevar a cabo este cálculo, se basa el procedimiento en una variante del algoritmo de Welzi [33] que consiste en un algoritmo recursivo descrito por los siguientes pasos:

1. Casos base**:**
    a. Si no hay puntos, se devuelve un círculo con radio 0 y un centro arbitrario.
    b. Si solo hay un punto, devolver un círculo con ese punto como el centro y radio 0.
2. Seleccionar aleatoriamente un punto ‘p’ del conjunto de puntos.
3. Recursivamente encontrar el mínimo círculo que inscribe el resto de los puntos.
    a. Ignorar el punto seleccionado ‘p’ y recursivamente encontrar el mínimo círculo que inscribe el resto de los puntos.
    b. Este paso reduce el problema a un subconjunto menor de puntos.
4. Comprobar si el punto seleccionado ‘p’ está dentro del mínimo círculo que inscribe del paso 3:
    a. Si ‘p’ está dentro del círculo, devolver ese círculo como el mínimo círculo que inscribe para todo el conjunto de puntos completo.

5. Expandir el círculo para incluir ‘p’ en su frontera:
    a. Crear un nuevo círculo que incluye ‘p’ en su frontera.
    b. Recursivamente encontrar el mínimo círculo que inscribe el resto de los puntos considerando a ‘p’ como uno de sus puntos frontera.
    c. Este paso intenta encontrar el menor círculo que incluya ‘p’ en su frontera.
6. Comprobar si el nuevo círculo encierra todos los puntos:
    a. Si el nuevo círculo encierra todos los puntos, devolverlo como el mínimo círculo para todo el set de puntos.
7. Expandir el círculo aún más si fuese necesario:
    a. Si el nuevo círculo no encierra todos los puntos, expandirlo para que incluya al punto más alejado de su frontera.
    b. Este paso garantiza lograr que todos los puntos quedan dentro del círculo.
8. Devolver el mínimo círculo finalmente obtenido.

A través de ir seleccionando recursivamente puntos aleatorios, expandir el círculo y comprobando si encierra todos los puntos, el algoritmo de Welzi gradualmente encuentra el menor círculo que encierra todos los puntos dados. Continúa hasta que se obtiene el mínimo círculo que inscribe todo el *set* de puntos dado.

Podemos ver en la **Figura 4-25** para una supuesta distribución de clusters, se aplica este algoritmo para cada uno de ellos, obteniendo así el radio y centro de cada una de las circunferencias.

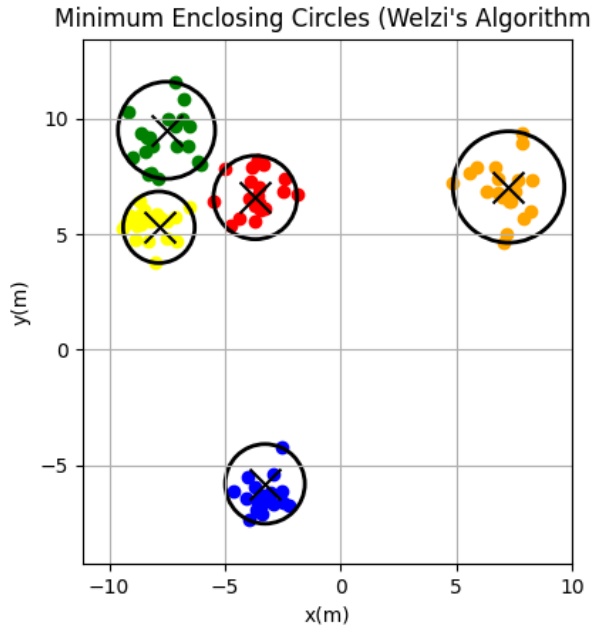


Figura 4-25. Obtención de circunferencia que inscribe los puntos de un cluster para varios de ellos.

Como se ha mencionado, la idea es encontrar el radio y centro de esta circunferencia para ajustar en consecuencia la distancia focal de la cámara que lo va a encuadrar.
La funcionalidad diseñada que se propone consiste en establecer una distancia focal por defecto con su valor mínimo y modificarla según las expresiones sugeridas a continuación para encontrar la distancia focal de referencia que encuadre mejor al grupo.

Se asume que el ancho de la imagen tiene una mayor resolución que el alto, aunque en caso contrario, el razonamiento sería totalmente análogo.

Lo que se busca es ajustar la longitud *R* (radio de la circunferencia) a una determinada longitud vertical en la imagen. Luego el problema a resolver consiste en dado un Δv en la imagen actual con una distancia focal *f*, encontrar la distancia focal *f'* para que Δv pase a valer $\Delta v'$. Se deja la **Figura 4-26** para clarificar este problema.

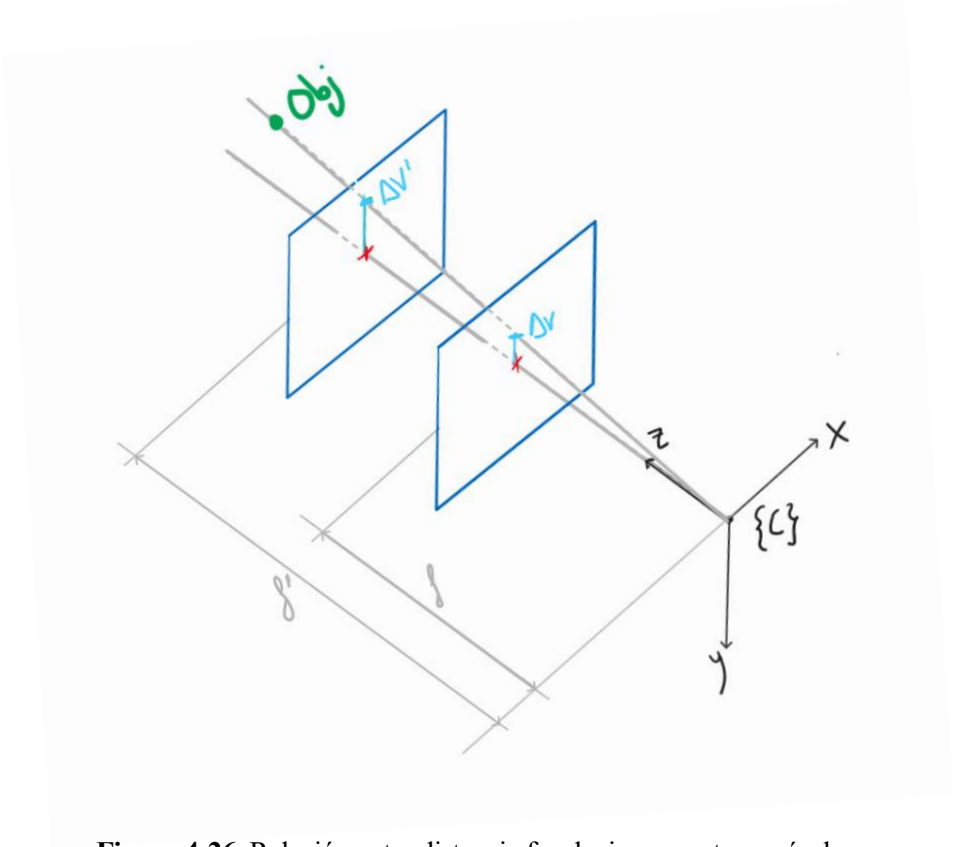


Figura 4-26. Relación entre distancia focal e incremento en píxeles.

Dadas las coordenadas tridimensionales del punto (X, Y, Z), las ecuaciones de proyección en el eje vertical quedarían:

$$y = f \cdot \frac{Y}{Z} \;\; ; \;\; v = \frac{y}{\rho_y} + v_0 \rightarrow \;\; v = \frac{f}{\rho_y} \cdot \frac{Y}{Z} + v_0 \qquad (4\text{-}22)$$

Por tanto, los incrementos mencionados pueden expresarse como:

$$\Delta v = \frac{f}{\rho_y} \cdot \frac{\Delta Y}{\Delta Z} \;\; ; \;\; \Delta v' = \frac{f'}{\rho_y} \cdot \frac{\Delta Y}{\Delta Z} \qquad (4\text{-}23)$$

Considerando que las coordenadas tridimensionales de los puntos no cambian y la altura del píxel es constante, se obtiene la siguiente relación:

$$\frac{\Delta v}{f} = \frac{\Delta v'}{f'} \rightarrow f' = f \cdot \frac{\Delta v'}{\Delta v} \qquad (4\text{-}24)$$

Es decir, para esta aplicación particular, queremos que la distancia *R* sea un tercio de la altura en píxeles menos un cierto margen, obteniendo la siguiente ecuación a implementar:

$$f' = f \cdot \frac{\frac{M}{3} - margen}{R} \qquad (4\text{-}25)$$

Donde *M* será el número de filas de la imagen (o también puede verse como la altura en píxeles del sensor).

De esta manera, se puede obtener la nueva f' a partir de conocer la distancia focal actual f y la distancia R comentada. Este valor será la referencia a fijar en el zoom óptico siempre y cuando no supere el valor máximo de distancia focal disponible en la cámara orientable, ya que en dicho caso se saturará a dicho valor la referencia obtenida.

Estos cálculos se efectuarán en todo momento cuando se esté apuntando a un grupo de objetivos, cambiando la distancia focal de la cámara según la estrategia propuesta, y la dispersión de los objetivos en tiempo real, consiguiendo encuadrar al grupo de objetivos en todo momento.

4.4 Zoom óptico en modo de objetivos individuales

Cuando el sistema esté trabajando en este modo, considerando que las cámaras orientables son inicializadas con la distancia focal mínima para garantizar en la medida de lo posible un buen apuntamiento en las fases iniciales, podemos encontrarnos que, una vez asignado un objetivo a la cámara y esta realiza el apuntamiento previo aproximado, el objetivo aparezca con poco tamaño, es decir, muy alejado en la imagen.
Esto no es deseable, puesto que el propósito de las cámaras orientables es apuntar a un objetivo individual con mayor detalle y resolución.
Por tanto, surge la necesidad de, al igual que para el modo clustering, tener un algoritmo que aproveche el zoom óptico de las cámaras y proporcione un seguimiento del objetivo con una mayor resolución, permitiendo una mejor visualización del objetivo.

Importante notar que la explicación de la dinámica aplicada al zoom óptico aparece en el siguiente capítulo aunque se mencione a continuación.

4.4.1 Algoritmo de obtención de referencia de zoom

En este caso se hace uso del área en píxeles del objetivo en cuestión donde buscaremos que mediante modificaciones en el zoom de la cámara tome un valor que permita la visualización adecuada de dicho objetivo. Tras una serie de pruebas en las simulaciones, se toma un valor de área de 600 $píxeles^2$ como valor necesario para obtener una imagen adecuada del objetivo.

Para calcular el área del objetivo en cuestión, se procede obteniendo la máscara binaria similar a las de la **Figura 4-6** e identificando al objetivo de interés como el más cercano al centro de la imagen, lo cual sucede gracias a ese apuntamiento previo aproximado.

Una vez teniendo la "etiqueta" binaria del objetivo de interés se obtiene su bounding box y se cálcula el área mencionada como:

$$área = ancho_{bbox} \cdot alto_{bbox} \quad (4\text{-}26)$$

Teniendo dicho alto y ancho píxeles como unidad.

Con esto, una vez termina el proceso de apuntamiento aproximado, se plantea el siguiente algoritmo simple para determinar la referencia de distancia focal:

- Para cada una de las cámaras orientables:
    - Iterar desde la distancia focal mínima (6 mm) hasta la distancia focal máxima (22 mm) con incrementos de 1 mm por iteración haciendo lo siguiente:
        - Establecer como referencia el valor de distancia focal de la iteración actual
        - Esperar 50 ms a que la dinámica del zoom actúe y modifique realmente el zoom.
        - Calcular el área del objetivo más céntrico de la manera previamente descrita.

- Si el área es igual o superior al área objetivo designada (600 $píxeles^2$), se sale del bucle que itera sobre valores de distancia focal.

Para visualizar el por qué de la conveniencia de este proceso también para este modo de funcionamiento, podemos contrastar la diferencia en el seguimiento resultante sin aplicar zoom y aplicándolo en las siguientes figuras obtenidas de las simulaciones.

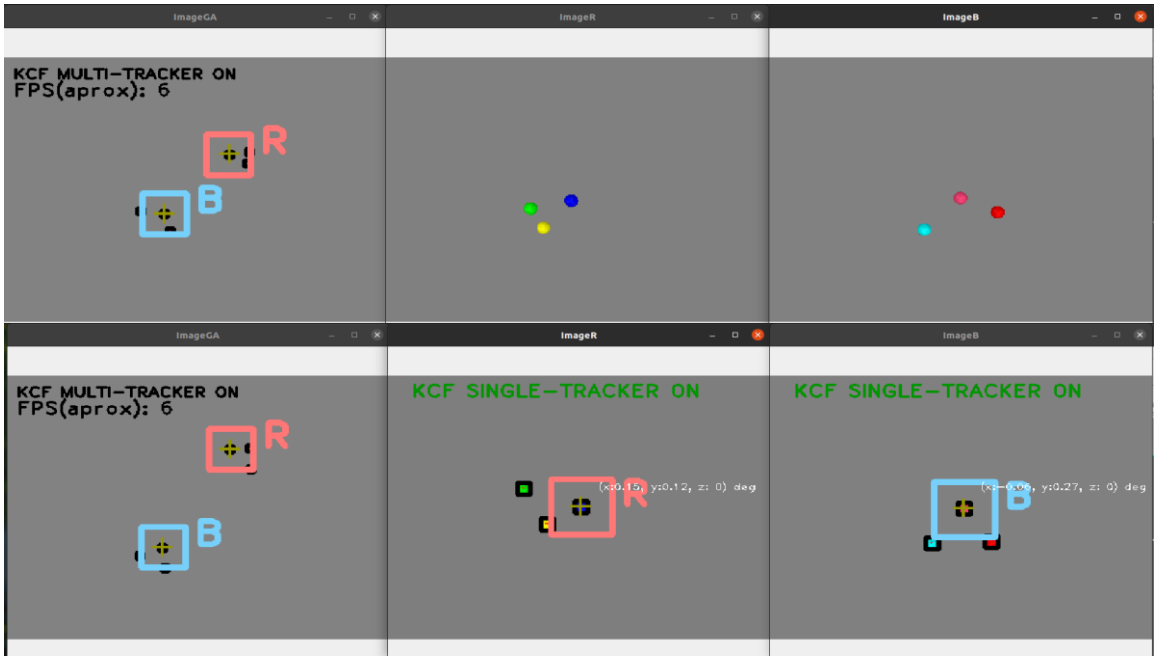


Figura 4-27. Comparativa entre fase de apuntamiento aproximado y fase de seguimiento sin incluir zoom.

Aunque se identifica y se realiza correctamente el seguimiento del objetivo, está claro que la imagen no está siendo aprovechada y el objetivo podría verse mejor si se incrementase el zoom. Para ello, vemos en la **Figura 4-28** una comparativa similar aplicando el algoritmo anteriormente descrito y modificando por tanto el zoom entre ambas fases.

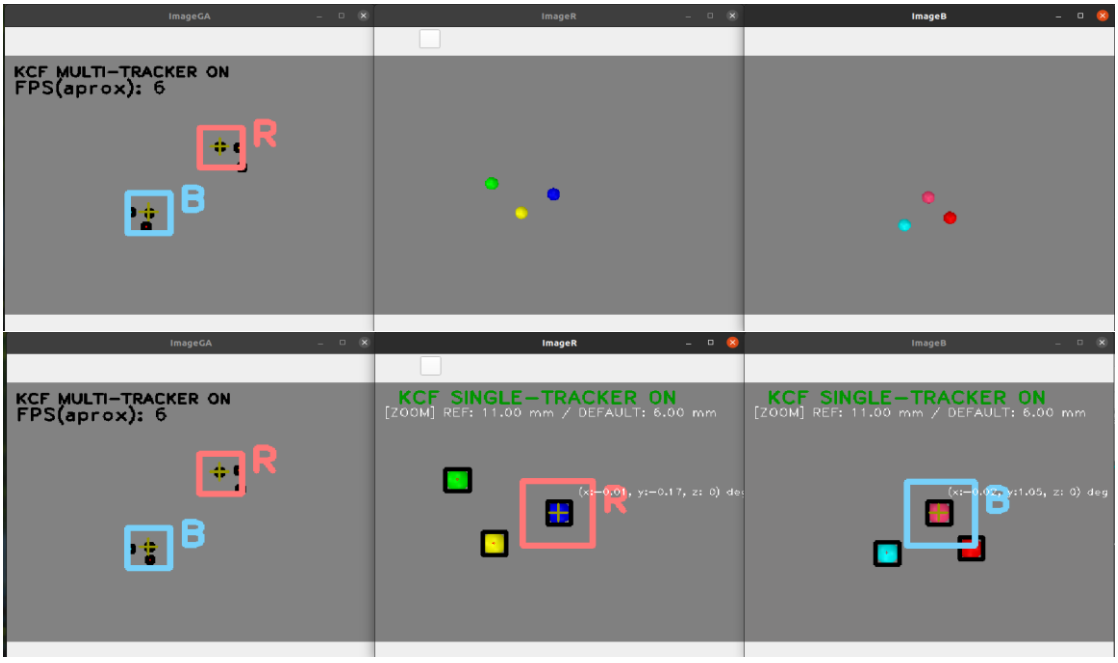


Figura 4-28 Comparativa entre fase de apuntamiento aproximado y fase de seguimiento incluyendo zoom. El área deseada para el objetivo a seguir es de 600 $píxeles^2$.

Vemos de nuevo cómo la identificación y seguimiento se realizan correctamente, sin embargo, ahora el objetivo de interés es claramente apreciabe con mejor resolución.

Aunque podría aumentarse el área de dicho objetivo para que aparezca con mayor tamaño, también hay que tener en cuenta que existe un compromiso con conocer los alrededores del objetivo, es decir, puede que no sólo interese ver al objetivo con el mayor tamaño posible sino también ver las entidades de su entorno. De ahí que se busque un punto intermedio entre el tamaño que se busca del objetivo y la información que se mantiene de su entorno a través de los cambios de zoom.

5 MODELADO DE SISTEMAS

Otro problema que se debe resolver es cómo modelar las dinámicas de los diferentes elementos del sistema completo. Por un lado, se quiere describir los comportamientos para poder simularlos, y por otro, se necesitan unos modelos en base a los cuales diseñar las estrategias de control que veremos en el siguiente apartado. Por tanto, en este apartado se van a explicar las decisiones de diseño tomadas y se especificará qué modelos tomaremos en cada caso.

Cierta parte del estudio de este apartado se ha realizado con el software *Matlab 2023a* y *Simulink* que ha facilitado calcular y simular los modelos cómodamente.

La idea es obtener unos modelos en tiempo continuo, que se descretizarán con un cierto tiempo de muestreo fijo permitiéndonos simular los comportamientos a través de código en *Python* en el entorno de *CoppeliaSim.*

Es importante destacar que se pretende que el desarrollo sea adaptable para cualquier futuro sistema, por lo que, aunque en este caso se van a fijar unos comportamientos concretos, todo lo explicado será adaptable a cualquier otra dinámica que pueda ser modelada por una función de transferencia.

Hay que recalcar también que la finalidad aquí es lograr unos modelos simplificados que permitan diseñar el control y realizar diferentes simulaciones para verificar el adecuado funcionamiento de todos los subsistemas en conjunto.

La razón principal de hacer uso de estas dinámicas modeladas es lograr que el proyecto esté desacoplado de un sistema particular, y simplemente cambiando las ecuaciones que definen las dinámicas se pueda adaptar de la manera más simple posible el sistema a otro caso particular, habilitando la profundización elaborando modelos más complejos y realistas.

5.1 Discretización de modelos

En este apartado, surge la necesidad de discretizar las dinámicas descritas por funciones de transferencia continuas, ya que como se ha comentado, se querrá poder implementarlo en una simulación que se ejecute con paso fijo.

La idea es obtener sus modelos en descripción interna y discretizarlos. De esta forma, se trabajará en el espacio de estados, se simplificarán las operaciones, reduciéndose a multiplicaciones matriciales de bajo orden que son implementables de manera eficiente en *Python* a través del módulo *Numpy.*

Sin embargo, para obtener las matrices *G, H, C y D* a partir de cierta función de transferencia continua, se utiliza el **Código 5-1** en Matlab, estas matrices se guardarán en un directorio y serán implementadas en *Python.* Es decir, el proceso de discretización y obtención de matrices de espacio de estados se realiza en Matlab y se implementa en *Python.*

Sea *G(s)* una función de transferencia que modela el comportamiento deseado, y *Tm* el tiempo de paso fijo en la simulación, se querrán obtener las matrices *G, H, C y D* del espacio de estados discreto.

Por tanto, es interesante introducir brevemente el espacio de estados, para contextualizar la resolución adoptada y comprender mejor las matrices buscadas.

Considerando una descripción interna del sistema tal que:

$$\dot{x}(t) = f\big(x(t), u(t)\big) \;;\; y(t) = g\big(x(t), u(t)\big) \qquad (5\text{-}1)$$

Donde todas las variables pueden ser vectoriales, habilitando el modelado de un sistema con múltiples entradas y salidas (*MIMO, Multiple Inputs Multiple Outputs*). En esta expresión, '$y(t)$' y '$u(t)$' hacen referencia a la salida y entrada del sistema, pero se está introduciendo además la variable '$x(t)$', que es la variable de estados. Determina la evolución del sistema, y en ciertas aplicaciones puede tener sentido físico.

Las relaciones *f y g* pueden presentar alto carácter no lineal, por lo que deben ser sometidas a un proceso de linealización entorno a un punto de funcionamiento, pasando ahora a la siguiente expresión, donde estas variables serían ahora incrementales respecto al punto de operación:

$$\dot{\bar{x}}(t) = A\bar{x}(t) + B\bar{u}(t) \;\; ; \;\; \bar{y}(t) = C\bar{x}(t) + D\bar{u}(t) \qquad (5\text{-}2)$$

Para sistemas con salida y entrada única, (*SISO, Single Input Single Output*), la expresión anterior es equivalente a una función de transferencia, justo como ocurre en este proyecto. Luego, por tanto, el primer paso sería obtener las matrices *A, B, C y D* dada una función de transferencia continua que modele el comportamiento deseado.

Las expresiones en (5-2) deben ser discretizadas, quedando de la siguiente manera:

$$x(k+1) = Gx(k) + Hu(k) \;\; ; \;\; y(k) = Cx(k) + Du(k) \qquad (5\text{-}3)$$

Donde *G* y *H* son dependientes del tiempo de muestreo y se definen analíticamente como:

$$G(T_m) = e^{A \cdot T_m} \;\; ; \;\; H(T_m) = (e^{A \cdot T_m} - I) \cdot A^{-1} \cdot B \qquad (5\text{-}4)$$

Para resolver esta conversión se plantea una función en *Matlab* la cual a partir de una *G(s)* y T_m devuelve las matrices deseadas y además representa en una gráfica la respuesta del modelo continuo y la respuesta discretizada, equivalente a lo que se busca para la simulación. Dicha función guardará además las matrices *G*, *H, C* y *D* en unos archivos en formato *CSV* para poder tener acceso a las mismas en la implementación de la simulación. Podemos ver en la **Figura 5-1** el resultado de discretizar para T_m = 0.01s la función de transferencia *G(s)* = 1/s frente a un escalón unitario y en el **Código 5-1** el fragmento de la función de *Matlab* que realiza la discretización explicada.

Código 5.1 Discretización de sistemas.

```
% Cálculo de las matrices en espacio de estados discreto
% dada una función de transferencia continua:

function [G,H,C,D] = calculoGHCD(G_sist, Tm, str)

% ------------------------------------------
% ----- CÁLCULO DE LAS MATRICES ------------
[num, den] = tfdata(G_sist, 'v');

% Pasar a espacio de estados
[A,B,C,D] = tf2ss(num, den);

% Discretizar:
[G,H] = c2d(A,B,Tm);
```

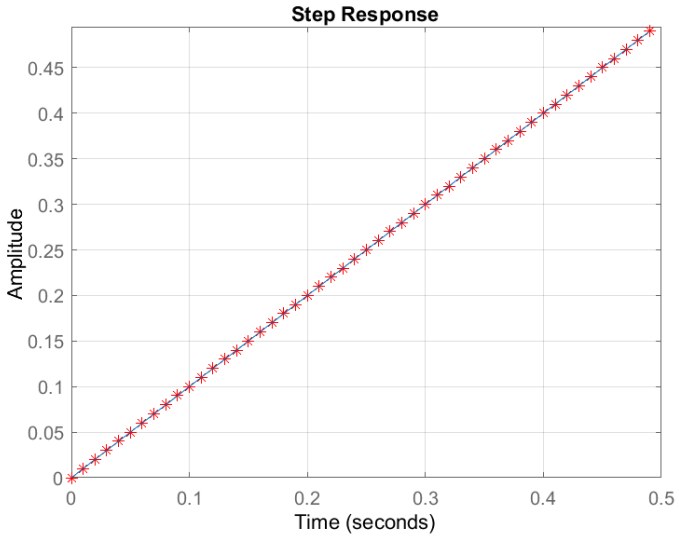


Figura 5-1. Discretización de "1/s" con Tm = 0.01s evaluado con un escalón unitario.

5.2 Dinámica de los gimbals

En este apartado se busca obtener un modelo que simule el comportamiento de los gimbals. Para ello, se puede partir de las conclusiones obtenidas en [34] que consiste en un *Trabajo Fin de Grado* llevado a cabo con el departamento de *Ingeniería de Sistemas y Automática* de la *Universidad de Sevilla* que modela y diseña el control para un quadrotor de un gimbal.

Al igual que en [10], se opta directamente por usar las funciones de transferencia obtenidas en dicho análisis, aunque es interesante explicar brevemente como surge el modelo que se asume y que simplificaciones implica.

Estos gimbals estarán dotados por 3 grados de libertad. Teniendo en cuenta que un gimbal no deja de ser un caso particular de un brazo robótico cuyo objetivo principal es fijar la orientación de su efector final a una orientación particular, el análisis se realiza basándose en modelado de robótica manipuladora.

Los pasos que se llevan a cabo para modelar el sistema buscado son:

- Aplicación del algoritmo de *Denavit-Hartenberg* para desarrollar el modelo cinemático de la cadena de articulaciones completa. Como resultado se obtienen la posición y orientación del efector final a partir de los ángulos de giro de cada articulación.
- Aplicación del algoritmo de *Newton-Euler* para obtener las ecuaciones que rigen el comportamiento dinámico del gimbal. A partir del par motor de las articulaciones, de la posición y velocidad articulares actuales, se puede conocer la aceleración que se le aplica a cada articulación, y por tanto la evolución dinámica del sistema.

En este caso, interesa especialmente el modelo dinámico inverso del sistema, que se expresa mediante:

$$\ddot{q} = (M(q) + R^2 \cdot J_m)^{-1} \cdot (\tau_m - V(q,\dot{q}) + R^2 \cdot B_m \cdot \dot{q}) - G(q) - F(q,\dot{q})) \quad (5\text{-}5)$$

$$\ddot{q} = M_A(q)^{-1} \cdot (\tau_m - V_A(q,\dot{q}) - G_A(q) - F(q,\dot{q})) \quad (5\text{-}6)$$

Este modelo puede ser reducido mediante un conjunto de aproximaciones:

- Considerando velocidades bajas, lo que permite anular el vector de Coriolis: $V(q, \dot{q}) \approx 0$.
- No se consideran fricciones estáticas: $F(q, \dot{q}) \approx 0$.
- No se considera el efecto de la gravedad en el modelo, al asumir un buen equilibrado mecánico de la cámara sobre el gimbal: $G(q) \approx 0$.
- Los motores se consideran de accionamiento directo: $R = 1$.

Esta serie de términos que se han estimado como nulos o simplificado su valor, en la realidad o simulaciones más realistas entrarían como perturbaciones en este modelo simplificado.

Este modelo reducido se linealizará en torno a un punto de operación caracterizado por tener velocidades nulas. Es decir, este modelo será válido siempre y cuando las velocidades de operación no sean muy altas.

Ahora, la matriz ampliada de inercias $M_A(q)$ se considerará diagonal, produciendo así un desacople de las articulaciones. Esta será una aproximación válida cuanto más equilibrado se encuentre el gimbal.

Como resultado de este proceso de simplificación del modelo, se obtiene una función de transferencia que modela cada articulación del gimbal, es decir, cómo evoluciona la posición angular de cada articulación frente al par motor:

$$G_i(s) = \frac{Q_i(s)}{\tau_i(s)} = \frac{1}{s \cdot (M_{A_{ii}}^{eq} s + B_m)} \quad (5\text{-}7)$$

Para continuar, se va a particularizar a unos datos concretos, siendo estos los del sistema del proyecto citado anteriormente en [34]. Sin entrar a recalcar las masas de los eslabones, inercias, características de los motores, etc., se va a proceder a tomar directamente los modelos resultantes, que es lo fundamentalmente relevante para este proyecto.

$$G_1(s) = \frac{1}{s \cdot (6.3 \cdot 10^{-4} s + 1.35 \cdot 10^{-5})} \quad (5\text{-}8)$$

$$G_2(s) = \frac{1}{s \cdot (5.2 \cdot 10^{-4} s + 1.35 \cdot 10^{-5})} \quad (5\text{-}9)$$

$$G_3(s) = \frac{1}{s \cdot (1.8 \cdot 10^{-4} s + 1.35 \cdot 10^{-5})} \quad (5\text{-}10)$$

Sin profundizar ahora más en estos modelos, se analizará su comportamiento en bucle cerrado en el siguiente capítulo.

5.3 Dinámica del UAV

En cuanto al movimiento del UAV, es necesario considerar unas dinámicas en su movimiento, ya que este deberá desplazarse para ser posicionado encima de los puntos deseados. Para ello, se parte de los modelos obtenidos en el proyecto [10].

La principal relevancia de este proyecto recae en el análisis de las imágenes, por tanto, al asumir que las cámaras irán montadas sobre los gimbals, se estará cancelando gran parte del efecto del alabeo y cabeceo del dron durante su movimiento de vuelo. Además, se considerará que no se van a realizar maniobras bruscas, y

que la inclinación que tomará el sistema al desplazarse será reducida.

Considerando estas asunciones, se toma la decisión de diseño de modelar el movimiento del UAV como traslaciones puras en los ejes X, Y, Z.

Además, se va a asumir también que el sistema se encuentra controlado en velocidad, tarea que puede realizar un software de autopiloto[7]. Por tanto, la entrada al sistema será la referencia en velocidad a lo largo del eje deseado, y la salida la posición del UAV siguiendo la dinámica propuesta.

La dinámica escogida para la traslación en cada eje del UAV viene dada por la siguiente función de transferencia:

$$G_{traslacion}(s) = \frac{1}{s \cdot (0.3s + 1)} \qquad (5\text{-}11)$$

Dicha función de transferencia tiene una dinámica como la reflejada en la **Figura 5-2:**

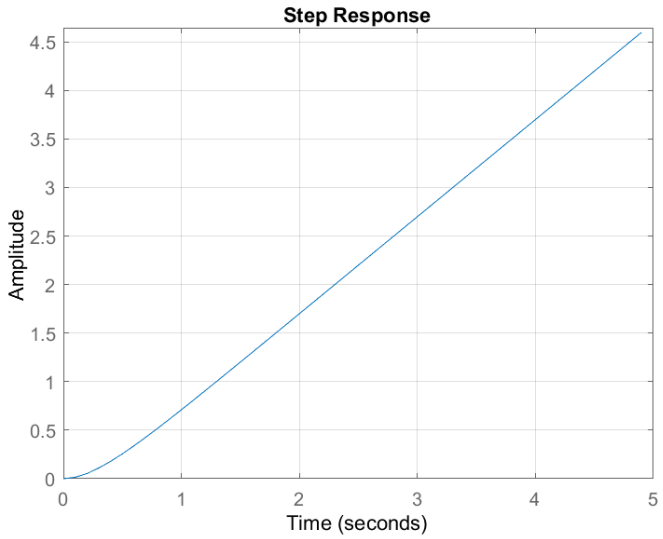


Figura 5-2. Dinámica escogida para el UAV controlado en velocidad frente a escalón unitario.

La referencia, o entrada del sistema controlado, será en velocidad (m/s) y la salida será en posición axial (m).

Se escoge el sistema con $\tau = 0.3$ s ya que consigue que se llegue al 63% de la velocidad de crucero fijada en dicho tiempo. En $3\tau = 0.9$ s aproximadamente, se habrá conseguido que la velocidad que se establezca sea la deseada. Se considera un tiempo aceptable partiendo de las suposiciones previas de que se trabaja con velocidades reducidas.

En el siguiente capítulo veremos cómo se discretiza tanto esta G(s) como su controlador propuesto.

[7] Sistema o software que realiza operaciones y control de vuelo autónomas, permitiendo a la aeronave navegar, mantener estabilidad y hacer determinadas tareas sin necesidad de intervención humana constante.

5.4 Dinámica del zoom óptico

Las cámaras simuladas en *CoppeliaSim* permiten trabajar variando su distancia focal durante la simulación, lo que, debido a su dependencia directa, implica poder modificar el zoom de dicha cámara.

Este zoom óptico, en la mayoría de las cámaras suele implicar un accionamiento mecánico por parte de un pequeño motor, ya que implica una variación de la distancia focal de la lente. Típicamente, se tendrá un objetivo que encapsule una serie de lentes, que modificarán su posición relativa, para conseguir así variar la distancia focal de la lente equivalente a ese conjunto.

En la simulación de CoppeliaSim, al modificar dicha distancia focal, el cambio es instantáneo, lo cual es poco realista. Para mejorar esto, se puede aplicar por encima una dinámica que modele el funcionamiento interno del zoom de una cámara. La idea de esto es fijar una distancia focal de referencia y que este modelo devuelva la distancia focal que debe adoptar la cámara simulada a lo largo del tiempo.

Esta idea procede del desarrollo previo del problema similar del proyecto [10].

El plantemiento es simular un modelo de segundo orden, con una saturación interna, modelando la saturación que tendría el motor en velocidad en la realidad.

Un modelo de segundo orden viene descrito por la siguiente expresión:

$$G(s) = \frac{K}{(\tau_1 s + 1)\cdot(\tau_2 s + 1)} = \frac{K}{\tau_1\tau_2 s^2 + (\tau_1 + \tau_2)s + 1} = \frac{\frac{K}{\tau_1\tau_2}}{s^2 + \frac{\tau_1+\tau_2}{\tau_1\tau_2}s + \frac{1}{\tau_1\tau_2}} \qquad (5\text{-}12)$$

La anterior función de transferencia, a la cual se le incorpora una saturación interna (en rojo en la **Figura 5-3**), puede descomponerse en el siguiente esquema de bloques:

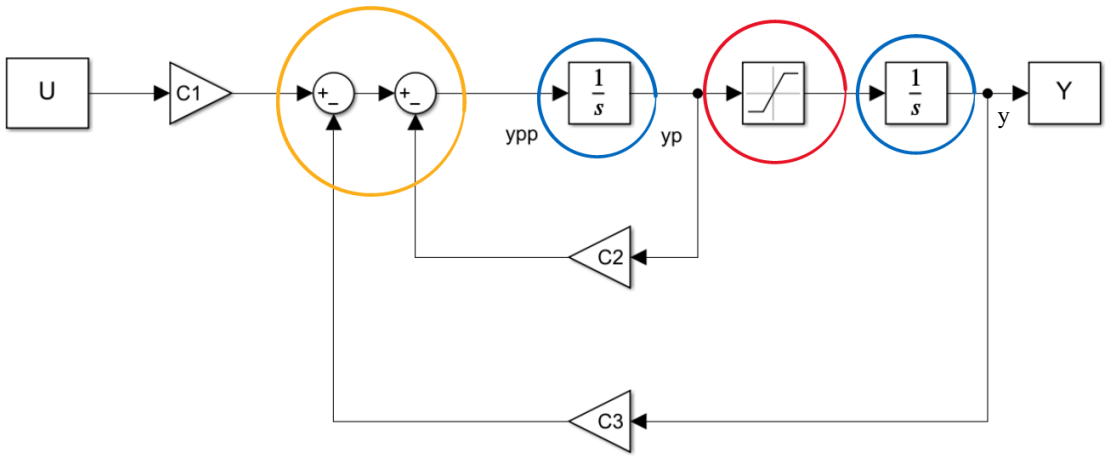


Figura 5-3. Esquema de función de transferencia de segundo orden con saturación interna.

Donde las constantes que aparecen en los bloques triangulares del esquema se definen como:

$$C_1 = \frac{K}{\tau_1 \cdot \tau_2} \quad ; \quad C_2 = \frac{\tau_1 + \tau_2}{\tau_1 \cdot \tau_2} \quad ; \quad C_3 = \frac{1}{\tau_1 \cdot \tau_2} \qquad (5\text{-}13)$$

En este modelo, se estaría saturando $\dot{y}$, es decir, se está modelando una saturación en velocidad dentro del propio sistema.

Esta saturación es una operación no lineal, y no se puede expresar a través de una función de transferencia que sea discretizable con el método anterior. Por tanto, se va a tratar de plantear directamente las ecuaciones discretas.

Se tienen dos integradores, marcados en azul en la **Figura 5-4**, que son los únicos bloques a discretizar. Para hacer esto, se consideran las aproximaciones de *Euler I y Euler II*.

$$\frac{1}{s} \approx \frac{T_m}{Z-1} = \frac{T_m z^{-1}}{1-z^{-1}} = G_{EI}(z)\ (Euler\ I)\ ;\ \frac{1}{s} \approx \frac{ZT_m}{Z-1} = \frac{T_m}{1-z^{-1}} = G_{EII}(z)\ (Euler\ II) \qquad (5\text{-}14)$$

Con esto, para el caso de *Euler I*, se obtendría la siguiente expresión:

$$\frac{y(z)}{u(z)} = \frac{T_m z^{-1}}{1-z^{-1}} \rightarrow y(z) = y(z)\cdot z^{-1} + T_m u(z)\cdot z^{-1} \rightarrow \boldsymbol{y(k) = y(k-1) + T_m u(k-1)} \qquad (5\text{-}15)$$

Mientras que, para el caso de *Euler II*, obtenemos:

$$\frac{y(z)}{u(z)} = \frac{T_m}{1-z^{-1}} \rightarrow y(z) = y(z)\cdot z^{-1} + T_m u(z) \rightarrow \boldsymbol{y(k) = y(k-1) + T_m u(k)} \qquad (5\text{-}16)$$

Entre ambas aproximaciones, se acaba escogiendo la opción *Euler I,* ya que *Euler II* al ser aplicada a este sistema en particular, implica la resolución de la operación inversa a la saturación, lo cual no es posible.

Por tanto, se usará la ecuación (5-23) aplicada a los dos integradores, obteniendo (5-25) para el integrador de la derecha y (5-26) para el de la izquierda, originando el siguiente sistema de ecuaciones:

$$\dot{y}(k) = \dot{y}(k-1) + T_m \cdot \ddot{y}(k-1) \qquad (5\text{-}17)$$

$$y(k) = y(k-1) + T_m \cdot \dot{y_{sat}}(k-1) \qquad (5\text{-}18)$$

$$\dot{y_{sat}}(k) = saturador(\dot{y}(k)) \qquad (5\text{-}19)$$

Además, del diagrama de la **Figura 5-4**, se puede deducir a partir del nodo rodeado en naranja la siguiente relación:

$$\ddot{y}(k) = u(k)\cdot C_1 - \dot{y}(k)\cdot C_2 - y(k)\cdot C_3 \qquad (5\text{-}20)$$

De esta manera, sustituyendo (5-28) en (5-25), se obtiene:

$$\dot{y}(k) = \dot{y}(k-1)\cdot(1 - T_m\cdot C_2) + u(k-1)\cdot C_1\cdot T_m - y(k-1)\cdot C_3\cdot T_m \qquad (5\text{-}21)$$

Esta expresión (5-29) permite obtener la derivada de la salida a partir del valor anterior de la entrada, la de la salida, y de sí misma. Partiendo de condiciones nulas, todo ello es conocido.

Además, sustituyendo (5-27) en (5-26), la salida también queda conocida:

$$y(k) = y(k-1) + T_m \cdot saturador(\dot{y}(k-1)) \qquad (5\text{-}22)$$

Para particularizar el modelo se escogen los siguientes valores:

- $K = 1$, para modelar un seguimiento de referencia.
- $\tau_1 = 0.1$ s
- $\tau_1 = 0.033$ s
- $T_m = 0.01$s

Con esto, se puede simular tanto de manera continua (en *Simulink*) como la aproximación discreta (mediante las ecuaciones previas), la respuesta del sistema ante escalones en su referencia. Se deja un ejemplo de esto en la siguiente **Figura 5-4.**

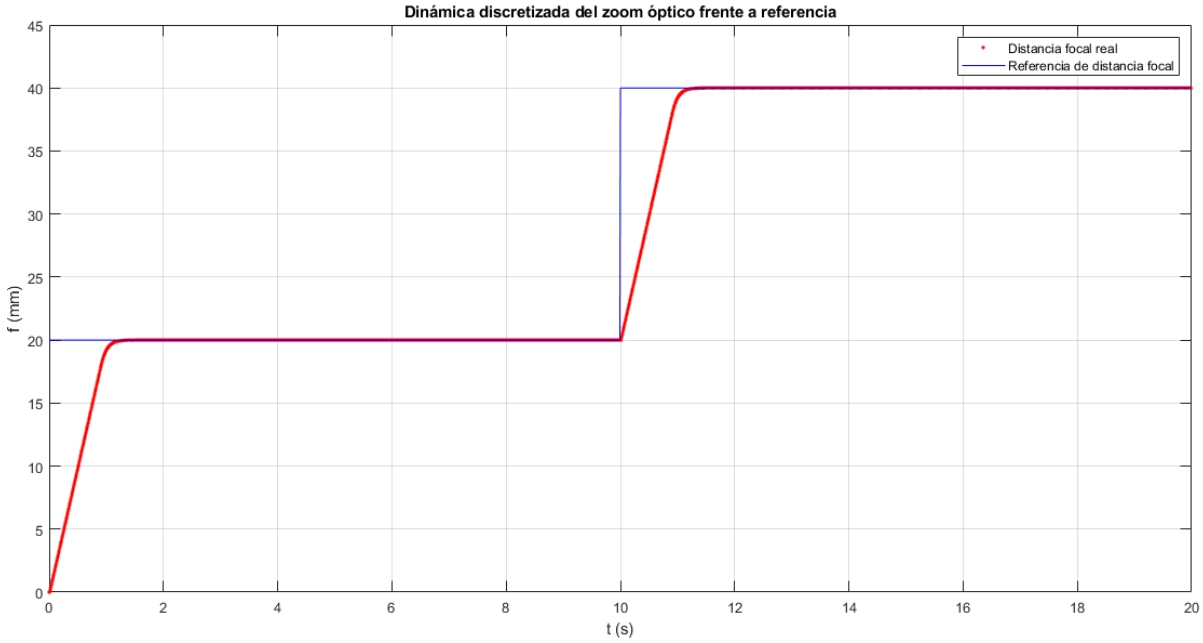


Figura 5-4. Respuesta de la dinámica de control modelada para el zoom ante escalón en su referencia.

Para apreciar mejor como la aproximación discretizada es muy cercana a la salida continua del sistema, se deja un detalle de la gráfica **Figura 5-4** (sin graficar la referencia, aunque es la misma) en la **Figura 5-5**.

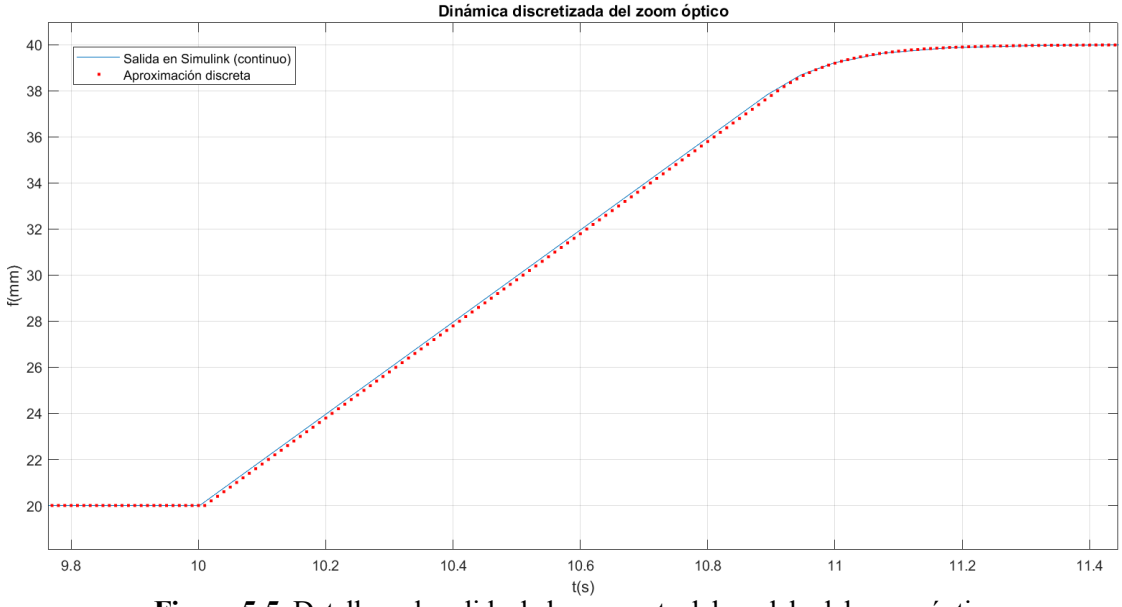


Figura 5-5. Detalle en la salida de la respuesta del modelo del zoom óptico.

Se vuelve a enfatizar en que la salida continua se ha obtenido parametrizando correctamente el modelo de la **Figura 5-3** en Simulink mientras que la salida parametrizada se ha obtenido a través de código de *Matlab* implementando las ecuaciones discretas, cómo se aprecia en el **Código 5-2.**

Código 5.2 Ecuaciones discretas de la dinámica del zoom en *Matlab*.

```
%%%%%%%%%%%%%%%%%%%%%%%%
% Simulación
for k = 2:n

 % Primer Integrador
 yp(k) = yp(k-1)*(1 - T*C2) + u(k-1)*C1*T - y(k-1)*T*C3;

 % Saturador
 if yp(k-1) > ypMax
 ypsat1 = ypMax;
 elseif yp(k-1) < ypMin
 ypsat1 = ypMin;
 else
 ypsat1 = yp(k-1);
 end

 % Segundo Integrador
 y(k) = T*ypsat1 + y(k-1);

end
```

Esta dinámica del zoom no es “puramente” un subsistema controlado, sino como se ha explicado, es una dinámica que se incorpora para hacer el cambio de zoom en las cámaras más realista, por lo que el funcionamiento simplemente consistirá en recibir referencias de zoom según los algoritmos explicados en el capítulo anterior para ambos modos (individual y clustering) e ir devolviendo y aplicando los valores de zoom que dicte la dinámica según el tiempo transcurrido desde que se recibe dicho cambio de referencia.

6 ESTRATEGIAS DE CONTROL

Tras haber definido los diferentes modelos que componen nuestro sistema, falta diseñar las estrategias de control. La mayor parte de este capítulo corresponde a comentar el control dinámico, es decir, los controladores responsables de que los subsistemas con las dinámicas del capítulo anterior (a excepción del zoom óptico) funcionen siguiendo a la referencia o regulando el error.

Además, también se menciona un planificador de trayectorias que se aplicará al controlador de movimiento del UAV cuando el error sea superior a cierto umbral, evitando así acciones excesivamente bruscas cuando el UAV esté suficientemente lejos del centro del cluster global de objetivos.

Esta situación se dará típicamente al iniciar dicho control, donde dependerá de la geometría de los objetivos y el propio UAV, así como en el caso de que se perdiera uno o más objetivos en la imagen de la cámara gran angular, lo que haría que cambiase el centro de dicho cluster global.

6.1 Control de los gimbals de 3 ejes

Además de los sistemas modelados en el capítulo anterior, también se van a reutilizar los controladores diseñados en [34] para dichas articulaciones. A través del algoritmo del lugar de las raíces, este proyecto concluye en estos controladores, que son los que se considerarán en este proyecto:

$$C_1(s) = 2.7776 \cdot (1 + \frac{1}{0.112s} + 0.028s) \quad (6\text{-}1)$$

$$C_2(s) = 2.2915 \cdot (1 + \frac{1}{0.112s} + 0.028s) \quad (6\text{-}2)$$

$$C_3(s) = 0.7952 \cdot (1 + \frac{1}{0.112s} + 0.028s) \quad (6\text{-}3)$$

Estos controladores son probados en dicho proyecto, y presentan el siguiente comportamiento en bucle cerrado:

- Tiempo de subida: $t_s \approx 0.015\ s$
- Sobreoscilación: $SO \approx 18\%$
- Tiempo de estabilización: $t_e \approx 0.072\ s$

Al calcular las G_{BC_i} se apreciaba la aparición de picos en la respuesta, concluyendo ante esto en [34] en la aplicación del siguiente prefiltro a las referencias:

$$G_f(s) = \frac{23.5}{s + 23.5} \quad (6\text{-}4)$$

Si se considera la función de transferencia del sistema completo (con el prefiltro) y se aplica a las tres articulaciones, se obtienen los siguientes modelos:

$$G_{mod_i}(s) = G_f(s) \cdot G_{BC_i} = G_f(s) \cdot \frac{C_i(s) \cdot G_i(s)}{1 + C_i(s) \cdot G_i(s)} \quad (6\text{-}5)$$

$$G_{mod_1}(s) = \frac{123.5s^2 + 4410s + 3.938 \cdot 10^4}{s^3 + 123.5s^2 + 4408s + 3.93 \cdot 10^4} \quad (6\text{-}6)$$

$$G_{mod_2}(s) = \frac{123.5s^2 + 4407s + 3.936 \cdot 10^4}{s^3 + 123.4s^2 + 4406s + 3.934 \cdot 10^4} \quad (6\text{-}7)$$

$$G_{mod_3}(s) = \frac{123.4s^2 + 4407s + 3.936 \cdot 10^4}{s^3 + 123.3s^2 + 4403s + 3.93 \cdot 10^4} \quad (6\text{-}8)$$

Estos últimos modelos se corresponderían al comportamiento del sistema ya controlado. Es decir, la entrada sería la referencia, y la salida, la evolución de la variable articular, correspondiente al sistema ya controlado.

Para ver la respuesta ante escalón unitario de $G_{mod_1}(s)$, se incluye la **Figura 6-1**, donde se puede ver su comportamiento continuo y superpuesto la discretización obtenida según lo explicado en el apartado 5.4 del capítulo anterior.

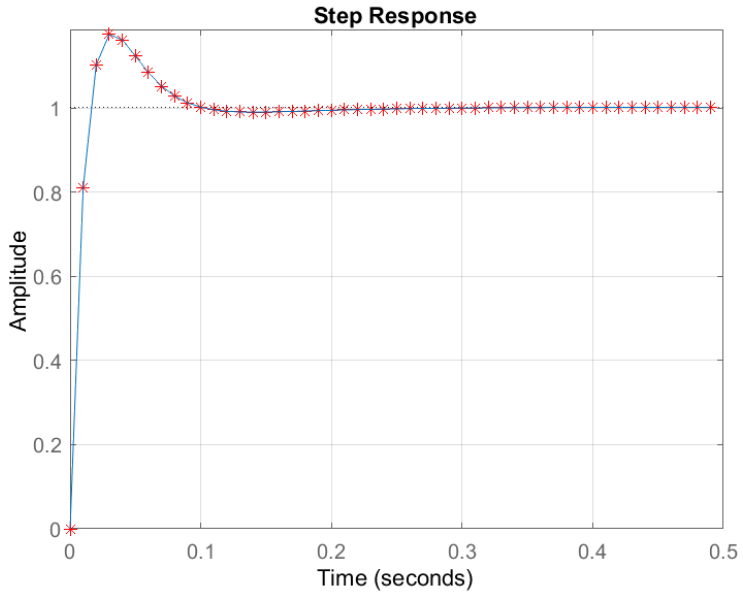


Figura 6-1. Dinámica discretizada con Tm = 0.01s de una articulación del gimbal en el sistema ya controlado.

Como se dijo al principio del apartado 5.2, una de las finalidades principales de estos modelos es tener la capacidad de diseñar estrategias de control en base a ellos. Además, es importante recalcar que los modelos $G_i(s)$ (descritos en el capítulo 5) describen el comportamiento de los ejes del gimbal frente a par motor, y los modelos $G_{mod_i}(s)$ describen la respuesta ya controlada en posición angular.

Tras haber propuesto los controladores y sistemas que controlarán a los gimbals de las cámaras orientables, es importante también explicar su propósito.

El propósito de controlar los gimbals es lograr que su cámara asociada realice un apuntamiento al punto que le ha sido asignado, sea este el punto representativo de la agrupación o el centro del objetivo individual en cuestión.

Para lograr un correcto apuntamiento, este control se divide en dos etapas diferentes en las que se aplican diferentes controladores. Para distinguirlos, tendremos en primera instancia un control en posición angular absoluta y tras este, un segundo control de regulación.

La dinámica de los gimbals se considera ya controlada en posición angular, de tal manera que el primer control, el control en posición angular absoluta, se encarga de fijar la orientación deseada del gimbal (rotación de cada grado de libertad del gimbal).

Por su parte, el segundo control, entrará a actuar una vez que el primero ya haya realizado una actuación previa y, por tanto, un primer apuntamiento aproximado del objetivo en cuestión. Partiendo de este apuntamiento, que aproximadamente dejará al objetivo centrado en la imagen con cierto error, el control de regulación buscará reducir el error que exista entre el punto a seguir y el centro de la imagen, tratando de lograr un apuntamiento más preciso.

Para comprender el uso de esta estrategia de control en dos etapas, hay que recordar lo comentado en el apartado 4.2, donde se explicaba cómo obtener los ángulos para apuntar la cámara hacia el objetivo.

La primera etapa de este control tomará como referencia los ángulos a girar para cada eje del gimbal a partir de la imagen de la cámara gran angular, obtenidos en el apartado 4.2.1, resultando este en un apuntamiento susceptible a errores por la estimación de la altura e imprecisiones en el modelo aplicado de la cámara. El objetivo principal de esta etapa de control es conseguir apuntar de manera aproximada al objetivo en cuestión con la precisión suficiente como para que sea identificable en la imagen de la cámara que lo apunta, es decir no se busca un apuntamiento fino.

Para la segunda etapa del control, se parte de la premisa de que ya se está apuntando aproximadamente al objetivo y se ha realizado además la identificación del mismo en la imagen. Ahora, se calcularán los ángulos a girar para recentrar el punto identificativo a seguir, acorde a lo explicado en el apartado 4.2.2. Este control recibe el nombre de control en regulación ya que los ángulos a girar en cada eje corresponden al error cometido en el apuntamiento, los cuales el controlador buscará llevarlos a cero. Con este control sí se busca un apuntamiento fino, cerrando ahora el bucle de control con lo que ve la propia cámara que realiza este apuntamiento fino.

Para la primera etapa de control, se toma como base el control ya realizado en el proyecto [34], siendo que su funcionamiento no es dependiente del análisis de la imagen que ve la propia cámara que a su vez controla su propia orientación, por tanto, es más que válido de manera directa.

Sin embargo, el control de regulación dependiente de la visión de la cámara sí debe ser desarrollado y explicado ya que aporta una nueva funcionalidad al control de un gimbal.

La idea, por tanto, es dejar actuar la primera etapa durante un cierto tiempo para garantizar ese primer apuntamiento aproximado del objetivo deseado, o bien si hay pérdidas en el seguimiento de objetivos y se requiere reinicio de subsistemas.

Una vez realizado ese apuntamiento inicial, entonces se conmutará a la segunda etapa de este control, donde se producirá un apuntamiento más fino realimentado con la propia imagen que proporciona la cámara orientable.

De esta manera, ambos se suceden secuencialmente (nunca actúan simultáneamente) para garantizar el mejor apuntamiento posible del objetivo asignado.

6.1.1 Primera etapa: Control en posición absoluta

Se toma para este sistema el desarrollo realizado en el apartado 6.1, donde se modela en las ecuaciones (6-6), (6-7) y (6-8) el comportamiento del sistema controlado con el esquema de control realizado en el trabajo [34].

Se toman los tres modelos $G_{mod_i}(s)$ desacoplados como la dinámica que seguirían nuestros gimbals. Se le dará como referencia la orientación a través de los ángulos de giro respecto de cada eje, y se obtendrá el valor articular de cada una de las articulaciones, por tanto, la orientación que debe seguir la cámara a lo largo del tiempo.

Estos tres modelos se discretizan de acuerdo a lo mencionado en el apartado 5.1, con el formato matricial del espacio de estados, y se simula como un sistema conjunto de la siguiente manera:

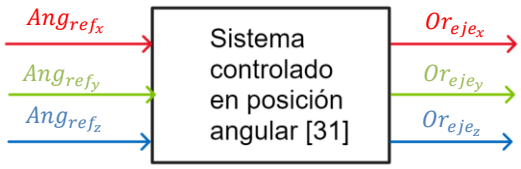


Figura 6-2. Gimbal controlado en posición angular.

Realmente, el control de la **Figura 6-2** corresponde a un control cinemático en bucle abierto. Según los modelos ya descritos y ya controlados de (6-6), (6-7) y (6-8), para cierto valor de ángulo de referencia de cada articulación se obtendrá una orientación de salida para cada ejeen función de las dinámicas descritas

6.1.2 Segunda etapa: Control de regulación

Como se mencionaba previamente, esta segunda etapa tiene como objetivo colocar el punto identificativo a seguir en el centro de la imagen de aquella cámara encargada de seguirlo.

Se parte de la premisa de que este punto a seguir, a parte de encontrarse contenido en la imagen, debe ser el más cercano al centro de la imagen gracias al apuntamiento previo respecto al resto de objetivos que pudiese haber en la imagen, ya que es a partir de dicha imagen y el mismo punto, de donde se obtienen los ángulos a girar.

Por tanto, a partir de la imagen, se obtienen los ángulos a girar, que se pueden considerar directamente como el error a reducir por parte del controlador. También se podría considerar que el propio análisis de la imagen devuelve la salida del sistema, si se les cambia el signo a esos ángulos, lo que haría referencia a la orientación de la cámara respecto del objetivo, esto queda más claro en el siguiente esquema:

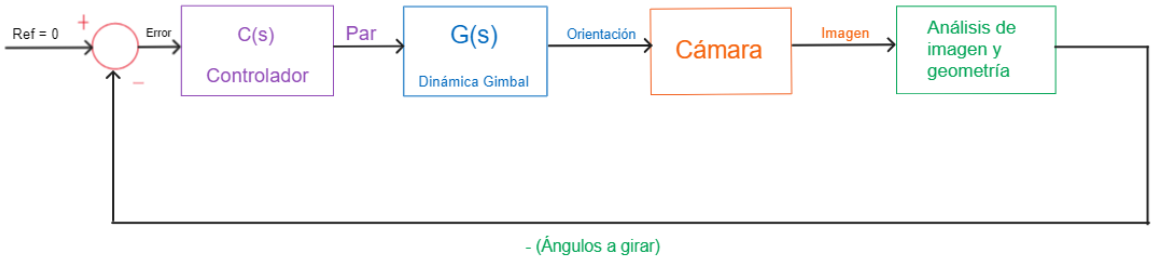


Figura 6-3. Esquema del control de regulación del gimbal.

Las líneas de *Ref, Error, Par, Orientación* y *Ángulos a girar*, se corresponderían con una tripleta de valores, uno por cada una de las articulaciones del gimbal.

El sistema a controlar será el compuesto por los modelos $G_i(s)$ que modelarán cada articulación del gimbal, tal y como se explicaba en el apartado 5.2 y consisten en las ecuaciones (5-8), (5-9) y (5-10).

Por tanto, el controlador de regulación consistirá en los $C_i(s)$ mencionados en el apartado 6.1, correspondientes a las expresiones (6-1), (6-2) y (6-3). El controlador diseñado tiene especificaciones adecuadas para resolver el problema de regulación descrito.

Ahora no se utilizan los prefiltros (6-4) en la referencia, ya que se asume que los errores a corregir serán reducidos gracias al apuntamiento aproximado de la primera etapa previamente comentada.

Por las características del problema a resolver por el control de regulación, se hace fundamental ver su respuesta rechazando perturbaciones, ya que es fundamentalmente lo que tendrá que realizar.

Esta idea se ve clara si se plantea el caso en que la cámara se encuentra estática y el objetivo a seguir, el cual se encontraba en el centro de la imagen, se mueve con aceleración constante, lo cual es equivalente a aplicar en las articulaciones del gimbal un par constante de forma mantenida considerando el objetivo estático. Este ensayo es fácilmente realizable en Simulink y permite comprobar la robustez del controlador frente a su principal tarea.

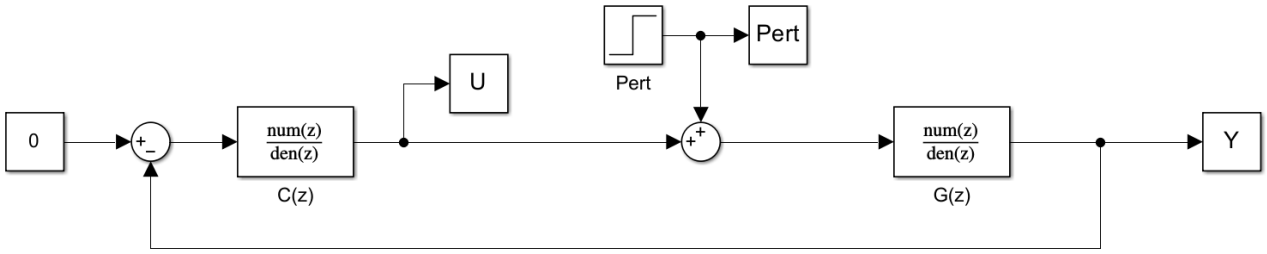


Figura 6-4. Esquema de control de Simulink para comprobar el control de regulación del gimbal frente a perturbación mantenida en el tiempo.

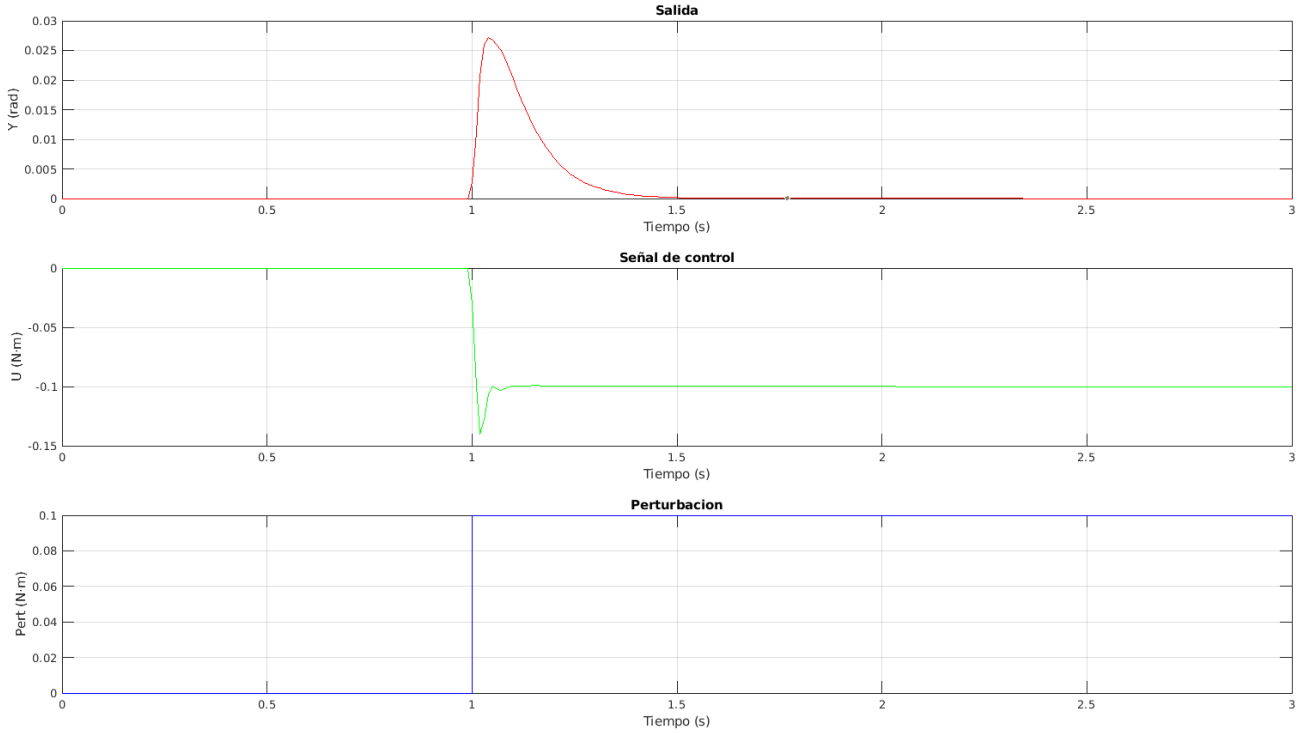


Figura 6-5. Comportamiento del control de regulación frente a perturbación mantenida en el tiempo.

Se aprecia en la simulación de Simulink como el controlador de manera rápida actúa para corregir el efecto de la perturbación mantenida en el tiempo y llevar de nuevo el error a cero. Es importante que esta actuación sea rápida para que los modelos sin prefiltros sean válidos logrando que los errores angulares no tomen valores mayores antes de ser cancelados.

6.2 Control de movimiento del UAV

El objetivo de este control es posicionar al UAV de tal manera que quede justo encima del centro de todos los objetivos avistados con la cámara gran angular, es decir, mediante movimientos del UAV, tratar de hacer coincidir el centro de la imagen de dicha cámara con el centro calculado de todos los objetivos, maximizando el avistamiento de objetivos y evitando potenciales pérdidas de objetivos.

La idea es diseñar un control que permita colocar al dron en ciertas coordenadas dadas respecto del punto objetivo, que en este caso será el centro de todos los objetivos avistados por la cámara gran angular, e idealmente se buscará que estas coordenadas acaben siendo cero.

Conceptualmente, se parte de tener unas coordenadas en la imagen *(u, v)* que corresponden al punto objetivo en la imagen. Asumiendo conocida la altura, que como se dijo en capítulos anteriores es estimable con cierta incertidumbre mediante un altímetro, se puede realizar la proyección inversa para obtener sus coordenadas tridimensionales. Este proceso de cálculo de la proyección inversa se analizó en el apartado 4.2, con las ecuaciones (4-1), (4-2), (4-3) y (4-4).

Al trabajar con las coordenadas tridimensionales del punto objetivo, se logra independencia de la altura o zoom de la cámara, aunque ambas influyen en el cálculo de la proyección inversa, pero esto pasa desapercibido para el control al trabajar de manera directa con el resultado, las coordenadas métricas espaciales.

Dado que la altura del dron que se fija al principio se mantiene todo el tiempo, interesan especialmente las coordenadas X e Y. La idea es que a las coordenadas obtenidas se les cambia el signo, obteniendo así la posición del centro óptico respecto del punto objetivo.

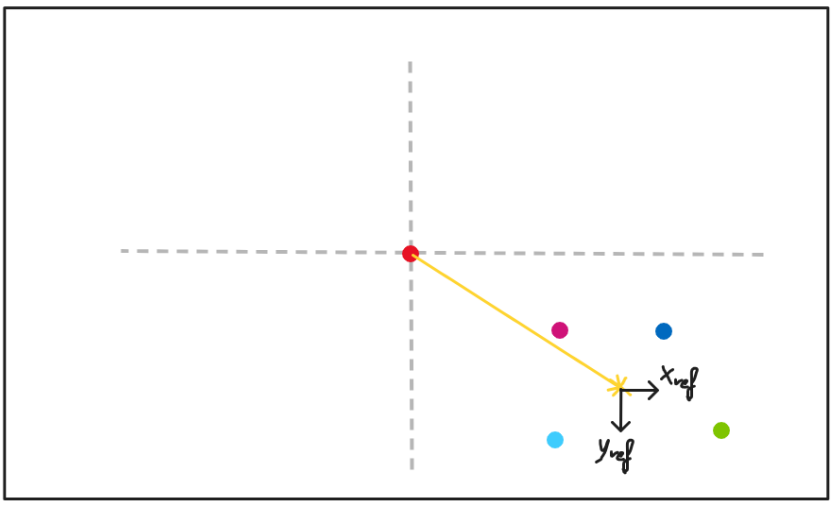


Figura 6-6. Centro óptico (punto rojo) respecto del punto objetivo (cruz amarilla).

En la **Figura 6-6** se representa el plano del terreno en sus unidades métricas obtenidas a partir de la imagen de la cámara gran angular mediante proyección inversa. Se obtiene la posición tridimensional del centro óptico (punto rojo) respecto del punto objetivo (cruz amarilla), obtenido como centro del grupo de objetivos avistados, con el procedimiento explicado en el apartado 4.3.1.

La referencia por tanto para el sistema de control es la magnitud del vector amarillo de la figura, es decir, donde queremos posicionar el centro óptico respecto del punto objetivo.

Para diseñar el control, se parte de que el UAV ya está controlado en velocidad (típicamente mediante software autopiloto). Esto estaba modelado en el apartado 5.3 con la ecuación (5-11).

La idea entonces es aplicar el controlador dentro de un control en cascada, teniendo unas especificaciones de diseño más lentas que la dinámica del UAV controlado en velocidad para lograr un funcionamiento correcto y coherente.

Dicho control en cascada tomaría la forma descrita en el esquema de la siguiente **Figura 6-7**:

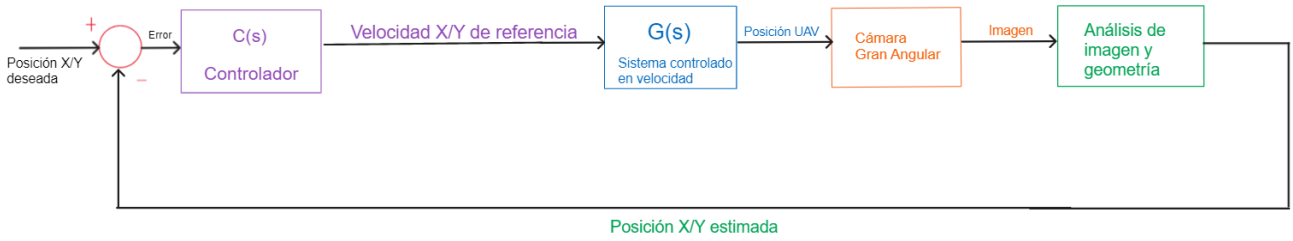


Figura 6-7. Esquema de lazo de control externo del UAV para el desplazamiento en eje X/Y.

Como se ha mencionado, las especificaciones de control serán holgadas, para evitar cambios bruscos en el dron, lo cual provocaría inclinaciones bruscas que no casarían con las suposiciones realizadas en el modelado.

Se realiza un diseño basado en la respuesta frecuencial del modelo escogido, escogiendo los siguientes parámetros:

- Tiempo de subida: $t_s \approx 1\ s$. Esto conlleva una frecuencia de control: $\omega_c \approx \frac{\pi}{t_s} = \pi\ rad/s$
- Margen de fase: $M_f \approx 90º$

Para cumplir estas especificaciones se opta por el siguiente controlador de tipo PID:

$$C(s) = 1.741 \cdot \frac{(0.8296 + 1)^2}{s} = \frac{1.1981s^2 + 2.888s + 1.741}{s} \qquad (6\text{-}9)$$

Gracias a los dos ceros del numerador se logra aportar la fase necesaria y la constante que multiplica fija la frecuencia de control a la deseada.

Para comprobar el cumplimiento de las especficaciones propuestas, se puede observar la respuesta frecuencial (bode) de $C(s) \cdot G(s)$, o G de bucle abierto, en la **Figura 6-8**:

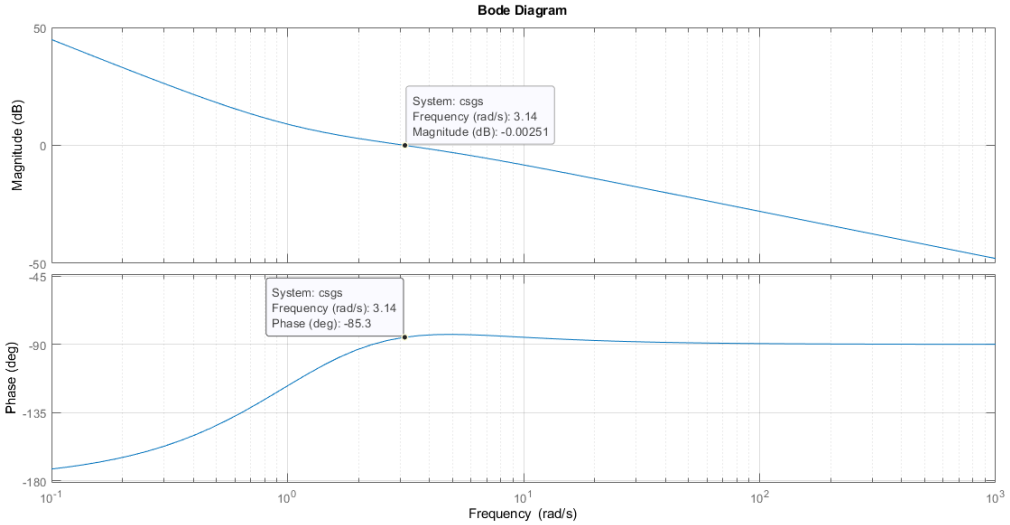


Figura 6-8. Respuesta frecuencial de la G de bucle abierto para el control de desplazamiento del UAV.

Finalmente, es interesante también obtener la respuesta del sistema en bucle cerrado, por ejemplo, frente a un escalón unitario en la referencia. Para ello se plantea el siguiente esquema en Simulink:

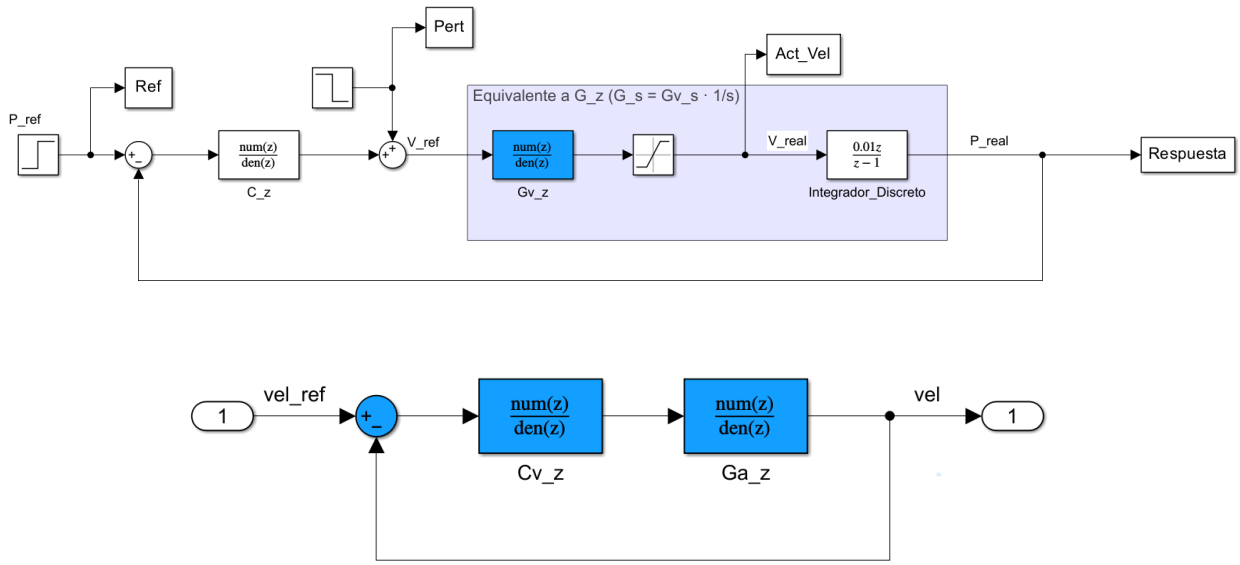


Figura 6-9. Esquema de Simulink para comprobar la respuesta del sistema de control de posicionamiento del UAV frente a escalón unitario en la referencia y perturbación mantenida en el tiempo.

Notar que el lazo inferior equivale a lo que está modelado directamente en la "Gv_z" del lazo superior.

Donde se obtiene la siguiente respuesta tras simular:

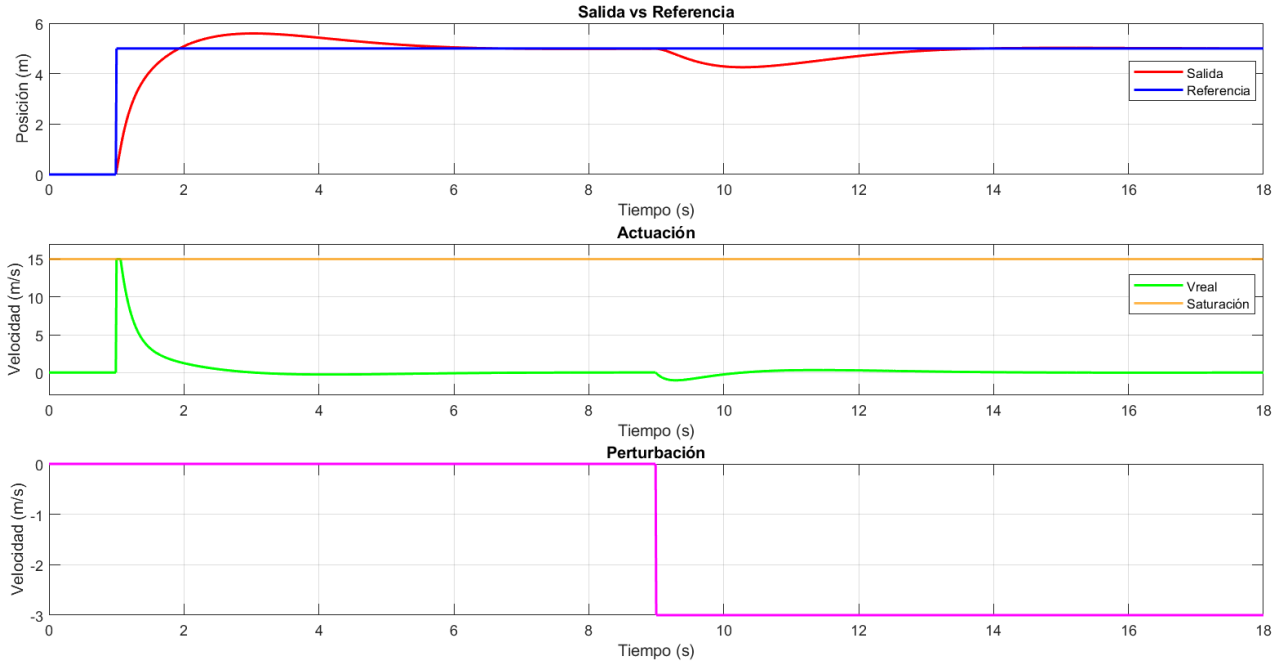


Figura 6-10. Respuesta del control de desplazamiento del UAV en bucle cerrado frente a escalón unitario en la referencia y perturbación mantenidos en el tiempo.

Observando la respuesta de la **Figura 6-10**, vemos que hay una ligera sobreoscilación. Sin embargo, con la colaboración del planificador de trayectorias cuando el cambio de referencia es relativamente grande, se consigue un movimiento suave del UAV que es lo que se buscaba con esta estrategia de control.

Finalmente, este sistema de control logrará posicionar al UAV en unas coordenadas concretas respecto del punto objetivo, donde idealmente estas deberán tender a cero incluso a pesar del movimiento del conjunto de

objetivos, para lograr así mantener al dron en una posición ventajosa para mantener el avistamiento de todos los objetivos y evitar en la medida de lo posible la pérdida de alguno de ellos en la imagen de la cámara gran angular.

Se observa también como ante una perturbación externa en la velocidad del UAV, que podría ser causada por una fuerte ráfaga de viento que variase la velocidad del dron, el control es capaz de rechazarla y volver a llevar al vehículo a la posición de referencia.

6.2.1 Discretización del modelo (G(s)) y controlador (C(s)) del UAV

En este apartado, se ha tomado la decisión de discretizar de manera diferente a la explicada en el apartado 5.1 (mediante espacio de estados). En su lugar, se hace la discretización a partir de la propia función de transferencia continua.

Esta técnica aporta la ventaja de lograr tener una discretización desacoplada del modelo ya controlado en velocidad o G(s) y de su controlador en cascada que entrega referencias de velocidad o C(s), con el objetivo de que, si se quisieran modelar de manera diferente o utilizar otro controlador, el procedimiento a seguir fuese similar pero aplicando diferentes funciones de transferencia que puedan modelar dinámicas más realistas o complejas.

Si se tratase de aplicar discretización usando Matlab, aparece el problema de que C(s) resulta ser impropia y por tanto no es inmediatamente discretizable con comandos de Matlab.

La primera alternativa sería añadir un polo de alta frecuencia al controlador, como por ejemplo (s+10000), cuyo efecto sea despreciable en las frecuencias de interés y solucionaría ese carácter impropio de la función de transferencia.

Otra alternativa sería discretizar conjuntamente la G de bucle cerrado, es decir C(s) · G(s), con lo que queda una función de transferencia propia y válida para ser discretizada con lo explicado en el apartado 5.1 o directamente mediante comandos de Matlab. Sin embargo, esto haría más compleja una posible modificación del modelo controlado en velocidad del UAV (G(s)) o bien de su controlador, al estar en esta solución ambas acopladas en una misma función de transferencia discretizada.

Ambas alternativas han sido implementadas y resultan funcionar adecuadamente controlando de la manera explicada en el apartado 6.2.

Sin embargo, cabe la posibilidad de realizar una implementación más desacoplada y adecuada a esta solución, es decir, discretizar separadamente de manera teórica G(s) y C(s) e implementar programáticamente sus ecuaciones en diferencias resultantes.

Discretización de G_v(s) o modelo del UAV ya controlado en velocidad

Para discretizar la G(s), se utiliza como método de discretización "zoh" (Zero-Order Hold), o mantenedor de orden cero, que permite una implementación simple, y además en este caso si es discretizable con la función "c2d" de MATLAB que es la técnica que usa por defecto e indicándole el tiempo de muestreo Tm = 10 ms.

Es importante destacar que para seguir la forma de implementación realizada en un modelo de Simulink que sirve de prueba, se ha decidido desacoplar el integrador que incluye la ecuación de G(s) recogida en (5-11). Por tanto, el integrador se discretiza por separado y la "nueva" G_v(s) a discretizar ahora es la siguiente:

$$G(s) = G_v(s) \cdot \frac{1}{s} \quad \rightarrow \quad G_v(s) = \frac{1}{(0.3s\ + 1)} \tag{6-10}$$

De esta manera, con dos llamadas en MATLAB del tipo: "gvs = tf([1], [0.3, 1])" y "c2d(gvs, 0.01)" nos resulta la siguiente función de transferencia discretizada:

$$G_v(z) = \frac{0.03278z}{z - 0.9672} \tag{6-11}$$

Donde para obtener la ecuación en diferencias se opera de la siguiente manera:

$$G_v(z) = \frac{0.03278z}{z - 0.9672} = \frac{0.03278z^{-1}}{1 - 0.9672z^{-1}} = \frac{Y(z)}{U(z)} \quad (6\text{-}12)$$

Con Y(z) la velocidad real del UAV resultante en cierto instante y U(z) la velocidad de referencia recibida en cierto instante.

Agrupando términos:

$$Y(z)(1 - 0.9672z^{-1}) = U(Z)(0.03278z^{-1}) \quad (6\text{-}13)$$

Sabiendo que z = k, $z^{-1} = k - 1, z^{-2} = k - 2$ ... y sustituyendo en la expresión anterior nos queda:

$$Y(k) - 0.9672 \cdot Y(k-1) = 0.03278 \cdot V(k-1) \quad (6\text{-}14)$$

Donde aislando, nos queda que la velocidad para un instante "k" se obtiene tal que:

$$Y(k) = 0.9672 \cdot Y(k-1) + 0.03278 \cdot V(k-1) \quad (6\text{-}15)$$

Será por tanto la expresión (6-15) la que habrá que implementar programáticamente para aplicar el modelo discretizado de G_v(s).

Discretización del integrador

Se repite el proceso utilizado para discretizar la G_v(s), ahora únicamente para el integrador, cuya función de transferencia es directamente I(s) = 1/s.

Discretizando con Tm = 10ms queda:

$$I(z) = \frac{0.01z}{z - 1} = \frac{0.01z^{-1}}{1 - z^{-1}} = \frac{X(z)}{V(z)} \quad (6\text{-}16)$$

Con X(z) la posición del UAV resultante en cierto instante y V(z) la velocidad real en cierto instante.

Agrupando términos:

$$X(z)(1 - z^{-1}) = V(Z)(0.01z^{-1}) \quad (6\text{-}17)$$

Sabiendo que z = k, $z^{-1} = k - 1, z^{-2} = k - 2$ ... y sustituyendo en la expresión anterior nos queda:

$$X(k) - X(k-1) = 0.01 \cdot V(k-1) \quad (6\text{-}18)$$

Donde aislando, nos queda que la posición para un instante "k" se obtiene tal que:

$$X(k) = X(k-1) + 0.01 \cdot V(k-1) \quad (6\text{-}19)$$

Será por tanto la expresión (6-19) la que habrá que implementar programáticamente para aplicar el modelo discretizado de un integrador (I(s)).

Discretización de C(s) o controlador que entrega las referencias de velocidad

Aquí es donde volvemos a esa función de transferencia impropia descrita en la ecuación (6-9). En este caso aplicaremos como aproximación para discretizar la técnica de "Euler Backwards", también conocida como "Euler II".

Para resolver esta tarea de discretización, se puede resolver con la aproximación de Euler II para la expresión genérica de un PID y posteriormente particularizar para nuestro PID concreto. Para ello empezamos escribiendo la función de transferencia típica de un PID y a su lado la transformación de Euler II:

$$C(s) = K_c\left[1 + \frac{1}{T_i s} + T_d s\right] \;;\; Euler\ II \rightarrow s = \frac{z-1}{Tz} = \frac{1-z^{-1}}{T} \qquad (6\text{-}20)$$

Realizando ahora la sustitución de la aproximación de Euler II:

$$C(z) = K_c\left[1 + \frac{T}{T_i}\cdot\frac{z}{z-1} + \frac{T_d}{T}\cdot\frac{z-1}{z}\right] = K_c\left[1 + \frac{T}{T_i}\cdot\frac{1}{1-z^{-1}} + \frac{T_d}{T}\cdot(1-z^{-1})\right] \qquad (6\text{-}21)$$

Sacando el término $(1-z^{-1})$ como factor común:

$$C(z) = K_c\left[\frac{(1-z^{-1}) + \frac{T_d}{T}(1-z^{-1})^2 + \frac{T}{T_i}}{1-z^{-1}}\right] \qquad (6\text{-}22)$$

Se extrae fuera de los corchetes para simplificar:

$$C(z) = \frac{K_c}{(1-z^{-1})}\left[1 - z^{-1} + \frac{T}{T_i} + \frac{T_d}{T}(1-z^{-1})^2\right] \qquad (6\text{-}23)$$

Se desarrolla el cuadrado:

$$C(z) = \frac{K_c}{(1-z^{-1})}\left[1 - z^{-1} + \frac{T}{T_i} + \frac{T_d}{T}(1 + z^{-2} - 2z^{-1})\right] \qquad (6\text{-}24)$$

Finalmente se separan términos independientes y en función de z^{-1} y z^{-2}:

$$C(z) = \frac{K_c}{(1-z^{-1})}\left[\left(1 + \frac{T}{T_i} + \frac{T_d}{T}\right) + \left(-1 - 2\frac{T_d}{T}\right)z^{-1} + \frac{T_d}{T}z^{-2}\right] \qquad (6\text{-}25)$$

Se simplifica la expresión incluyendo q0, q1 y q2 sustituyendo a los términos anteriores:

$$C(z) = \frac{K_c}{(1-z^{-1})}[q_0 + q_1 z^{-1} + q_2 z^{-2}] \qquad (6\text{-}26)$$

Podemos expresar la función de transferencia en función de z o z^{-1}:

$$C(z) = \frac{q_0 z^2 + q_1 z + q_2}{z(z-1)} = \frac{U(z)}{E(z)} \;;\; C(z^{-1}) = \frac{q_0 + q_1 z^{-1} + q_2 z^{-2}}{1 - z^{-1}} = \frac{U(z)}{E(z)} \qquad (6\text{-}27)$$

Al igual que antes, buscamos aislar U(k):

$$U(z)(1 - z^{-1}) = E(z)(q_0 + q_1 z^{-1} + q_2 z^{-2}) \qquad (6\text{-}28)$$

Sabiendo que z = k, $z^{-1} = k - 1, z^{-2} = k - 2$ ... y sustituyendo en la expresión anterior nos queda:

$$U(k) = U(k-1) + q_0 \cdot E(k) + q_1 \cdot E(k-1) + q_2 \cdot E(k-2) \quad (6\text{-}29)$$

Donde:

$$q_0 = \left(1 + \frac{T}{T_i} + \frac{T_d}{T}\right) \cdot K_c \quad (6\text{-}30)$$

$$q_1 = \left(-1 - 2\frac{T_d}{T}\right) \cdot K_c \quad (6\text{-}31)$$

$$q_2 = \left(\frac{T_d}{T}\right) \cdot K_c \quad (6\text{-}33)$$

Queda ahora concretar los valores de q0, q1 y q2 para el PID particular de la ecuación (6-9), para ello primero hay que obtener los parámetros del controlador en forma de PID estándar:

$$T_i = 2 \cdot \tau_c = 2 \cdot 0.8296 = 1.6592 \quad (6\text{-}34)$$

$$T_d = \frac{T_i}{4} = \frac{1.6592}{4} = 0.4148 \quad (6\text{-}35)$$

$$K_c = K_p \cdot T_i = 1.741 \cdot 1.6592 = 2.8887 \quad (6\text{-}36)$$

Ahora ya podemos obtener los valores concretos de q0, q1 y q2 a partir de las expresiones anteriormente obtenidas:

$$q_0 = \left(1 + \frac{T}{T_i} + \frac{T_d}{T}\right) \cdot K_c = \left(1 + \frac{0.01}{1.6592} + \frac{0.4148}{0.01}\right) \cdot 2.8887 = 122.7293 \quad (6\text{-}37)$$

$$q_1 = \left(-1 - 2\frac{T_d}{T}\right) \cdot K_c = \left(-1 - 2 \cdot \frac{0.4148}{0.01}\right) \cdot 2.8887 = -242.5352 \quad (6\text{-}38)$$

$$q_2 = \left(\frac{T_d}{T}\right) \cdot K_c = \left(\frac{0.4148}{0.01}\right) \cdot 2.8887 = 119.8232 \quad (6\text{-}39)$$

Por último, se puede ya sustituir estos valores en la expresión (6-24):

$$U(k) = U(k-1) + 122.7293 \cdot E(k) - 242.5352 \cdot E(k-1) + 119.8232 \cdot E(k-2) \qquad (6\text{-}40)$$

Esta expresión es la que habrá que implementar programáticamente para aplicar el modelo discretizado de C(s).

Combinando las expresiones (6-15), (6-19) y (6-40) con el propio análisis de la imagen de manera adecuada en un script, se modela el sistema completo en bucle cerrado y se comprueba su correcto funcionamiento.

6.3 Generación de referencias y planificador para control de movimiento del UAV

Aunque se explicó al principio del apartado 6.2, se va a replantear una explicación más formal el cómo se generan las referencias para dicho controlador para facilitar así también la comprensión del planificador cuya explicación le seguirá posteriormente.

Se comienza definiendo una serie de sistemas de referencia y notación convenientes:

- $\{O\}$: Sistema de referencia asociado al punto objetivo. Este punto corresponde al centro representativo del cluster global (el que engloba a todos los objetivos avistados por la cámara gran angular). Puede definirse a partir de la posición del centro del cluster global y con la orientación inicial del vehículo (con el supuesto de que está inicialmente en hovering[8], de manera que los ejes XY de dicho sistema de referencia se encuentren sobre un plano horizontal). A pesar de que el vehículo se mueva o gire, este sistema de referencia permanecerá inmutable. Otra forma de definir su orientación es de acuerdo al criterio ENU (east-north-up, con el cual se fijarían direcciones geostacionarias para los ejes X, Y y Z, respectivamente).
- $\{B\}$: Sistema de referencia ligado al cuerpo del vehículo (*B* corresponde a body).
- $\{C_0\}$: Sistema de referencia ligado a la cámara central del UAV, la gran angular. Esta cámara es la responsable exclusiva del algoritmo de posicionamiento del vehículo.
- $\{C_0'\}$: Sistema de referencia ligado a la cámara central (gran angular), con el origen coincidente al de $\{C_0\}$, pero con una orientación fija y siempre apuntando hacia abajo, independientemente de los ángulos del gimbal o del vehículo. Definido de esta forma, la rotación entre $\{C_0'\}$ y $\{O\}$, dada por la matriz de rotación ${}^{C_0'}\boldsymbol{R}_O$,sería siempre constante (por ejemplo, un giro de 180º respecto al eje X). Teóricamente, si el gimbal de la cámara central funciona de forma óptima, el sistema de referencia de dicha cámara $\{C_0\}$ y $\{C_0'\}$ sólo deberían diferir, salvo transitorios breves, en el ángulo de yaw.

[8] Capacidad de un UAV (Vehículo Aéreo No Tripulado) de mantenerse en una posición fija en el aire sin desplazarse horizontalmente.

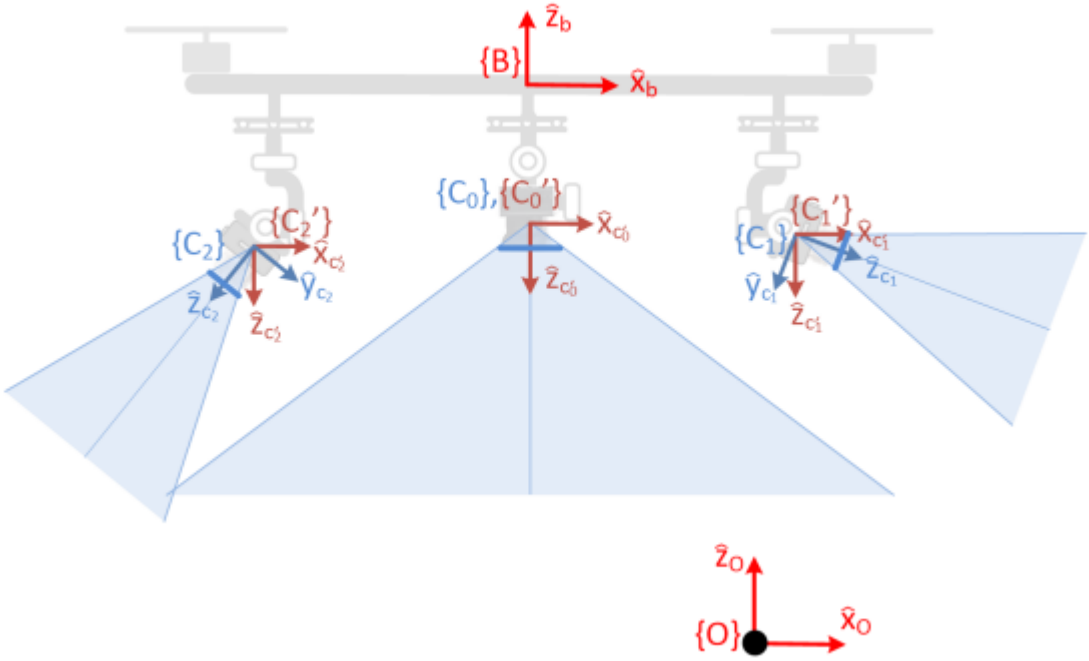


Figura 6-11. Distribución de los diferentes sistemas de referencias del UAV y cámaras, así como del centro del cluster global.

Considerando inicialmente que los targets u objetivos no se mueven, se tiene el caso de que $\{O\}$ es fijo. Los pasos a seguir serían entonces:

1. A partir del análisis de la imagen de la cámara gran angular $\{C_0\}$, se extrae la posición en la imagen del centro representativo del cluster global de los objetivos.
2. Se calcula la proyección inversa de dicho punto, conocida la altitud estimada de la cámara. Esto proporciona las coordenadas 3D estimadas del punto, respecto al sistema de referencia de la cámara gran angular: ${}^{C_0}\boldsymbol{p}_O$, posición del punto objetivo, coincidente con el origen de $\{O\}$, respecto a la cámara central.
3. Conocida la orientación inercial de $\{C_0\}$, se pueden convertir las coordenadas 3D del punto en coordenadas respecto a $\{C_0'\}$: ${}^{C_0'}\boldsymbol{p}_O$:

$$ {}^{C_0'}\boldsymbol{p}_O = {}^{C_0'}\boldsymbol{R}_O \cdot {}^{C_0}\boldsymbol{p}_O \qquad (6\text{-}41) $$

4. Como se quiere que el sistema de referencia para el guiado del vehículo sea $\{O\}$, se desea obtener a partir de lo anterior, la posición del vehículo respecto a $\{O\}$. Si se asume que la cámara central está justo en la vertical del punto origen de $\{B\}$, se puede definir de manera suficiente la posición del origen de $\{C_0'\}$ respecto a $\{O\}$:

$$ {}^{O}\boldsymbol{p}_{C_0'} = -{}^{C_0'}\boldsymbol{R}_O{}^{T} \cdot {}^{C_0'}\boldsymbol{p}_O \qquad (6\text{-}42) $$

5. Las coordenadas anteriores son las que deben hacerse variar, con el desplazamiento del vehículo, de forma que alcancen la referencia deseada, ${}^{O}\boldsymbol{p}_{C_0'}{}^{r}$. En principio, esta referencia se fijará siempre justo en la vertical del punto objetivo:

$$ {}^{O}\boldsymbol{p}_{C_0'}{}^{r} = \begin{bmatrix} 0 \\ 0 \\ h^r \end{bmatrix} \qquad (6\text{-}43) $$

Siendo h^r la altitud de referencia que se desea mantener en todo momento.

En el caso general en que los objetivos estén en movimiento, el origen de $\{O\}$ variará. Sin embargo, es de esperar que se pueda mantener ese mismo esquema, si se entiende que esa variación se puede ver como una perturbación que afecta al sistema controlado.

A la hora de aplicar el planificador de trayectorias, es sobre esta referencia sobre la que habría que aplicarlo.

6.3.1 Planificador de trayectorias

Se replantea visualmente la **Figura 6-12**, que ilustra la explicación previa y servirá como apoyo para explicar este planificador.

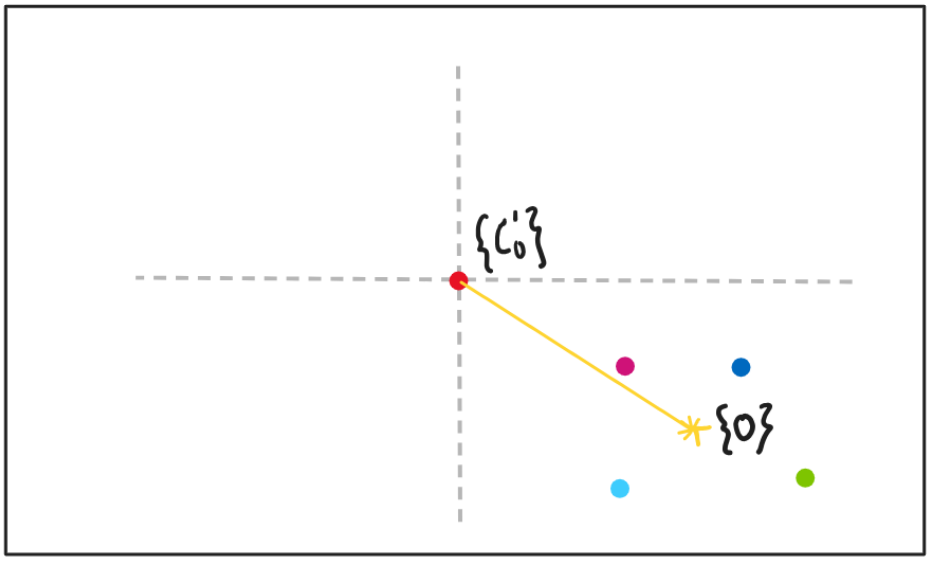


Figura 6-12. Centro óptico (punto rojo) respecto del punto representativo del cluster global (cruz amarilla).

Como se dijo al principio del apartado 6.2 y en la explicación anterior, el lazo de control equivalente de la **Figura 6-7** tomaría como referencia ${}^{O}\boldsymbol{p}_{C_0'}{}^{r} = \begin{bmatrix} 0 \\ 0 \\ h^r \end{bmatrix}$, y tras el análisis de la imagen y propio movimiento del UAV, se realimentaría con ${}^{O}\boldsymbol{p}_{C_0'} = -{}^{C_0'}\boldsymbol{R}_{O}{}^{T} \cdot {}^{C_0'}\boldsymbol{p}_{O}$. En la implementación, basta con introducir como error al controlador directamente las componentes X e Y de ${}^{O}\boldsymbol{p}_{C_0'}$, ya que Z es constante durante toda la simulación y no requiere de un lazo de control específico.

Es posible que se presenten varias circunstancias que provoquen que este error tome valores elevados, del orden de decenas de metros, lo cual provocaría movimientos rápidos del vehículo, haciendo saturar los actuadores hasta reducir suficientemente dicho error.

Principalmente, las dos circunstancias que llevarán a esta situación son las siguientes:

- **La altura del dron:** A cuanta mayor altura vuele el drone, los valores de error, es decir, ${}^{O}\boldsymbol{p}_{C_0'}$, serán mayores y, por tanto, mayores serán las actuaciones del vehículo.
- **La dispersión de los objetivos en la imagen:** Si el conjunto de objetivos se agrupa de tal manera que su centro representativo quede en uno de los extremos de la imagen de $\{C_0'\}$, nuevamente

los valores de error, es decir, ${}^{O}\boldsymbol{p}_{C_0'}$, serán mayores.

En ambos casos, se identifica que las componentes correspondientes a X e Y de ${}^{O}\boldsymbol{p}_{C_0'}$, toman valores de decenas de metros, y es cuando tengamos esta situación concreta cuando querremos aplicar el planificador para suavizar la referencia entregada.

El planificador elegido es del tipo polinómico, concretamente quíntico (de grado 5).

El algoritmo tiene una serie de entradas que definirán su funcionamiento:

- **t** → Instante de tiempo actual, determina en qué punto de la curva interpolada estamos.
- **pi** → Posición inicial.
- **pf** → Posición final. Es entre este par de valores donde se interpolan posiciones intermedias con trayectoria suavizada.
- **vi** → Velocidad inicial.
- **vf** → Velocidad final deseada al llegar a la posición final. Típicamente será nula.
- **ai** → Aceleración inicial.
- **af** → Aceleración final deseada al llegar a la posición final.
- **vMaxDes** → Velocidad máxima deseada. Influye en el cálculo del máximo necesario para realizar la trayectoria interpolada.
- **Tmin** → Tiempo mínimo en el que se desea realizar la trayectoria, influye también en la forma de la trayectoria interpolada.
- **distMin** → Valor umbral que indica que hay que recalcular la trayectoria.
- **primeraVez** → La primera vez que se llama al algoritmo, calcula la trayectoria completa con los parámetros previos y las consecuentes llamadas evaluará en qué parte de la trayectoria interpolada se encuentra según el tiempo transcurrido.

De manera resumida, se va a explicar cómo el algoritmo plantea la generación y evaluación de la trayectoria polinómica.

Se obtiene el número de grados de libertad de la trayectoria, en este caso, sería 2, las coordenadas en el eje X y en el eje Y.

Si es la primera vez o bien si se hace necesrio recalcular, se interpola completamente de nuevo la trayectoria, de lo contrario, se evalúa la trayectoria existente.

Se determina el tiempo máximo necesario para realizar la trayectoria, **Tmax**, para cada grado de libertad, y se guarda aquel valor que sea mayor. Es decir, puede ser que el desplazamiento en eje Y tarde más que en el eje X y sería el de Y el Tmax que prevalecería.

Con esto se calcula el tiempo final, **tf**, como **ti + Tmax**. Estos términos participan en el cálculo de los coeficientes del polinomio.

Ahora, para cada grado de libertad "**k**" (eje X y eje Y) Se calculan los coeficientes de la siguiente forma:

- Se obtiene un valor $\boldsymbol{dif} = \boldsymbol{pf}(\boldsymbol{k}) - \boldsymbol{pi}(\boldsymbol{k})$, que corresponde a la diferencia entre la posición final y la inicial.
- Se asignan velocidades y aceleraciones final nulas.
- Se calculan los coeficientes **a0, a1, a2, a3, a4** y **a5:**
  - $a0 = pi(k)$
  - $a1 = vi(k)$
  - $a2 = ai(k)/2$

- $a3 = \frac{(20*dif-(8*vf[k]+12*vi[k])*T_{max}-(3*ai[k]-af[k])*T_{max}^2)}{(2*T_{max}^3)}$
- $a4 = \frac{(30*(-dif)+(14*vf[k]+16*vi[k])*T_{max}+(3*ai[k]-2*af[k])*T_{max}^2)}{(2*T_{max}^4)}$
- $a5 = \frac{(12*dif-(6*vf[k]+6*vi[k])*T_{max}-(ai[k]-af[k])*T_{max}^2)}{(2*T_{max}^5)}$

Aunque para la tarea en cuestión sólo nos interesa las posiciones interpoladas, el planificador también genera un perfil polinómico de velocidades y aceleraciones. Para ello, se guardan como coeficientes los siguientes vectores:

- **Polinomio de posiciones:** $[a5, a4, a3, a2, a1, a0]$.
- **Polinomio de velocidades:** $[5 * a5, 4 * a4, 3 * a3, 2 * a2, a1]$.
- **Polinomio de aceleraciones**: $[20 * a5, 12 * a4, 6 * a3, 2 * a2]$.

Finalmente, en función del tiempo **t** que se recibe, se evalúan los polinomios para cada grado de libertad **"k"** obteniéndose la posición, velocidad y aceleración correspondientes para el instante en cuestión.

Para evaluar dicho polinomio, se puede hacer de manera sencilla haciendo uso de una función del paquete **numpy**, concretamente **polyval().**

Esta función recibe el polinomio y un segundo parámetro que es el valor con el que evaluarlo, en este caso el instante de tiempo en el que se quiere evaluar el valor de dicho polinomio.

Para tener un ejemplo visual de cómo resulta una trayectoria interpolada entre dos posiciones se acude a la siguiente **Figura 6-13.**

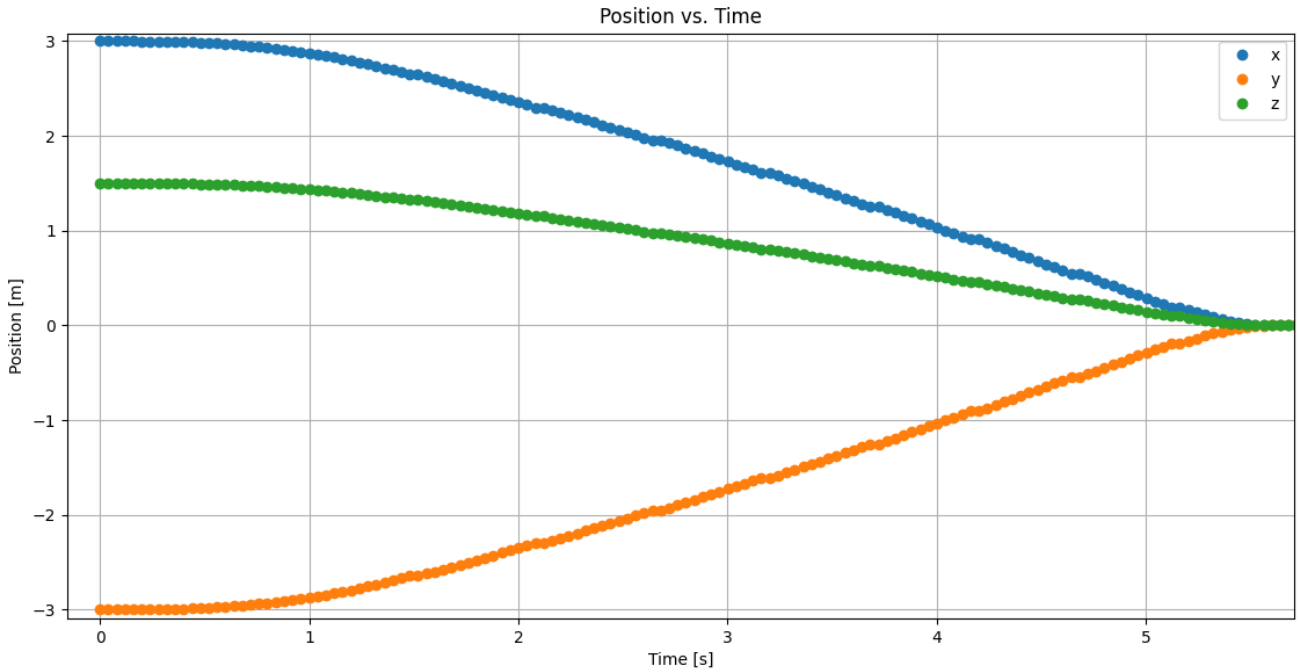


Figura 6-13. Ejemplo de trayectoria planificada para las posiciones de 3 ejes.

En la figura anterior, para el eje X se ha interpolado entre la posición inicial x = 3 y la final x = 0, para el eje Y entre la posición inicial y = -3 y la final y = 0, y finalmente, para el eje Z entre la posición inicial z = 1.5 y la

final z = 0.

Se observa como para los 3 ejes, al ser configurados con los mismos parámetros, se planifica una trayectoria suave a ser realizada a lo largo de 5 segundos.

Para el caso particular en el que nos encontremos con que la norma de ${}^{O}\boldsymbol{p}_{C_0'}$ sea superior a un umbral, que puede ser, por ejemplo, 5m, se hace actuar al planificador para la entrega de referencias en lugar de entregarlas directamente de la manera explicada al inicio del apartado 6.3.

Básicamente, la idea es que tras analizar la imagen y obtener ${}^{O}\boldsymbol{p}_{C_0'}$, se interpole entre dicha posición y ${}^{O}\boldsymbol{p}_{C_0'}{}^{r}$. Con esto, en lugar de introducir directamente ${}^{O}\boldsymbol{p}_{C_0'}$ como el error del controlador, se entregarán los puntos interpolados evaluados en cada instante, evitando así entregar un error tan grande y suavizando por tanto la respuesta del controlador.

Sin embargo, hay que tener en cuenta cómo el movimiento del UAV acorde a esa entrada interpolada hace variar ${}^{O}\boldsymbol{p}_{C_0'}$, es decir, una vez se vuelva a recibir la información analizada de la imagen que actualiza ${}^{O}\boldsymbol{p}_{C_0'}$, dado que estamos hablando de errores y distancias de gran magnitud, habrá que replanificar de nuevo para generar puntos intermedios acorde a la posición real del punto representativo del cluster respecto de la cámara gran angular.

Para comprender mejor cómo el planificador está actuando en el sistema, se recomienda acudir a la **Figura 6-14**, que aparece a continuación.

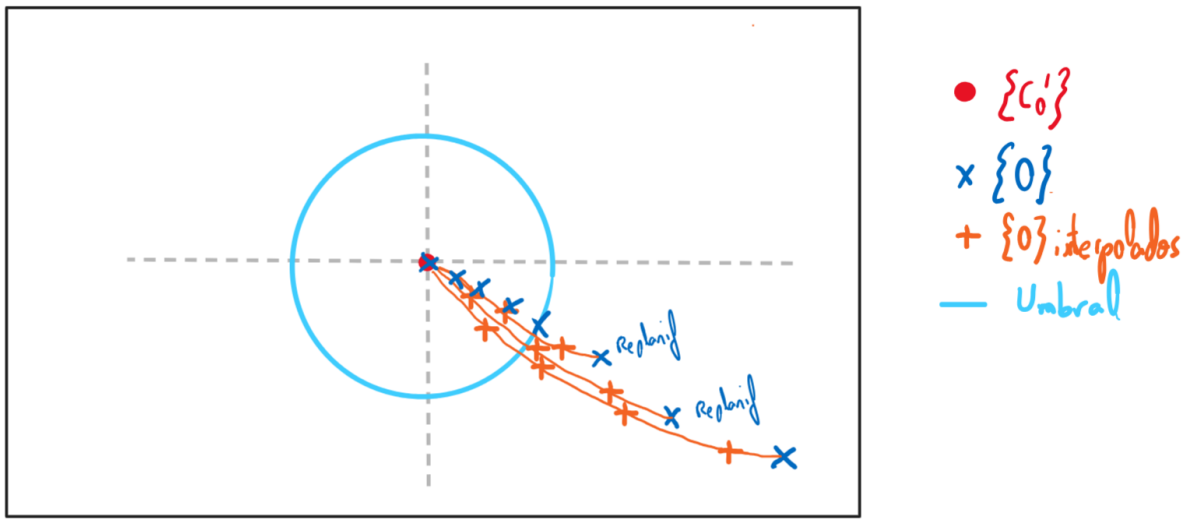


Figura 6-14. Planteamiento del planificador para la generación de referencias del control del UAV.

Para mayor claridad, se han omitido en este esquema los diferentes objetivos que conformarían el centro representativo $\{O\}$.

En esta figura, se plantea la idea aproximada de cómo se está aplicando el planificador en el sistema implementado para el control de posicionamiento del vehículo.

Como se dijo al final de la explicación previa, cuando los objetivos están en movimiento, el origen de $\{O\}$ irá variando, como se observa en la propia figura (cruces azul oscuro).

Podemos encontrarnos dos situaciones, y en cada una se actuará de manera diferente:

1. Que $\{O\}$ respecto de $\{C_0'\}$, es decir, ${}^{C_0'}\boldsymbol{p}_O$, esté a una distancia tal que se encuentre fuera de esa circunferencia que delimita el umbral para usar o no el planificador. En este caso, se genera una

trayectoria completa hasta ${}^{O}\boldsymbol{p}_{C_0'}{}^{r}$ y serán entregados en secuencia como referencias los $\{O\}$ interpolados (cruces de color naranja que definen la trayectoria interpolada).

Cuando estamos en este caso, la trayectoria interpolada no se realiza y se entrega a ciegas sin realimentación de los movimientos del UAV y por tanto, de $\{O\}$ en la imagen.

Es esperable que la nueva posición de $\{O\}$ recibida tras un nuevo análisis de la imagen no coincida con el $\{O\}$ interpolado que estaba siendo entregado previamente. Por esto, se vuelve a replanificar otra trayectoria que una el nuevo $\{O\}$ real con ${}^{O}\boldsymbol{p}_{C_0'}{}^{r}$, y se repite el proceso de entrega de referencias los nuevos $\{O\}$ interpolados.

Esto seguirá ocurriendo y haciendo actuar el planificador hasta que el control de posicionamiento reduzca el error, o posición ${}^{C_0'}\boldsymbol{p}_{O}$, por debajo del umbral, de manera que sería en este momento cuando pasamos a la situación 2.

2. Como también se aprecia en la **Figura 6-14**, una vez $\{O\}$ queda situado dentro de la circunferencia umbral, se deja de usar el planificador. El control recibirá directamente la posición ${}^{C_0'}\boldsymbol{p}_{O}$, obtenida cada vez que se analice la imagen, ya que, dentro de este rango, el error ya es suficientemente reducido. Por tanto, aplicando directamente esas referencias, las actuaciones serán moderadas y adecuadas para terminar de centrar al UAV respecto del punto representativo del cluster global de targets.

7 IMPLEMENTACIÓN

Está claro que el estudio teórico es una parte fundamental del proyecto, de ahí que la extensión de capítulos anteriores son la base teórica del proyecto. Sin embargo, la finalidad fundamental del proyecto es, basándose en la implementación previa de [10], plantear una versión alternativa del sistema en un simulador software diferente a ROS, que es el simulador por excelencia en robótica.

La idea es programar las funcionalidades y algoritmos de los capítulos anteriores e implementarlos en el software CoppeliaSIM, que es una alternativa menos conocida para simular algoritmos relacionados también con robótica. En cuanto al trasladar los algoritmos desarrollados a un sistema real, es menos inmediato que ROS, pero también tiene plugins relacionados con ROS y en cierto modo también habilitaría incorporar lo desarrollado en un sistema real.

De esta manera, se va a comentar en este apartado cómo se han resuelto diferentes problemas de modelado y los resultados obtenidos en simulación para verificar la validez y la estrategia necesaria para simular el sistema descrito.

Podemos ver en la **Figura 7-1** cómo resulta modelado el sistema UAV con múltiples cámaras:

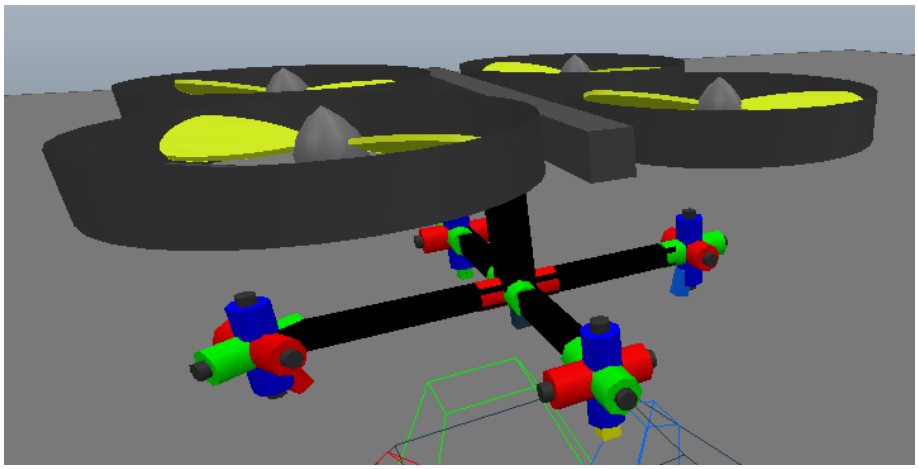


Figura 7-1. Modelo del sistema con 4 cámaras orientables junto a la gran angular.

A diferencia del framework *Gazebo + ROS*, que combina dos de las herramientas más usadas en robótica actualmente, CoppeliaSIM ya ofrece la posibilidad de simular los elementos y su comportamiento en el mismo programa a través del uso de clientes que controlan el comportamiento del conjunto de los elementos de la escena, como se explicaba en el Apartado 2 del proyecto.

Aunque sea distinto a ROS, en el sentido de que se puede programar funcionalidad asociada a los objetos de la escena desde el propio CoppeliaSIM en lenguaje Lua, esto tiene ciertas limitaciones en cuanto a que sólo se puede tener un script por objeto de la escena y además las comunicaciones entre ellos son complejas.

De ahí a que, aunque no sea el esquema de programación típico de CoppeliaSIM, para el problema planteado en este proyecto donde cooperan multitud de subsistemas y se comunican entre ellos, se hace imprescindible el esquema paralelizado de ROS, basado en nodos y publishers/subscribers. Para ello, se utiliza la librería ZeroMQ como se mencionaba en el Apartado 2, que ofrece similares funciones a estas herramientas nativas de ROS.

7.1 Introducción al esquema software implementado

Aunque la idea original de la implementación es que sea parametrizable en cuanto a que funcione con cualquier número de cámaras, este software no es tan flexible en ese aspecto y es que se necesita que los elementos ya estén instanciados en la escena, es decir, no se pueden crear objetos desde un cliente externo u otro tipo de fichero. Con esto en cuenta, se han desarrollado las funcionalidades descritas para el caso de contar con 2 ó 4 cámaras orientables, lo cual ya es una prueba suficiente de funcionamiento.

Para otro número de cámaras podría implementarse realizando ciertas modificaciones en la escena y en los diferentes códigos.

Esto limita un poco la flexibilidad de la implementación de CoppeliaSIM respecto a la implementación en Gazebo + ROS que sí que permite flexibilizar esta generación de las cámaras en el sistema y probar diferentes configuraciones.

La implementación se basa en un cierto número de nodos para cada una de las diferentes tareas que son ejecutados como scripts de Python en un terminal de Ubuntu y cada uno de ellos se conecta con la simulación de CoppeliaSIM, la cual es iniciada desde uno de los scripts en concreto, que podría definirse como el "master".

Esto se logra mediante un script de tipo "bash" que se ejecuta desde un propio terminal dándole como comandos el "número de cámaras" que puede ser 2 ó 4 y el "modo de funcionamiento" que puede ser "individual" ó "clustering":

Figura 7-2. Esquema de nodos cliente implementados según número de cámaras orientables.

Además de este "launch.sh", hay otro llamado "launchTrayIndiv.sh", cuya única diferencia es que no incluye el script encargado de generar targets y gestionar su movimiento (movObjetivos.py), ya que, en algunas escenas, el movimiento de los targets se programa directamente en CoppeliaSim. Esto se verá más adelante.

Es importante mencionar, que además de estos scripts, hay otros que no se ejecutan, pero contienen funciones y algoritmos desarrollados para implementar diferentes tareas y son importados en la mayoría de los que se ejecutan en la **Figura 7-2** y que serán mencionados a continuación.

Para comprender mejor las tareas de estos scripts cliente, así como los que incluyen diversas funcionalidades, conviene explicar brevemente la utilidad de cada uno de ellos y qué problemas resuelve:

Scripts que no se "ejecutan" como clientes, pero sí son llamados e importados desde los scripts cliente:

- **GeometryUtils.py:** Incorpora funciones de segmentación, obtención de bounding boxes a partir de una imagen, cálculo de distancias, obtención de matrices de transformación, cálculo de proyección inversa, función para obtener los ángulos a girar para el apuntamiento aproximado, cálculo del centroide del convex hull, etc.

- **CamerasParams.py:** Define los valores de resolución, tamaño del sensor, distancia focal máxima y mínima tanto de la cámara gran angular como las cámaras orientables.
- **CoppeliaComms_ZMQ.py:** Incluye funciones para inicializar las cámaras con los parámetros adecuados y obtener sus handlers para trabajar con ellas desde los clientes, así como funciones para trabajar con la distancia focal y obtener imágenes a partir de las diferentes cámaras.
- **GestionObjetivosFn.py:** Incluye funciones para calcular el convex hull según los objetivos que aparecen en la imagen y para mostrarlo en la propia imagen. Otras tareas que resuelve son obtener el punto central del convex hull y el cálculo del radio de la circunferencia que circunscribe al convex hull. También alguna función para identificar los objetivos en la imagen, para pintar los bounding boxes y para encontrar al objetivo más cercano al centro de la imagen.
- **trackingCamGA.py:** Incluye funciones para obtener las coordenadas 3D del centro del cluster global de objetivos para el control del UAV, así como una para realizar la asignación de objetivos a cada cámara en función de la verticalidad. También incluye una para inicializar y actualizar los trackers o multi-trackers de la cámara gran angular.

Scripts que sí se "ejecutan" como clientes e interactúan con la escena de CoppeliaSIM:

- **master_tracking.py:** actúa como script "master", es decir, es el que inicia remotamente la simulación de CoppeliaSIM, inicializa las cámaras y en un bucle infinito realiza las siguientes tareas:
    - Crear/resetear el multi-tracker, asignar los objetivos avistados en la cámara gran angular a las diferentes cámaras orientables
    - Mientras no haya errores en el seguimiento en la cámara gran angular:
        - Comprobar si hay que reiniciar y realizar de nuevo el apuntamiento aproximado (etapa 1).
        - Actualizar trackers y obtener ángulos de giro para cada cámara para ese apuntamiento aproximado.
        - Durante 50 frames, aplicar esos giros en los gimbals del apuntamiento previo aproximado para centrar aproximadamente en las diferentes imágenes los objetivos asignados.
        - Una vez han pasado esos frames, avisar a los nodos correspondientes de que se conmuta al control de regulación en los gimbals.
        - Si hay error en los trackers de la cámara gran angular, reiniciar la asignación de objetivos y repetir desde el principio.
        - Mostrar las imágenes con la información adecuada (bounding boxes, identidad de cámaras, etc.).
- **TrackingCamOrientables.py:** Se define una función encargada de encontrar al objetivo asignado en la cámara orientable y calcular las referencias de giros necesarios de los gimbals para realizar el apuntamiento fino. En un bucle infinito se realizan las siguientes tareas:
    - El programa "escucha" la notificación de conmutación a control de regulación en cada iteración.
    - Si estamos aún en etapa 1 (apuntamiento aproximado), este nodo esperará bloqueado hasta que se le notifique dicha conmutación.
    - Una vez le llega, se aplica el algoritmo explicado para hallar el zoom a aplicar en cada cámara en modo individual y se termina publicando las referencias de zoom adecuadas.
    - Se calculan los giros necesarios para el apuntamiento fino de cada cámara y se publican como referencia para sus respectivos nodos que tendrán sus subscribers configurados para recibirlas.

  - Se muestran las imágenes de cada cámara durante el control de regulación.
  - Si hubiese un reinicio y se tuviese que volver a aplicar la etapa 1 (apuntamiento aproximado), el nodo se volverá a bloquear esperando que se le vuelva a notificar la conmutación al control de regulación.

- **ControladorEulerII.py:** Se define una función que implementa un interpolador polinómico para generar referencias para el control de posición del UAV en caso de que el error supere cierto valor umbral. Se implementan tres funciones para las ecuaciones (6-15), (6-19) y (6-40) y en un bucle infinito se realiza lo siguiente:
  - Actualizar (si hay información entrante) las referencias de coordenadas 3D del centro representativo del cluster global de objetivos.
  - En función de si el error en coordenadas 3D (tras realizar proyección inversa), definido como la distancia euclídea entre (X, Y) del centro representativo del cluster global con el centro de la imagen (0, 0) supera cierto valor, 4m en este caso, se toman referencias para el control de diferente manera:
    - Si supera el umbral, el planificador genera referencias interpoladas implicando así un movimiento más suave del UAV.
    - Si no se supera el umbral, se entrega directamente como referencia la información recibida anteriormente.
  - Se aplican las ecuaciones en diferencias que modelan G(s) (con el integrador que incluye desacoplado) y C(s) discretizadas y el integrador de manera independiente.
  - Se satura la velocidad del UAV a 15 m/s en caso de que la salida del sistema implique una velocidad superior en su movimiento.
  - Se actualizan términos
  - Se actualiza la posición del UAV en CoppeliaSIM de manera remota.

- **master_control_GimbalRed.py / master_control_GimbalBlue.py / master_control_GimbalGreen.py / master_control_GimbalYellow.py:** Se tienen 4 scripts similares para cada uno de los gimbals, que se ejecutarán o no en función de si se configura el sistema con la opción de 2 o 4 cámaras orientables. En cualquier caso, su función principal es recibir las referencias de orientación para cada grado de libertad del gimbal y modificarla en consecuencia a sus dinámicas. Para llevar esto a cabo, se realiza en estos programas lo siguiente:
  - Inicializar variables que intervienen en el control del gimbal.
  - Cargar del directorio que los contiene los diferentes modelos, control en posición angular absoluta (etapa 1) y control de regulación (etapa 2), para cada grado de libertad del gimbal y guardarlas en variables.
  - Se entra en un bucle infinito en el que cíclicamente se realiza lo siguiente:
    - De manera no bloqueante, se comprueba si ha llegado la notificación de cambio de modo de control, es decir, si hay que conmutar de etapa 1 (apuntamiento aproximado) a etapa 2 (apuntamiento más preciso con realimentación de la propia imagen).
    - En caso de que sea la primera iteración, o bien se haya transicionado de la etapa 1 a la 2 o viceversa, se inicializan nulas las variables de estado de cada articulación del gimbal.
    - Para el resto de las iteraciones donde el funcionamiento es el habitual:
      - Si el **modo de control es el de la etapa 1**, se recogen las referencias de orientación calculadas en otros scripts y se pasan a formato matricial.
      - Se realiza el cálculo de las variables de estado y la propia salida de

orientación de cada articulación del gimbal siguiendo la explicación del apartado 6.1.1.

- Se modifica la orientación de cada eje del gimbal acorde a la salida del control basado en espacio de estados.
- Se actualizan valores de las variables de estado, y se recogen en variables el último valor de cada salida de cada eje del gimbal. Estos serían usados como valor inicial de la orientación del control de regulación en caso de que conmute el tipo de control. Esto es fundamental, ya que, para funcionar correctamente, el control de la etapa 2 debe tomar como condiciones iniciales de orientación la nueva orientación modificada por el control de la etapa 1.
- Tras cierto tiempo realizando el apuntamiento aproximado en la etapa 1, **se conmuta a la etapa 2.** Aquí nuevamente se reciben referencias de orientación para cada articulación del gimbal, pero en este caso calculadas con el procedimiento del apartado 4.2.2.
- En este caso sólo se actuarán sobre los grados de libertad correspondientes al eje X e Y del gimbal.
- Nuevamente, se calculan variables de estado y salidas de orientación de cada articulación del gimbal siguiendo ahora la explicación del apartado 6.2.2.
- Se actualizan valores de las variables de estado.
- Se modifica la orientación de cada eje del gimbal acorde a la salida del control de regulación planteado en espacio de estados.
- Se actualiza la variable que gestiona si ha habido un cambio de etapa.

- **Referencias_Ctrl_UAV.py:** La función principal de este script es obtener la posición representativa del cluster global de targets basándose en la imagen de la cámara gran angular, de acuerdo a lo explicado en el apartado 6.3.

  Para ello, el código realiza lo siguiente:

  - Configura publisher para enviar al nodo de control de posicionamiento del UAV la nueva posición de dicho centro ($\{O\}$).
  - A partir de la imagen de la cámara gran angular y las técnicas de procesamiento previamente mencionadas, se calcula el centro del cluster global de objetivos en coordenadas métricas 3D.
  - Se publica en el topic adecuada esta posición.
  - Muestra visualmente cómo el dron está centrándose respecto de los múltiples objetivos, adoptando una posición más ventajosa para realizar el seguimiento.

- **movObjetivos.py:** El objetivo de este script es generar trayectorias elípticas para diferentes agrupaciones en función del número de cámaras orientables del sistema.

  El cómo se generan las elipses será explicado más adelante de manera más detallada, luego la explicación de este script será mas concisa. La idea es:

  - Se tiene una función que genera centros, radios, sentido, orientación y excentridad de las elipses que constituirán las trayectorias de los objetivos.
  - Se tiene otra función que en función de los parámetros aleatoriamente generados para generar el conjunto de los puntos que describen cada una de las elipses.
  - Se llama a las funciones, generando así las diferentes elipses.
  - En un bucle infinito:

        - Se itera para cada uno de los objetivos realizando las siguientes tareas:
            - Se construye el "handler" del objetivo correspondiente al que tiene dicho objetivo en la escena de CoppeliaSim.
            - Se calcula la posición actual del objetivo en función de la trayectoria elíptica aleatoria que le corresponda.
            - Se modifica la posición del objetivo en la escena.
            - Si se ha terminado de recorrer la elipse, se continúa por el comienzo de la misma, constituyendo un movimiento sin fin de los objetivos.

- **master_zoom_red.py / master_zoom_blue.py / master_zoom_green.py / master_zoom_yellow.py:** Se tienen 4 scripts similares para cada una de las cámaras, que se ejecutarán o no en función de si se configura el sistema con la opción de 2 o 4 cámaras orientables. En cualquier caso, su función principal es recibir las referencias de zoom para cada cámara orientable y modificarlo según lo explicado en el Apartado 5.4. Para llevar esto a cabo, se realiza en estos programas lo siguiente:
    - Se inicializan valores que intervienen en el modelado de la dinámica del zoom.
    - Se precalculan constantes del modelo del zoom.
    - En un bucle infinito:
        - Se espera de manera no bloqueante una nueva referencia de zoom.
        - Si llega, se recoge y se actualiza.
        - Si no llega, se trabaja con la referencia que se tenga y se aplica a la dinámica discretizada del zoom del apartado 5.4.
        - Se modifica el zoom en CoppeliaSim de manera remota en función de la salida de la dinámica anterior.

Tras este análisis de todos los diferentes códigos que actúan conjuntamente, se entiende mejor el alcance y propósito de la implementación del proyecto. Este conjunto de programas permite simular todo el desarrollo teórico presentado en los capítulos anteriores.

Se puede así verificar los algoritmos desarrollados en un entorno de simulación novedoso y que, con cierto esfuerzo y salvando las diferencias con un sistema real, podría extrapolarse y llegar a aplicarse en un UAV real que disponga de la distribución de cámaras propuesta, con una algoritmia similar.

Como se ha dejado entrever en las explicaciones de los diferentes códigos, se puede configurar el sistema haciendo que funcione con 2 ó 4 cámaras orientables (la gran angular es necesaria siempre) y dos modos de funcionamiento:

- Seguimiento de objetivos individuales (uno para cada cámara orientable).
- Seguimiento de agrupaciones de múltiples objetivos.

Esto se configura desde los lanzadores de la simulación, indicando en uno de los argumentos "clustering" o "individual".

7.2 Modelado de los diferentes elementos simulados en CoppeliaSim

Este apartado es fundamental, ya que ante el desconocimiento inicial de CoppeliaSim, modelar adecuadamente los diferentes elementos y subsistemas era una tarea compleja. Se va a recoger en los siguientes apartados cómo se han ido modelando cada uno de ellos.

7.2.1 Mundo

En primer lugar, es importante mostrar el mundo que se genera por defecto al ejecutar CoppeliaSim. Aunque se trata de un mundo vacío, el suelo que ofrece por defecto no sería adecuado para la naturaleza del sistema que queremos implementar.

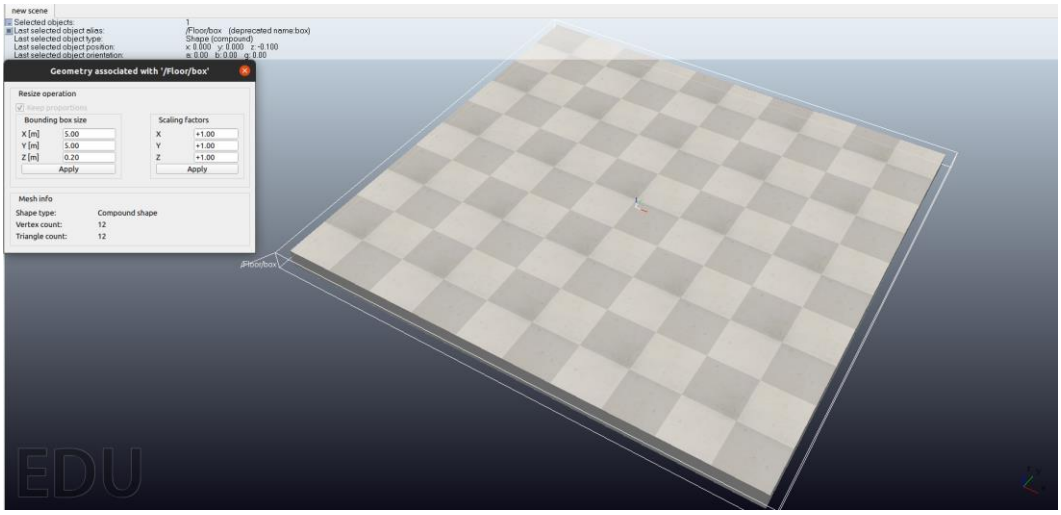


Figura 7-3. Mundo generado por defecto al ejecutar CoppeliaSim inicialmente.

Los principales inconvenientes de este mundo son en primer lugar su tamaño, consistente en un suelo en forma de cuadrado de 5x5m. Considerando que el UAV está a su altura típica de 25m, faltaría espacio para que los objetivos se desplazasen en un tamaño tan reducido. Esto sin contar que la dispersión de los objetivos podría ser aún mayor y el dron podría trabajar a mucha mayor altura.

Por otro lado, el patrón de tablero de ajedrez que presenta el suelo sería problemático para los algoritmos de segmentación básicos planteados en el Apartado 4.1.1.

De esta forma, se hace necesario incrementar las dimensiones del suelo en gran medida, así como modificar el color y apariencia del suelo.

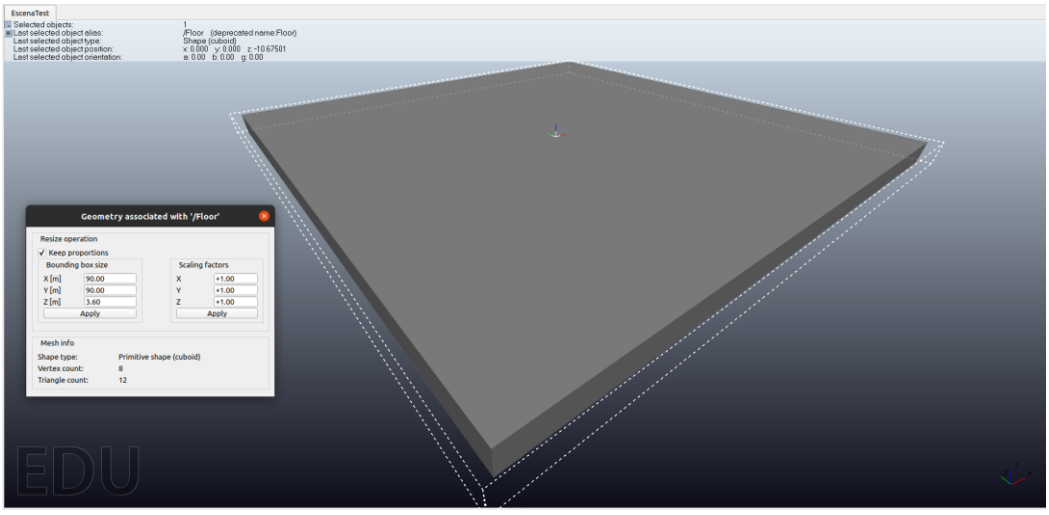


Figura 7-4. Mundo vacío modificado para adaptarse al problema del proyecto.

Vemos ahora en la **Figura 7-4** como las dimensiones del suelo pasan a ser de 90x90m, lo cual es más que suficiente para el UAV trabajando a 25m de altura. En caso de que trabajase por ejemplo a una altura de 100m por ejemplo, está claro que habría que duplicar o triplicar el tamaño del suelo nuevamente para que los objetivos tengan espacio para desplazarse.

Además del tamaño del suelo, su apariencia ahora es homógenea y de color gris, que hará contraste con los objetivos o targets que luego veremos y permitirá su segmentación respecto de dicho fondo en la propia imagen de las cámaras, facilitando su identificación.

Este modelado del mundo es perfecto para el algoritmo de detección del Apartado 4.1.1, sin embargo, en el caso de haber escenarios más complejos y heterogéneos, pero que aún así tengan un fondo predominante y uniforme, los objetivos seguirían siendo identificables. En este caso, podría ocurrir que sólo con este algoritmo se detecten como objetivos entidades que no se desean, luego habría que refinar el algoritmo o combinarlo con otras técnicas de procesamiento adicionales para diferenciar qué entidades son de interés y cuáles no.

Además, se busca un fondo sencillo también por razones de rendimiento, ya que cuantos más elementos haya presentes en la escena, y más si se desplazan en ella, la simulación se enlentece.

Por la complejidad del sistema y el conjunto de códigos actuando en paralelo, el rendimiento ya es bastante limitado, luego complicarlo aún más haría casi imposible que los algoritmos visuales tengan una continuidad en cuanto a FPS para funcionar de manera adecuada.

7.2.2 UAV

El siguiente elemento del sistema es el UAV, al que se ha añadido una estructura para alojar las diferentes cámaras que posee, en lugar de dejarlas sin soporte, ya que de lo contrario, caerían al suelo al comenzar la propia simulación.

Aunque estructuralmente la diferencia entre un UAV con 2 ó 4 cámaras orientables es notoria, para agilizar los experimentos y evitar redundancias creando simulaciones, en todas las escenas se usa el modelo con 4 cámaras orientables, y la idea es que, si indica al lanzar la simulación que sólo dispone de 2 de ellas, sólo 2 de las 4 serán funcionales durante esa simulación.

Antes de describirlo, conviene visualizarlo en la **Figura 7-5.**

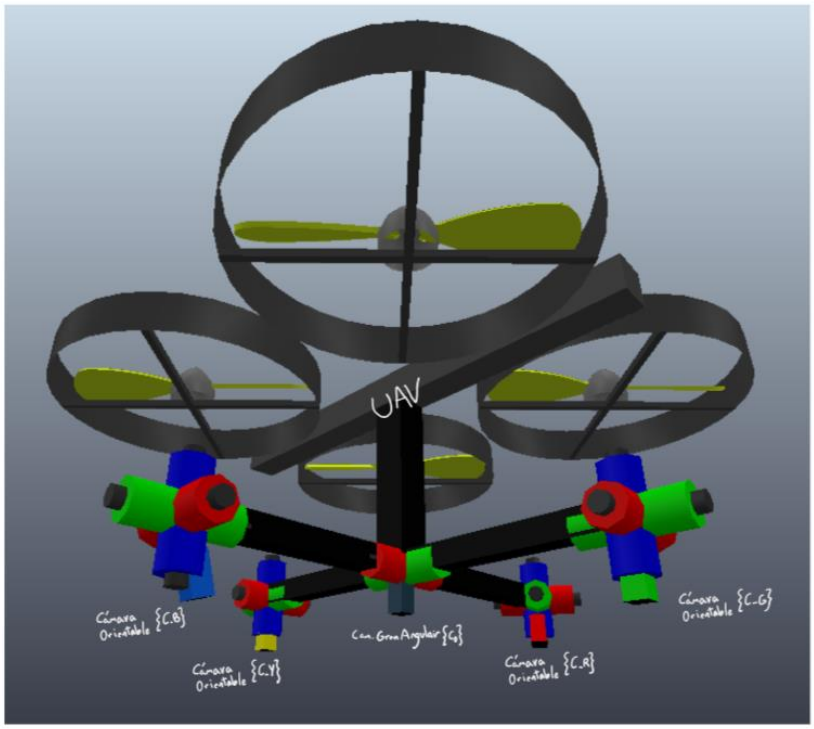


Figura 7-5. Modelo del UAV en CoppeliaSim con 4 cámaras orientables.

El modelo del UAV está disponible en la colección de modelos de CoppeliaSim, como se veía en el Apartado 2.2.1 y es el que se utiliza aquí. El resto de la estructura que sporta las cámaras, los gimbals de cada cámara y las propias cámaras se han agregado a dicho modelo del UAV a posteriori para conformar el sistema completo.

Es destacable que la cámara gran angular se dispone justo en la vertical del cuerpo (body) del UAV, y que la disposición de las cámaras orientables es simétrica en cuanto a la longitud a la que se encuentran de la gran angular y de cada una de las otras cámaras orientables.

Otro aspecto importante es que el propio modelo del quadrotor incorpora un algoritmo de control, que aplica inclinaciones del mismo a la hora de ir a la posición objetivo. En un principio, parecía una alternativa interesante, sin embargo, al inclinarse el UAV, por mucho que se intentase contrarrestar ese efecto con los propios gimbals, todas las cámaras se desestabilizaban durante el movimiento del UAV y daban imágenes movidas que no eran coherentes para aplicar algoritmos de visión.

Sería interesante llegar a poder simularlo con ese movimiento más realista, pero por las limitaciones de CoppeliaSim y los gimbals desarrollados era impracticable.

La alternativa adoptada, por tanto, es asumir que los gimbals serían capaces de cancelar el efecto de alabeo y cabeceo del UAV durante su movimiento, de tal manera que en nuestra simulación las inclinaciones del UAV serán nulas, aunque en el sistema real estarían presente y su magnitud dependería de la velocidad con la que se mueva el UAV.

Concretando entonces, el UAV se podría mover en los ejes X, Y, Z realizando traslaciones puras, sin inclinarse, aunque realmente la altura en el eje Z no variará por la forma de operar. La estrategia de control que realiza esto es la explicada en los Apartados 6.2 y 6.3.

Se podría plantear algoritmos de optimización que trabajen conjuntamente sobre la distancia focal de las cámaras y la altura del dron, para evitar llevar al máximo el zoom óptico, entrando en juego el ascenso o descenso del dron para favorecer el avistamiendo de objetivos.

Finalmente, queda mencionar respecto de la orientación de las cámaras del UAV, la cámara gran angular deberá mantener una orientación constante todo el tiempo de manera que apunte siempre hacia el suelo, para así avistar al máximo número de objetivos posible. Por su parte, el resto de las cámaras deberán ser orientables siguiendo el funcionamiento de giro de un gimbal de 3 grados de libertad.

7.2.3 Cámara gran angular y cámaras con zoom óptico

Cada una de las cámaras que se incluyen se obtienen del modelo de "sensor visual" disponible en CoppeliaSim, con la versión que implementa el modelo de cámara basado en proyección perspectiva, que es el que se corresponde a las explicaciones del Apartado 4.2. Para facilitar la explicación de las mismas, también conviene acudir a una captura que muestra el modelo físico de la cámara y sus parámetros configurables.

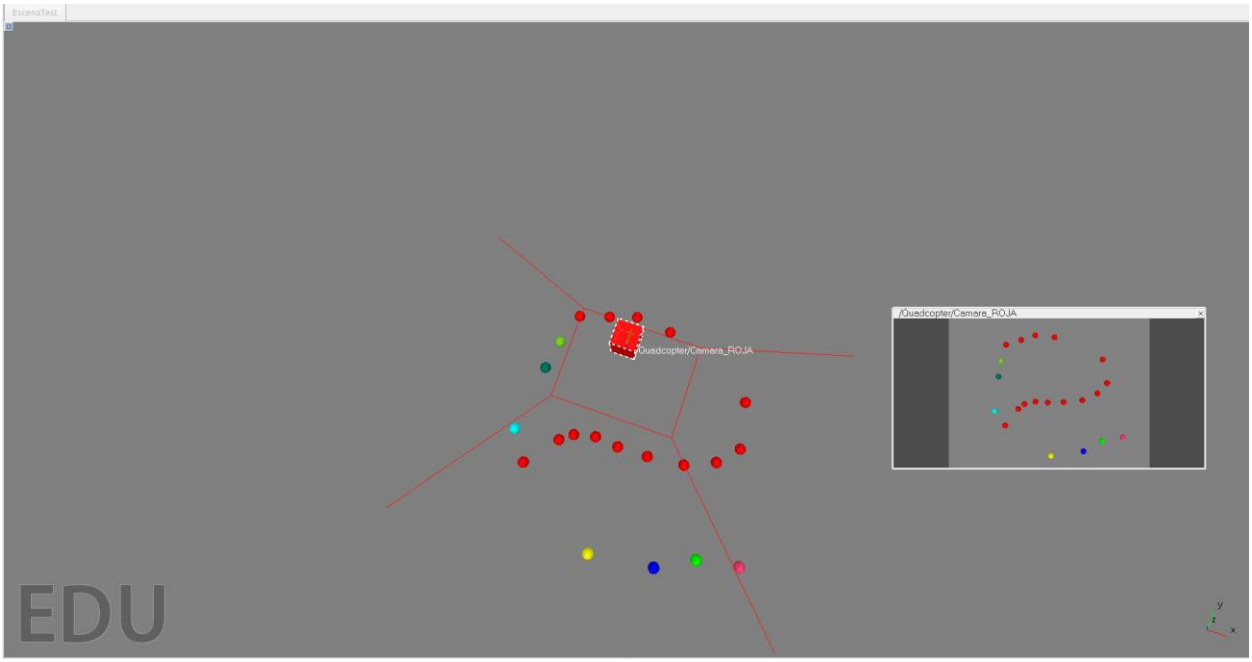


Figura 7-6. Ejemplo de funcionamiento de cámara en CoppeliaSim.

Es importante notar que la configuración de esta cámara corresponde a la de las cámaras orientables.

Para explicar la configuración de las cámaras orientables y gran angular se trata de modelar las siguientes cámaras comerciales reales:

- Raspberry Pi Camera HQ Lente 2.8 – 12mm
  - wSensor (ancho): 6.287; hSensor (alto): 4.712 [mm].
  - Resolución:4056 x 3040 píxeles.
  - fmin (distancia focal mínima): 2.8; fmax (distancia focal máxima): 12 [mm**].**
- Raspberry Pi Camera HQ Lente 6 – 22mm
  - wSensor (ancho): 6.287; hSensor (alto): 4.712 [mm].
  - Resolución:4056 x 3040 píxeles.
  - fmin (distancia focal mínima): 6; fmax (distancia focal máxima): 22 [mm**].**

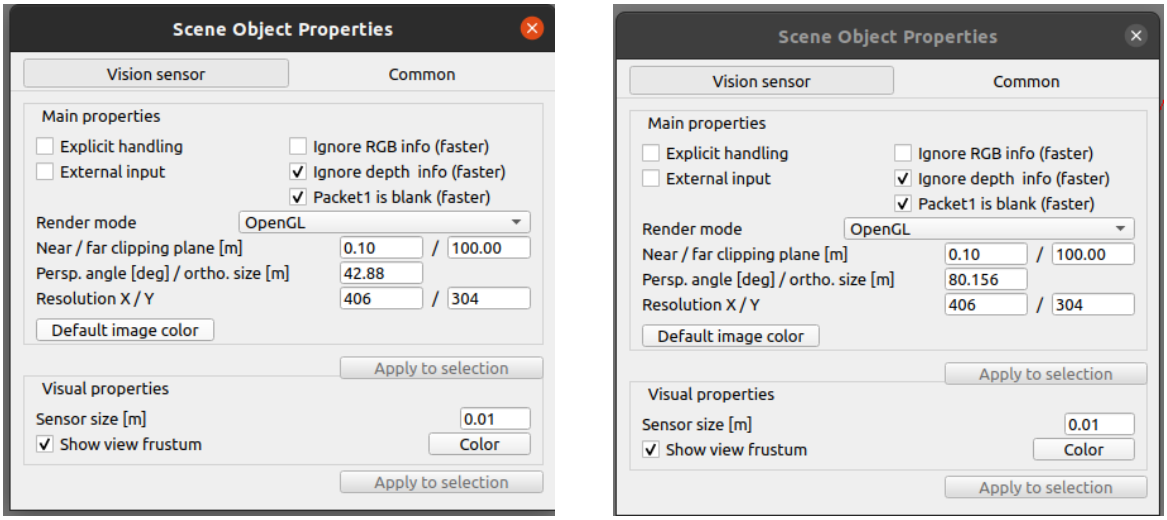
La primera cámara corresponde a la cámara gran angular y es la que tiene menor valor de distancia focal, que será el que se use para que el zoom sea el mínimo posible, con el objetivo de tener la mejor capacidad de avistamiento de objetivos al cubrir un campo visual mayor.

Por su parte, la segunda cámara, corresponde a las que se pretenden usar como cámaras orientables. La diferencia es que estas sí disponen de una mayor distancia focal máxima, permitiendo aplicar zoom desde grandes distancias para obtener una imagen más cercana y detallada del objetivo.

Para ambas cámaras aunque la resolución original es de 4056 x 3040 píxeles, en la implementación se ha modelado un 10% de esa resolución, quedando 406 x 304 píxeles, lo cual es equivalente a realizar un reescalado sobre la imagen original de la cámara.

El motivo de esto es que generar imágenes de alta resolución, para múltiples cámaras, es extremadamente costoso para la simulación, teniendo sin esta reducción un rendimiento de la simulación que impide realizar ninguna tarea por la lentitud de recepción de imágenes. Para ello, se ha modelado en el script que configura las cámaras esta resolución. Para cambiarlo, bastaría con hacerlo en dicho programa, aunque nuevamente, cuanto mayor sea esta resolución peor funcionará la simulación.

Veamos en la siguiente **Figura 7-7** cómo se configuran las dos cámaras en CoppeliaSim.



a) b)

Figura 7-7. a) Configuración de modelo de cámara orientable. b) Configuración de modelo de cámara gran angular.

Vemos como la primera opción configurable es **"Near / far clipping plane [m]",** que consiste en el rango de distancias en el que la cámara "genera" imagen. Es decir, para esta configuración la cámara captaría imágenes desde 10 cm hasta 100m.

La tercera opción se trata de la resolución en ejes X e Y, que toma los valores ya mencionados con la reducción aplicada para mejorar el rendimiento de la simulación.

Finalmente, la opción del medio **"Perspective angle [deg]"** Corresponde al ángulo de apertura de la cámara, que está directamente relacionado con la distancia focal.

A la hora de modificar el zoom de las cámaras orientables de acuerdo a las referencias que generen los algoritmos correspondientes, habrá que realizar una conversión de distancia focal en mm calculada por dichos algoritmos al valor en grados correspondiente para modificarlo programáticamente en ese valor de *Perspective angle.*

A la hora de encontrar la relación entre distancia focal y *perspective angle*, hay que prestar atención a qué motor de renderizado está usando el modelo de la cámara, siendo que en este caso se aprecia que es **OpenGL**.

Sabiendo esto y realizando cierta investigación sobre cómo modela la cámara OpenGL, se puede acudir al siguiente enlace:

https://www.gamedev.net/forums/topic/566855-simulating-cameras-focal-length-in-opengl/

Donde en las propias respuestas se aprecia cómo plantear esta conversión y que se va a recoger aquí para mayor claridad.

OpenGL usa entre otras (a un nivel más bajo que la configuración accesible de CoppeliaSim), la siguiente función para parametrizar una cámara virtual:

```
gluPerspective(  fovy,     aspect,    near,    far  );
```

Aquí, a partir de la configuración de CoppeliaSim definimos cada uno de los parámetros de la función. Sin embargo, es notable que lo que CoppeliaSim parametriza como *Perspective Angle*, constituye realmente el FOVy, es decir, el Field Of View en el eje Y. Por tanto, si para una cámara queremos variar su f (distancia focal) entre el valor mínimo, fmin, y el valor máximo, fmax, habrá que obtener el FOVy correspondiente a dicha distancia focal.

Para poder realizar esta conversión, basta con acudir al siguiente enlace:

https://en.wikipedia.org/wiki/Angle_of_view#Calculating_a_camera.27s_angle_of_view

y extraer que la expresión que relaciona ambos parámetros es, que por aclarar se deja la siguiente figura, en ella se extrae el FOV_x, pero la expresión es idéntica salvo que se usa wSensor (ancho) en lugar de hSensor (altura).

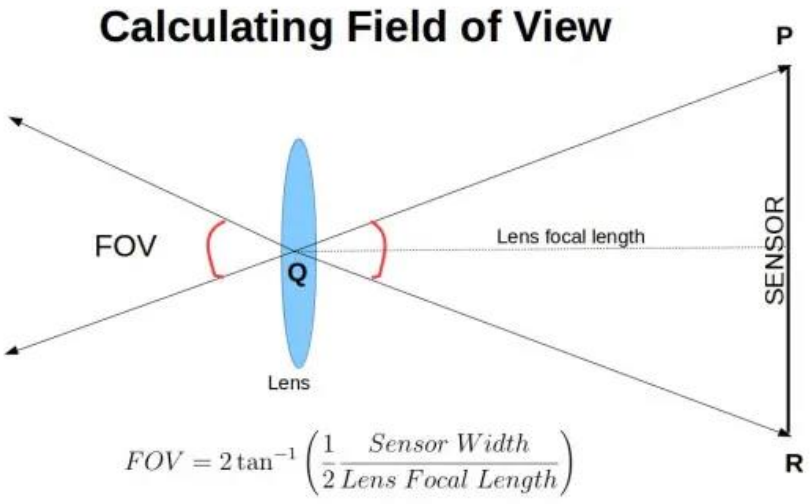


Figura 7-8. Esquema ilustrativo de la obtención del FOV a partir de la distancia focal [35].

$$PerspectiveAngle = FOV_y[rad] = 2 * \arctan\left(\frac{d}{2f}\right)[rad] \quad (7\text{-}1)$$

Donde d corresponde a la altura en mm del sensor (film), es decir, hSensor = 4.712mm. y f la distancia focal.

Como el parámetro en CoppeliaSim se define en grados, hay que convertir el cálculo anterior en radianes a grados:

$$FOV_y[deg] = \left(\frac{180}{\pi}\right) * 2 * \arctan\left(\frac{d}{2f}\right)[deg] \quad (7\text{-}2)$$

Concretando todo esto, se puede tanto iniciar la distancia focal al mínimo al principio de la simulación, como cuando sea necesario aplicar zoom en las cámaras orientables, ejecutar una función que permite modificar el parámetro *Perspective Angle* en función de la distancia focal necesaria según las ecuaciones (7-1) y (7-2).

7.2.4 Gimbals

A la hora de modelar los gimbals, hay que tener claro cómo se va a expresar la orientación del mismo. Al haber diferentes representaciones de la orientación la implementación puede variar ligeramente en función de si se utilizan *quaternions*, ángulos de Euler u otra representación.

En este caso, se decide trabajar con la orientación de los gimbals mediante los ángulos de Euler ZY'X'', comúnmente denominados como *yaw, pitch* y *roll.* Al poder utilizar esta implementación directamente en CoppeliaSim, no hay que realizar complejas conversiones matemáticas para pasar de una representación matemática a otra.

Una de las dificultades que presenta la implementación de esta representación de ejes móviles, es que la rotación alrededor de un eje debe "hacer rotar" al resto de los ejes, para que así la siguiente rotación en el siguiente eje, sea con la nueva orientación resultado del primer giro. Esto se conoce como concatenación de rotaciones. Para comprender mejor como funciona estas rotaciones en ejes móviles, se proporciona la **Figura 7-9.**

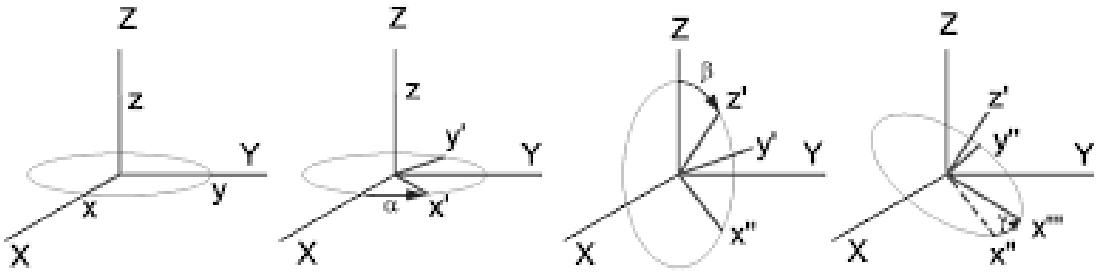


Figura 7-9. Concatenación de rotaciones sobre ejes móviles, Ángulos de Euler.

Luego es importante que la implementación realizada cumpla este esquema de ejes móviles. Para ello, se usa una combinación del elemento **"articulación de revolución"**, de manera que cada una de ellas constituya uno de los grados de libertad del gimbal. Tras ello, aprovechando la estructura jerárquica de los objetos padre-hijo, hay que establecer una estructura en la que el grado de libertad del eje Z sea el padre original, el grado de libertad del eje Y sea hijo del eje Z y este a su vez sea padre del eje X, una especie de escalera conformando ese orden de prioridad de giros ZY'X''. Con esto se consigue que los elementos hijos vean afectadas su posición y orientación en función de los movimientos o giros del padre.

Para comprender mejor esta estructura jerárquica, se presenta en la **Figura 7-10** cómo resulta el gimbal modelado en Coppelia y las relaciones de prioridad entre cada grado de libertad.

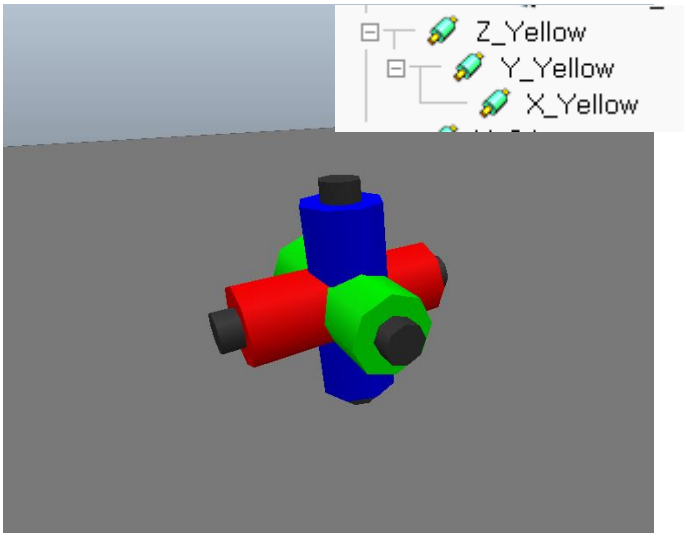


Figura 7-10. Modelado de Gimbal de 3 grados de libertad y relaciones jerárquicas entre ellos.

En esta figura, la articulación azul correspondería al eje Z, la verde al eje Y y la roja al eje X, que coincide con la codificación de colores para los sistemas de referencias típica en simuladores de robótica.

Para comprobar que el Gimbal realiza adecuadamente la composición de rotaciones, se va a plantear unas capturas de un experimento de ejemplo.

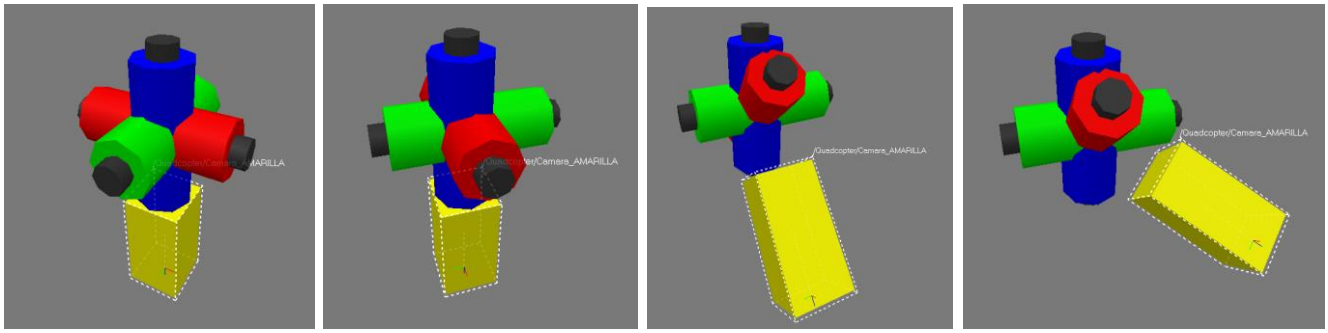


Figura 7-11. Rotaciones compuestas sobre ejes móviles. De izquierda a derecha: Posición original, Giro 45º eje Z, Giro 45º eje Y, Giro 45º eje X.

Se puede observar como la cámara asociada va modificando su orientación en función de cómo van girando los diferentes grados de libertad del gimbal, esto es importante para que, cuando se calculen los ángulos a girar como en el Apartado 4.2.1 y 4.2.2 el gimbal gire de acuerdo a este procedimiento y la cámara apunte correctamente al objetivo adecuado.

7.2.5 Objetivos ó targets

El modelado de los objetivos es un apartado importante, ya que son elementos fundamentales para verificar el correcto funcionamiento del sistema.

Es importante destacar que, aunque en el Apartado 4.1 se planteaban objetivos constituidos por cuboides, para la implementación final se ha optado por definir los objetivos como esferas. El motivo de este cambio es que los cuboides son problemáticos a la hora de desplazarlos programáticamente, ya que, si rozan con el suelo con alguna de sus caras, generan colisiones y rebotan de manera irregular. En cambio, con las esferas se evita este problema y su movimiento es natural y fluido.

A pesar de esta ligera modificación en el modelado de los targets, todo lo explicado en los apartados del capítulo 4 con cuboides es totalmente equivalente para las nuevas esferas.

Con esto, cada escena para los experimentos contendrá un cierto numero de esferas que se desplazarán, con diferentes colores para facilitar el seguimiento por parte del algoritmo de tracking KCF.

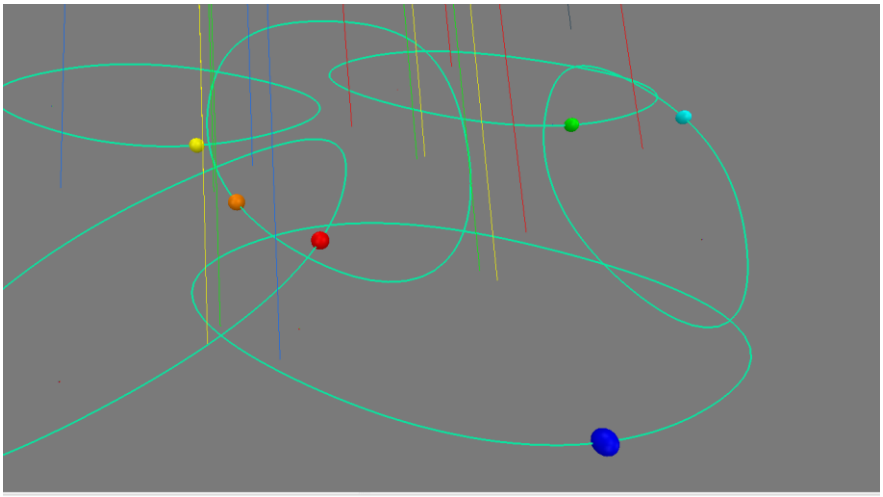


Figura 7-12. Modelado de los objetivos en el mundo como esferas de diferentes colores.

7.3 Movimiento de los diferentes elementos en CoppeliaSim

Esencialmente, el movimiento de todos los elementos se resuelve de manera remota desde los scripts cliente. Sin embargo, en algunas escenas, las trayectorias de los objetivos están directamente implementadas en CoppeliaSim para mayor control sobre qué tipo de trayectorias se desean y generar situaciones de interés para los algoritmos.

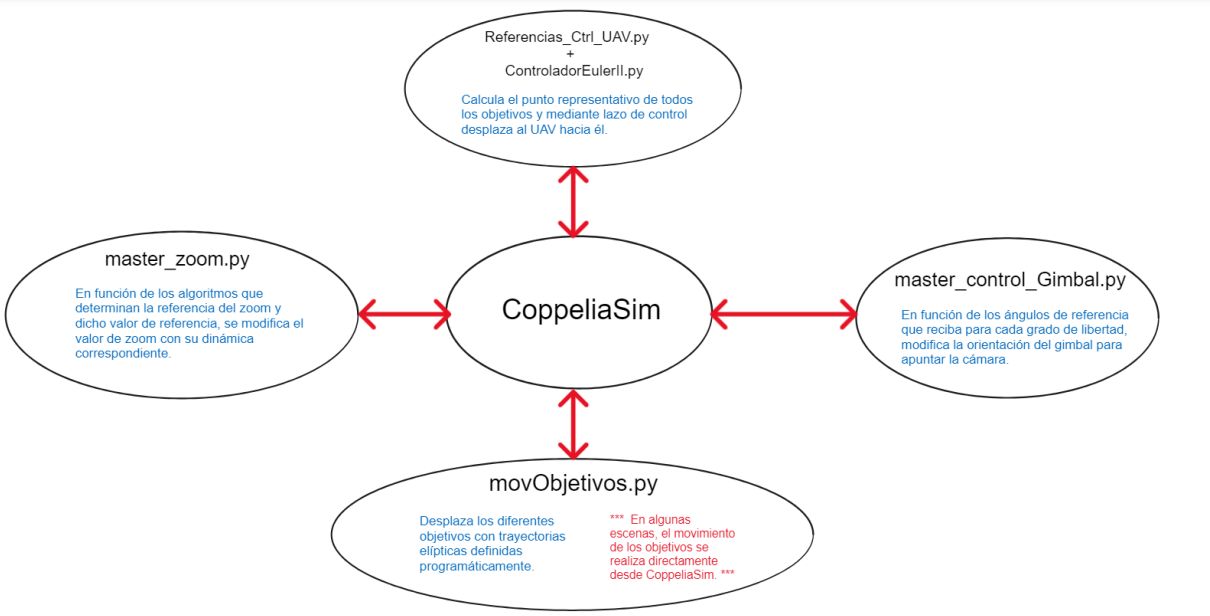


Figura 7-13. Esquema de los nodos que actúan sobre los elementos de la simulación.

Se resume en este esquema las comunicaciones remotas que se realizan con la simulación de CoppeliaSim y brevemente qué realiza cada programa.

7.3.1 Trayectorias de los objetivos

Al igual que se aprecia en el esquema anterior, hay dos alternativas para generar las trayectorias de los objetivos.

Vamos a empezar describiendo la forma basada en el código *movObjetivos.py*. La trayectoria de cada objetivo se representará respecto al sistema de referencia del mundo (*world*), de esta manera, la idea es trasladar los objetivos a través de los diferentes puntos de paso que discretizarán la trayectoria elíptica generada, sin modificar la orientación de las esferas (aunque no tendría efecto si se hiciese más que añadir carga computacional).

La decisión de trabajar con trayectorias elípticas se debe a que al tener multitud de parámetros que las definen, caracterizar aleatoriamente cada uno de ellos, permite obtener diferentes movimientos en cada simulación que se ejecuta.

Se quiere distinguir dos posibles escenarios, uno en el que los objetivos se desplacen de forma aleatoria e inconexa durante toda la simulación (estas trayectorias se han modelado manualmente en CoppeliaSim para tener más control sobre lo que ocurre, se verá más adelante) y otra situación en la que se generen agrupaciones de objetivos que "orbiten" de manera cercana a los objetivos de su agrupación.

En las siguientes **Figura 7-14** y **Figura 7-15**, se observan trayectorias para el segundo caso comentado en que se buscan trayectorias que conformen agrupaciones de objetivos.

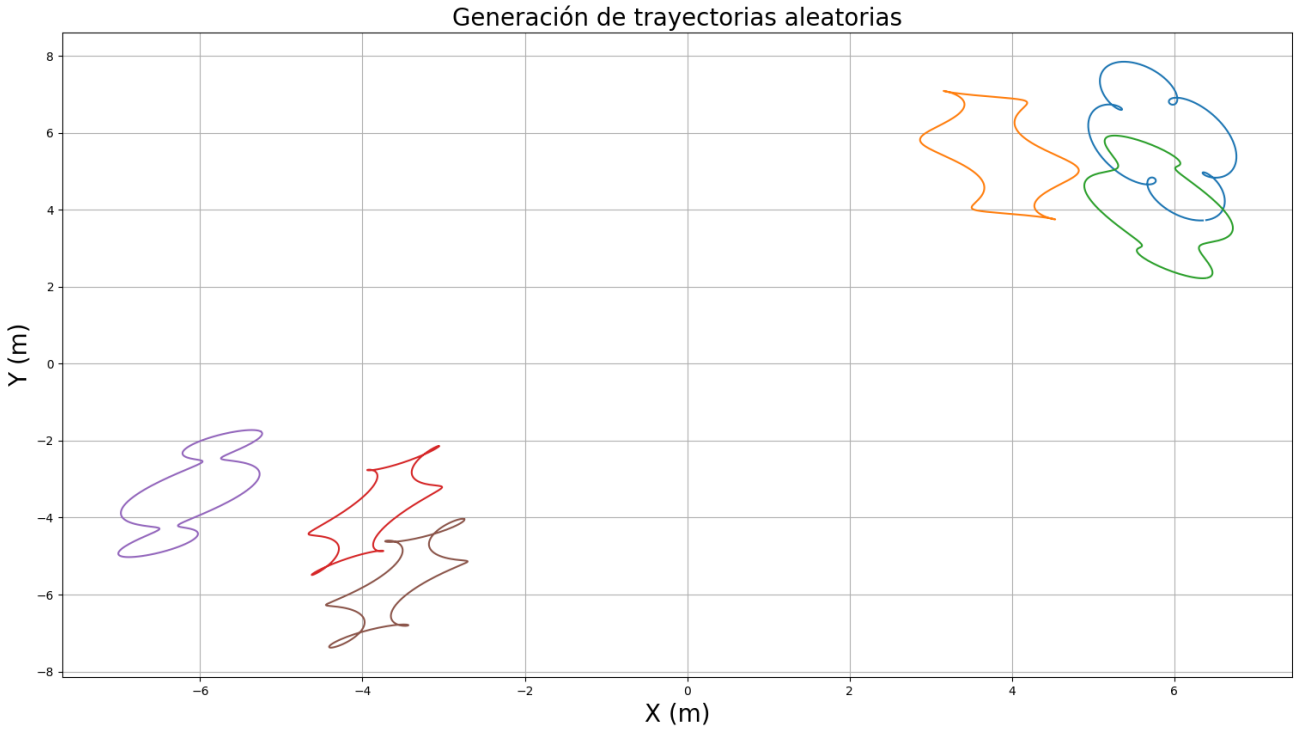


Figura 7-14. Ejemplo de trayectorias para 2 agrupaciones de 3 objetivos cada una.

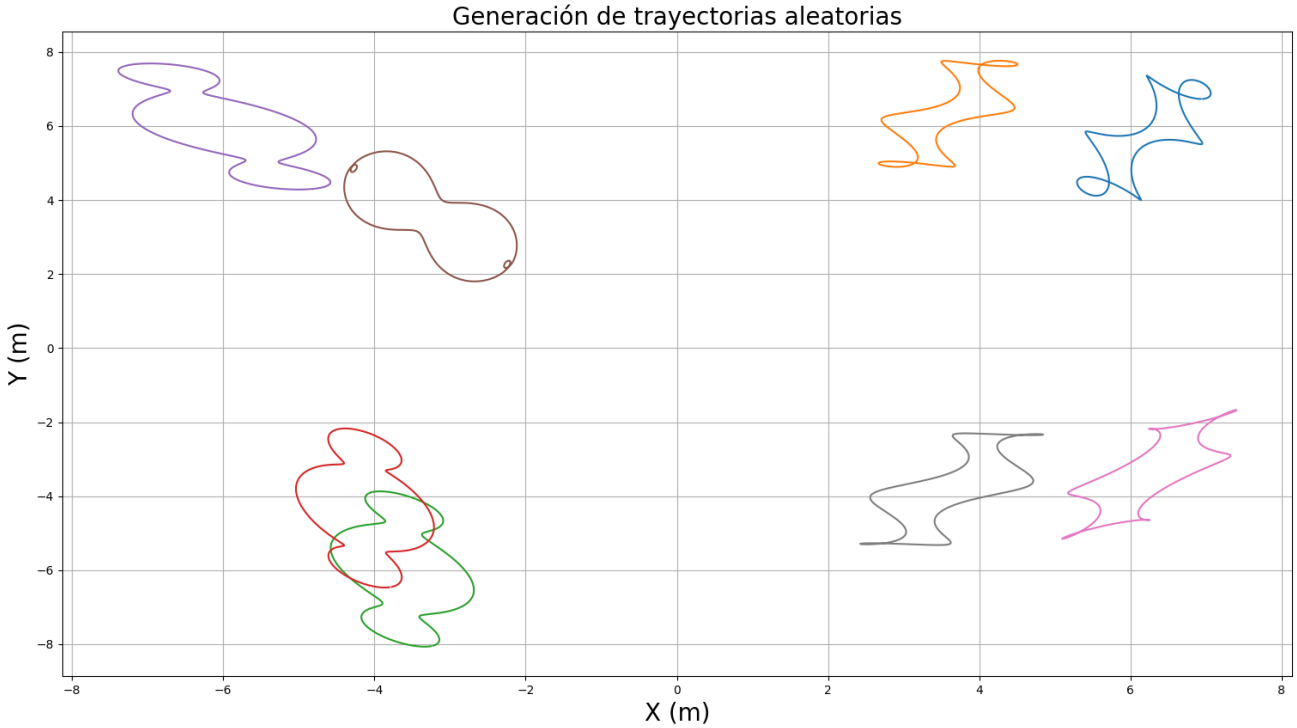


Figura 7-15. Ejemplo de trayectorias para 4 agrupaciones de 2 objetivos cada una.

En este tipo de trayectorias, cada objetivo tiene asociada dos elipses, la elipse que define su cluster, que es compartida por todos los objetivos que pertenecen a su agrupación y su propia elipse particular que define su movimiento individual completo.

Por tanto, para generar trayectorias con agrupaciones de objetivos, la elipse del cluster recogerá las coordenadas de los centros de las elipses individuales de los objetivos. De esta forma, se logra que los objetivos en se conjunto se muevan conforme a una elipse mayor, pero que cada uno de manera individual oscile entorno a ella, con una elipse de menor tamaño, como se veía en las dos figuras anteriores.

Para programarlo, se ha desarrollado una clase: *generador_trayectorias,* que recoge los métodos para generar los parámetros que definen las trayectorias y permiten indexarlas de forma discreta a través de ciertos puntos de paso, con una resolución fija.

De esta manera, los parámetros que definirán una elipse serán los siguientes:

- Coordenadas de su centro (x, y).
- Radio (r).
- Excentricidad (e).
- Ángulo de orientación (θ).
- Sentido (s).

Estos valores serán escogidos aleatoriamente para cada una de las elipses a generar, dentro de unos rangos de valores adecuados. Una vez obtenidos, los puntos de las elipses quedarán definidos de la siguiente forma:

$$ {}^{w}P_{obj} = \begin{bmatrix} x \\ y \end{bmatrix} + r \cdot \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} e & 0 \\ 0 & e \end{bmatrix} \cdot \begin{bmatrix} cos(s \cdot \alpha) \\ \sin(s \cdot \alpha) \end{bmatrix} \quad (7\text{-}3) $$

En dicha ecuación, el sentido (s) valdrá 1 ó -1, y α irá entre 0 y 2π. Con ello, tendremos la definición de la elipse y podremos obtener puntos de ella a partir de valores discretos de α.

Para este caso en que se están generando trayectorias de las agrupaciones, basta con concatenar la ecuación de la elipse superior con la de la elipse individual, correspondiéndo el punto obtenido con la primera ecuación a las coordenadas (x, y) del centro de la segunda elipse.

Para este caso, en la ecuación de la segunda elipse, hay que multiplicar α por un escalar entero mayor a 1, para lograr que la elipse pequeña del objetivo individual realice vueltas más rápidamente que la elipse del cluster, generando así ese efecto de órbita y oscilaciones en torno a ella.

Con una frecuencia fija, se incrementará el valor de α, se calculará para dicho valor la posición actual de cada uno de los objetivos, y se trasladará a CoppeliaSim, ejecutando efectivamente el movimiento de los mismos.

Aunque esta codificación también habilita la generación de trayectorias inconexas que se entrelazan, como se dijo previamente, se ha optado por generar estas trayectorias de manera manual para forzar situaciones en las que haya oclusión parcial o cruces entre objetivos y poner a prueba el sistema.

En este caso, la programación de las trayectorias está directamente realizada dentro del simulador.

Para ello, podemos tomar como ejemplo la siguiente figura, que es la escena que se ha realizado con trayectorias elípticas inconexas y que se entrelazan para cada objetivo individual.

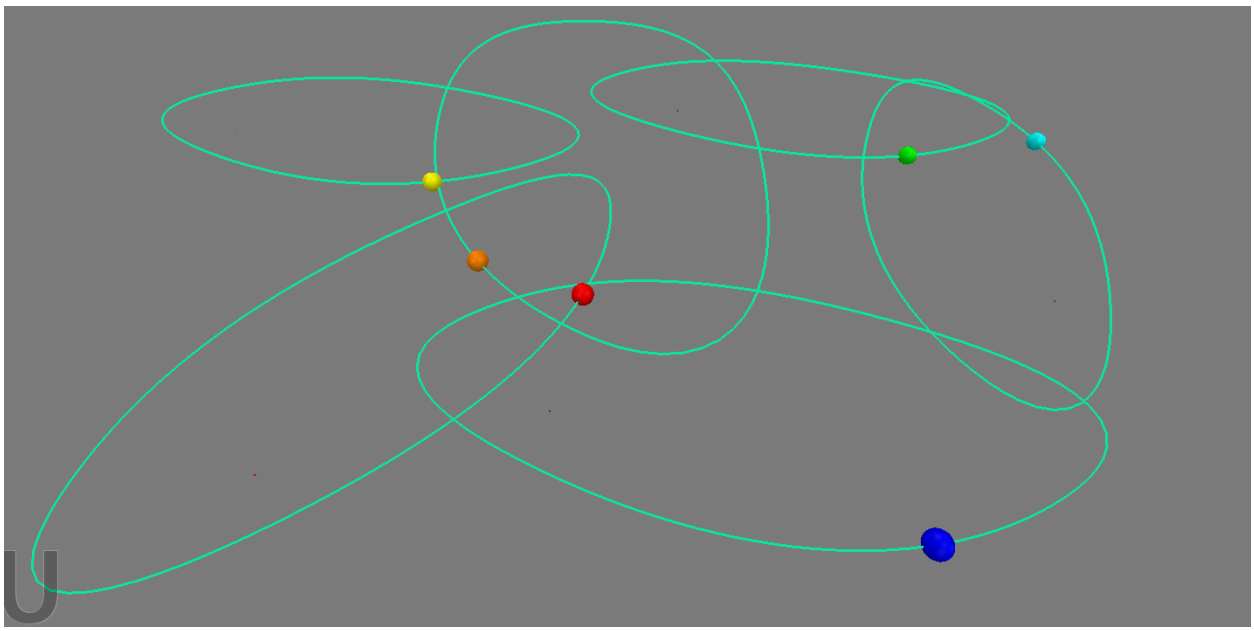


Figura 7-16. Ejemplo de trayectorias descritas manualmente para 6 objetivos.

En este caso, para crearlas, hay que usar el Menú "Add", seleccionar "Path" y para una elipse la opción "Closed". Con esto se genera una trayectoria con ciertos puntos de control que permitirán modificarla libremente. Los puntos de control se pueden desplazar para cambiar la forma de la trayectoria a cualquier geometría imaginable. Una vez terminada la trayectoria, esta se puede trasladar en el mundo donde se quiera para que quede en la posición que más convenga, como por ejemplo en la **Figura 7-16**, donde intencionalmente se buscaban cruces entre trayectorias.

Para comprender cómo los puntos de control modifican la forma de la trayectoria, se presenta en la **Figura 7-17** una comparativa con una de las trayectorias elípticas y cómo resulta tras modificar la posición de dichos puntos de control.

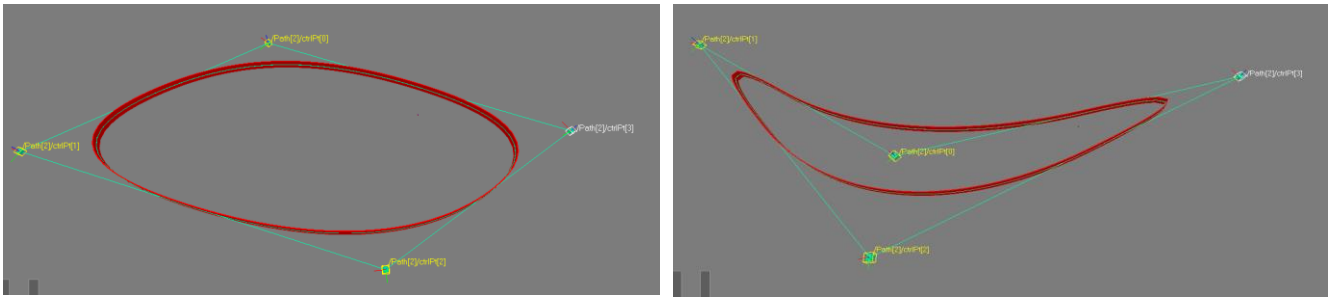


Figura 7-17. Ejemplo de modificación de trayectoria en CoppeliaSim mediante puntos de control.

Existe la posibilidad de añadir un número indefinido de puntos de control, lo cual daría infinitas posibilidades a la hora de definir diferentes geometrías para las trayectorias.

Una vez comentado cómo generar la trayectoria de esta forma, queda ver cómo es el código que hace que cierto elemento de la simulación recorra esta trayectoria.

Esto se hace asociando a la esfera que hace de objetivo un *child script* que le otorgará el comportamiento deseado de que se desplace a lo largo de la trayectoria.

Código 7.1 Código asociado al objetivo en CoppeliaSim para que siga una trayectoria.

```
function sysCall_init()
    -- Inicializacion de parametros y handlers (solo se ejecuta la primera
vez)

    cube = sim.getObject('/Sphere[0]')
    path = sim.getObject('/Path[0]')
    pathData = sim.unpackDoubleTable(sim.readCustomDataBlock(path, 'PATH'))
    local m = Matrix(#pathData//7,7,pathData)
    pathPositions = m:slice(1,1,m:rows(),3):data()
    pathQuaternions = m:slice(1,4,m:rows(),7):data()
    pathLengths, totalLength = sim.getPathLengths(pathPositions,3)

    velocity = 0.25 --m/s
    posAlongPath = 0
    previousSimulationTime = 0

end

function sysCall_actuation()
    -- Actuacion: Calcular posicion y orientacion del objetivo a lo largo
del path
    -- y actualizar el tiempo anterior

    local t = sim.getSimulationTime()
    posAlongPath = posAlongPath + velocity * (t - previousSimulationTime)
    posAlongPath = posAlongPath % totalLength
    local pos = sim.getPathInterpolatedConfig(pathPositions, pathLengths,
posAlongPath)
    local quat = sim.getPathInterpolatedConfig(pathQuaternions,
pathLengths, posAlongPath, nil, {2,2,2,2})
    sim.setObjectPosition(cube, path, pos)
    sim.setObjectQuaternion(cube, path, quat)
    previousSimulationTime = t


end
```

Básicamente, con un código similar a este para cada uno de los objetivos, donde el parámetro con más impacto es la velocidad de movimiento, podremos tener múltiples objetivos que se desplacen a lo largo de trayectorias generadas manualmente con el procedimiento explicado.

7.3.2 Dinámicas y controladores discretizados mediante espacio de estados

Para los sistemas que han sido discretizados con el método del Apartado 5.1 hay que entender cómo se accede a los modelos generados, que corresponden a las dos etapas de control de los gimbals, es decir, el control en posición angular absoluta (etapa 1) y el control de regulación (etapa 2).

Estos modelos tras crearlos como se explicaba con Matlab, se almacenaron en el directorio '~/TFG/Modelos/', donde encontraremos una carpeta para cada modelo, y en cada una de ellas las matrices que definen los modelos discretizados.

Se organizan de acuerdo a lo que se ve en la **Figura 7-18.**

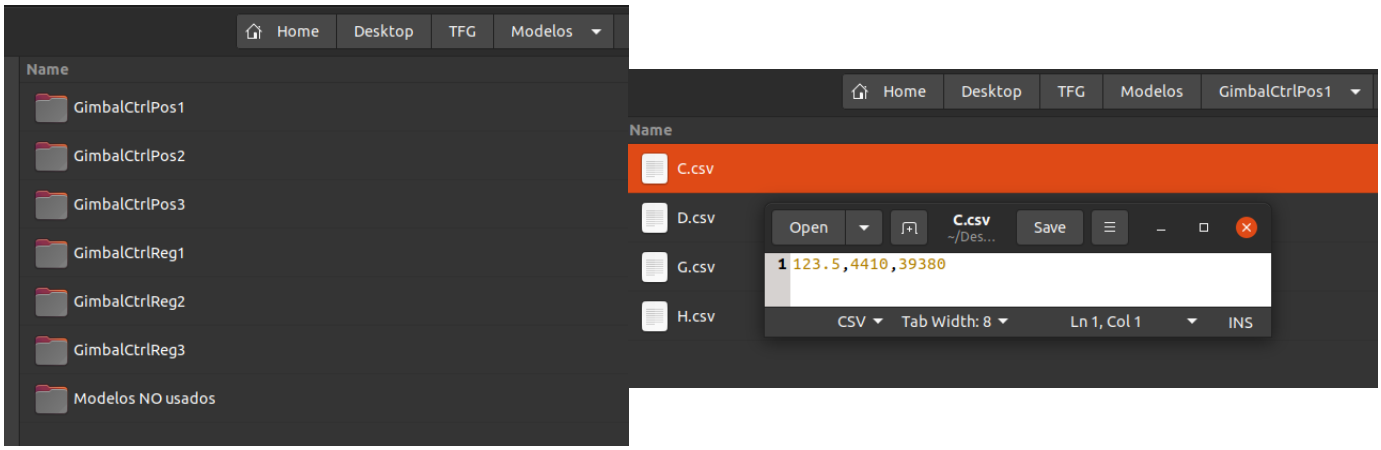


Figura 7-18. Localización de los diferentes modelos de control de gimbal para cada eje.

Como se aprecia en la **Figura 7-18**, cada carpeta contiene las matrices en formato CSV que modela uno de los sistemas discretizados. De esta manera, las matrices C, D, G y H de los diferentes sistemas pueden ser importadas en el código *Python* de los controladores y aplicarlas en la implementación de los controladores en espacio de estados.

7.4 Algoritmos de Visión

La implementación de los algoritmos de visión se realiza en una serie de scripts como son: *master_tracking.py, trackingCamGA.py, trackingCamOrientables.py, camerasParams.py,* entre otros que también incluyen funciones auxiliares.

Aunque ya se comentó resumidamente en el Apartado 7.1 lo que realizaba cada script, no está de más recapitular brevemente ciertas tareas principales antes de mostrar resultados de estos algoritmos.

En el script que parametriza las cámaras, se hace la reducción de resolución a un 10% de la original, para tratar de mejorar el rendimiento de la simulación en la medida de lo posible. También se realizan otras tareas como fijar las distancias focales máximas y mínimas de cada cámara, hacer la conversión a FOV y fijar el tamaño del sensor de las cámaras.

Por su parte el algoritmo de segmentación explicado en el Apartado 4.1.1 se implementa y permite identificar aquellos elementos que se diferencien del fondo uniforme que constituye el suelo de la escena. Este análisis devuelve una máscara binaria que será etiquetada y a partir de la cual se obtienen *bounding-boxes* y centroides de cada objetivo, quedando totalmente definidos en la imagen.

Para llevar a cabo el seguimiento de múltiples objetivos se utiliza el paquete *Multitracker* de la librería *OpenCV*, que permite un seguimiento con mejor precisión para varios objetivos a la vez. La alternativa sería instanciar varios trackers individuales KCF, pero sería más costoso computacionalmente y menos preciso. Como dicha clase *Multitracker*, permite elegir que algoritmo de seguimiento se quiere aplicar a cada objetivo, bastará con configurarlo para que use KCF.

También es útil para mejorar la precisión del seguimiento visual el hecho de inicializar el algoritmo con una *bounding-box* de mayor tamaño que la del objetivo, es decir, haciendo que incluya cierta información de su vecindad o entorno.

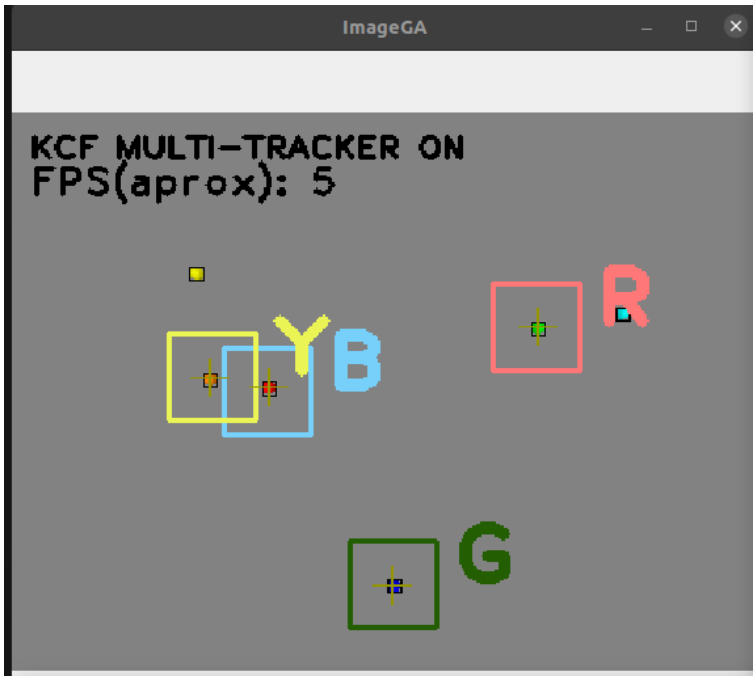


Figura 7-19. Segmentación, identificación y *multitracking* de diferentes objetivos, así como asignación a cada cámara orientable.

La imagen de la **Figura 7-19** es proporcionada por la cámara gran angular, la cual, con su gran campo de visión, permite captar bien los objetivos incluso si están alejados unos de otros. Los 6 objetivos de la imagen han sido identificados y marcados con una *bounding-box* de color negro (en la implementación final se elimina para facilitar la identificación de objetivos en cámara gran angular y la cámara orientable asignada). Por su parte, los objetivos asignados a cámaras orientables son encuadrados por una *bounding-box* de mayor tamaño del color correspondiente a la cámara orientable que lo seguirá y una letra identificativa. Esto se corresponde con:

- R (Red) → Cámara orientable roja.
- B (Blue) → Cámara orientable azul.
- G (Green) → Cámara orientable verde.
- Y (Yellow) → Cámara orientable amarilla.

Asimismo, también se incluye una cruz amarilla que identifica el centroide de los objetivos asignados.

7.4.1 Cámara gran angular

La cámara gran angular realiza tareas fundamentales en cuanto a gestión de objetivos y es la que resuelve la mayor parte de tareas del sistema, siendo que hasta inicialmente las cámaras orientables dependen de ella para apuntar a su objetivo asignado.

Una de sus tareas principales y que permite mejorar el avistamiento de objetivos desde esta cámara, es obtener la posición del UAV respecto del centro representativo del cluster conformado por todos los objetivos avistados por esta cámara gran angular.

Esta información se comunica al control de posicionamiento del UAV, que buscará centrar el UAV respecto a dicho punto representativo.

Con esto se logran dos aspectos esenciales: situar al UAV en una posición ventajosa para avistar y apuntar adecuadamente a los diferentes objetivos. También permite seguir el movimiento general de los objetivos, si es que este fuese relativamente uniforme, tratando de evitar así perder objetivos identificados.

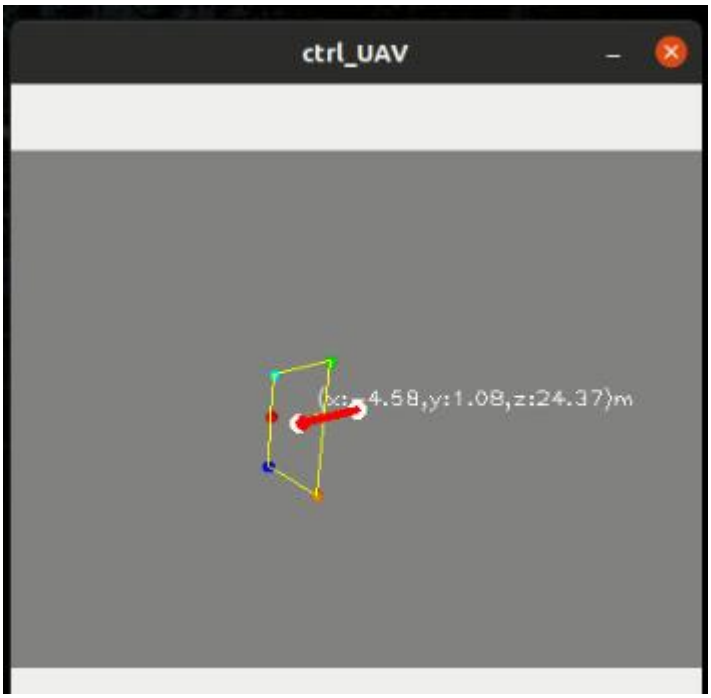


Figura 7-20. Posición del UAV respecto del punto de referencia (centro del cluster global).

La flecha de color rojo indica la distancia entre ambos puntos, teniendo dos puntos blancos al inicio y al final de dicha flecha. El punto en el que nace la flecha corresponde al centro de la imagen de la cámara gran angular, mientras que el punto blanco donde acaba la flecha corresponde al centro representativo de los objetivos avistados. Además, se aprecia en amarillo el polígono convexo constituido por la agrupación de objetivos, a partir del cual se calcula el mencionado centro representativo.

Por otra parte, requiere mención el hecho de que la distancia se define en metros, siendo que la del eje z se corresponderá en todo momento con la estimación de la altura del UAV proporcionada por el altímetro.

Para emular el funcionamiento de un altímetro de una manera más realista, se obtiene de la simulación la altura del UAV y se le añade un ruido gaussiano con una desviación típica de 0.5m.

Una vez el control de posicionamiento del UAV logre situarlo en ese centro representativo, y las coordenadas X e Y (también visibles en la **Figura 7-20**) sean cercanas a cero, se iniciará el proceso de asignación inicial de objetivos y consecuentemente el resto de las tareas visuales.

La asignación inicial de objetivos de la **Figura 7-19**, se ha realizado de manera que se busca maximizar el grado de verticalidad de las cámaras orientables, mediante el algoritmo del Apartado 4.2.3.

A estos objetivos que sean asignados se les iniciará un algoritmo de *tracking* en esta cámara gran angular, con el objetivo de mantener en todo momento la información de qué objetivos están siendo seguidos por cada cámara y valores identificativos de cada uno, como su centroide y demás.

Una vez asignados los objetivos, se deben calcular los ángulos de apuntamiento para realizar un apuntamiento aproximado del objetivo asignado por parte de cada cámara orientable. Estos ángulos serán calculados y aplicados durante cierto tiempo (etapa 1) como referencia del control de posición angular absoluta de los gimbals, logrando apuntar con relativa precisión al objetivo asignado. Con esto se consigue que dicho objetivo sea el que quede más cercano al centro de la imagen, lo que permitirá su identificación de manera sencilla al conmutar al control de regulación (etapa 2).

En caso de que una vez el control de regulación de la cámara orientable esté centrando con mayor precisión al objetivo ocurran errores, bastará con volver atrás y realizar de nuevo ese apuntamiento aproximado, gracias a que se mantiene la identificación del objetivo en la cámara gran angular.

Para hallar los ángulos del primer apuntamiento aproximado basado en la imagen de la cámara gran angular (etapa 1), basta con resolver la geometría de la escena de la manera explicada en el Apartado 4.2.1.

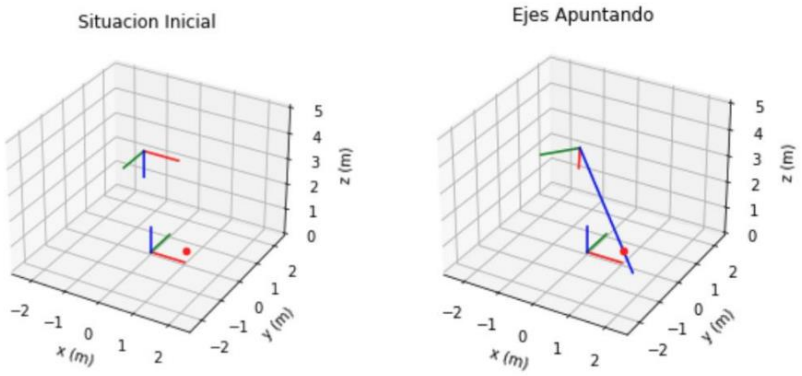


Figura 7-21. Ejemplo de apuntamiento en Matlab aplicando las ecuaciones del Apartado 4.2.1 [10].

Se puede ver en esta figura cómo se puede verificar la correcta orientación de los ejes de la cámara para apuntar al objetivo, siendo el eje X el rojo, eje Y el verde y eje Z el azul.

En cuanto al modo de funcionamiento basado en agrupaciones de objetivos, la generación e identificación de grupos se hace con el algoritmo *Kmeans* como ya se explicaba y cada cámara orientable tendrá asignada un conjunto de objetivos, en lugar de un solo objetivo individual.

La posición a la que se buscará apuntar será aquella dada por el centro del *convex-hull* calculado para cierto grupo de objetivos, de la forma explicada en el Apartado 4.3.1. En este caso, la asignación inicial de objetivos se basará en maximizar la verticalidad de la cámara al realizar el apuntamiento de ese centro del cluster.

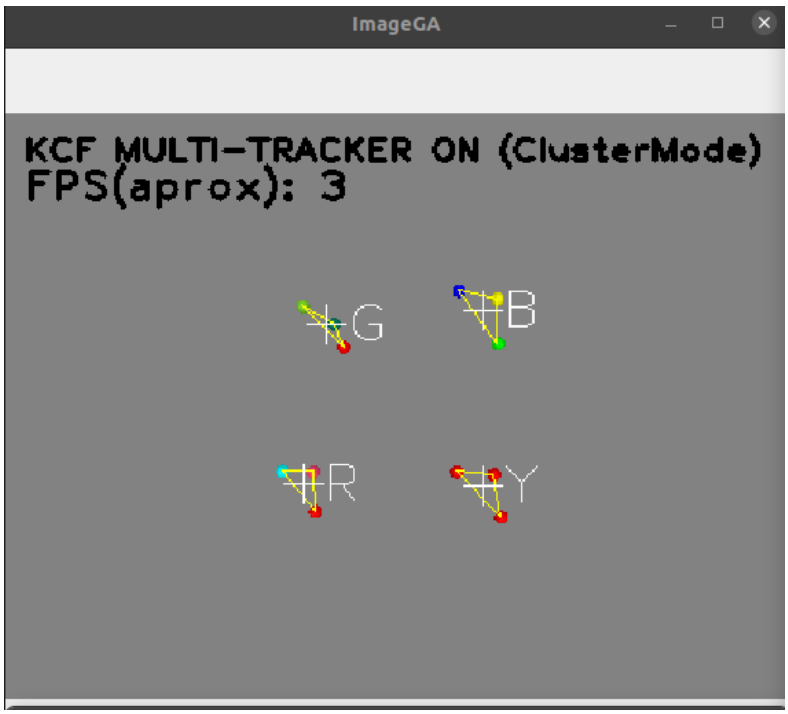


Figura 7-22. Asignación inicial de 4 clusters de objetivos en la cámara gran angular para 4 cámaras.

Para facilitar la visualización de los objetivos, sólo se muestra el centro representativo de cada cluster y el identificador de qué cámara ha sido asignada a cada grupo. Con esto es suficiente para comprender cómo se ha realizado la asignación de los diferentes grupos en función de la verticalidad de cada cámara.

7.4.2 Cámaras orientables

A partir de las funciones que resuelve la cámara gran angular, se habilita la posibilidad de que las cámaras orientables apunten a los objetivos asignados y obtener imágenes más cercanas y con mejor resolución de ellos.

Su número puede variar entre 2 y 4, por lo mencionado al comienzo del Apartado 7.1 y vendrá definido por el argumento que se indique al lanzar la simulación desde un terminal, donde se instanciarán archivos "bash" que hacen de lanzadores de todos los nodos y sólo contemplan como números válidos de cámaras orientables 2 y 4 de ellas, para el resto daría un mensaje de error.

La idea es partir de que, gracias al apuntamiento aproximado realizado con la información de la cámara gran angular, cada cámara orientable ya estará apuntando aproximadamente a su objetivo. Luego en primer lugar, se debe comenzar identificando en la imagen de la propia cámara orientable a su objetivo asignado. Para esto, es suficiente con considerar que el que le corresponde es aquel que queda más cerca del centro de la imagen, lo cual se consigue siempre gracias a la información de la cámara gran angular y su primer apuntamiento previo aproximado.

Tras identificar al objetivo, se inicia un algoritmo de *tracking* KCF para el mismo, y a partir de su posición en la imagen, se calcularán sus errores para el control de regulación.

El error angular cometido se obtiene según las ecuaciones (4-12), (4-13) y (4-14) del Apartado 4.2.2. De esta manera se está considerando la simplifación de que el centro óptico se encuentra en el centro de giro, lo cual ocurre en la simulación. Si no fuese así, seguiría siendo una aproximación válida, ya que el controlador absorbería el error cometido por la aproximación a la hora de cumplir su objetivo de llevar el centro de la imagen al punto a apuntar.

Podemos ver en la siguiente figura el apuntamiento aproximado realizado a partir de la cámara gran angular y el error respecto al centro de la imagen (punto blanco), que variará ligeramente por la estimación de la altura del UAV.

Figura 7-23. Apuntamiento aproximado al objetivo asignado (objetivo verde) a partir de información de cámara gran angular.

Se aprecia como este apuntamiento es relativamente preciso y permite al control de regulación identificar fácilmente a su objetivo asignado y tratar de centrarlo con mayor precisión.

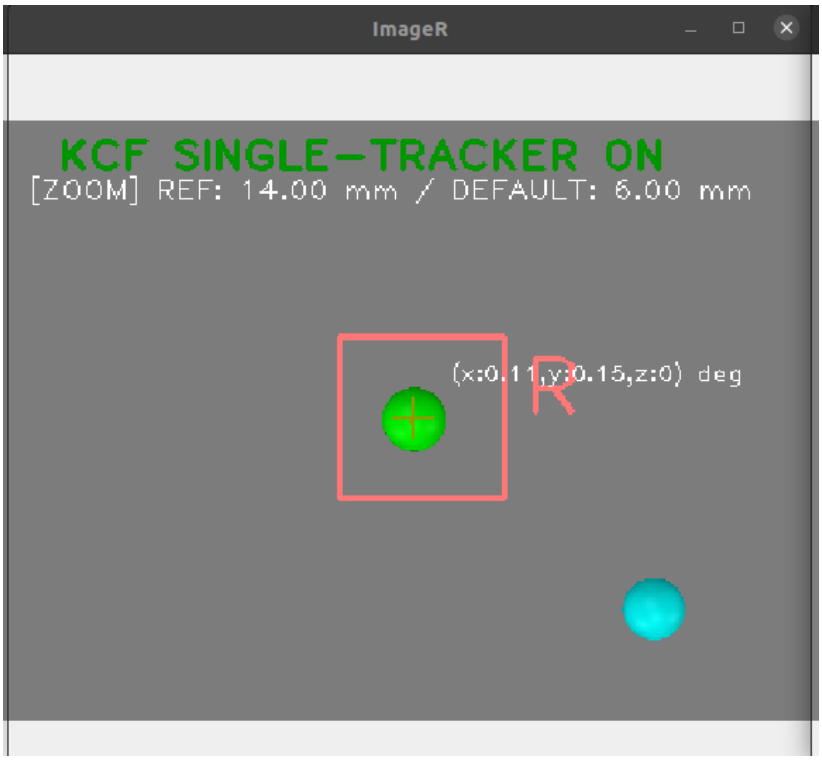


Figura 7-24. Apuntamiento de objetivo individual mediante control de regulación.

Este control tratará de reducir los errores angulares en grados que se aprecian a cero. Debido tanto al movimiento del UAV como de los propios objetivos, habrá desviaciones del objetivo respecto del centro de la imagen constantemente, aunque este control tratará de contrarrestarlo y centrarlo. Esto es fundamental ya que de lo contrario, debido a esos movimientos, el objetivo se perdería muy rápidamente de la imagen de la cámara.

Es importante también notar que se está aplicando zoom en modo de objetivos individuales, lo cual hace que cualquier movimiento del objetivo se "magnifique" en la imagen y cueste más contrarrestarlo, sin embargo, se considera adecuado aplicar el zoom, ya que la finalidad de las cámaras orientables es proporcionar una imagen detallada y de mayor resolución de su objetivo asignado. Si no se aplicase zoom, el objetivo se vería del mismo tamaño que en la **Figura 7-23**, con lo cual la diferencia entre el detalle con el que se ve respecto de la **Figura 7-24** es más que notable.

Pasando ahora al modo de agrupamiento de objetivos, los cambios son que se usa el centro del *convex-hull* en lugar de un centroide particular y que el algoritmo para obtener el valor de la distancia focal cambia como ya se explicaba en capítulos anteriores.

Una vez se ha hecho que las cámaras orientables apunten a los centros de las agrupaciones con la información de la cámara gran angular, se aplicarán las ecuaciones del capítulo 4.3.3 para modificar la distancia focal de la cámara para encuadrar correctamente al cluster en cuestión.

Lógicamente, los algoritmos de *tracking* se inicializarán una vez se encuadren los objetivos con el nuevo valor de zoom, para evitar cambios de escala que KCF no resolvería bien sin necesidad alguna.

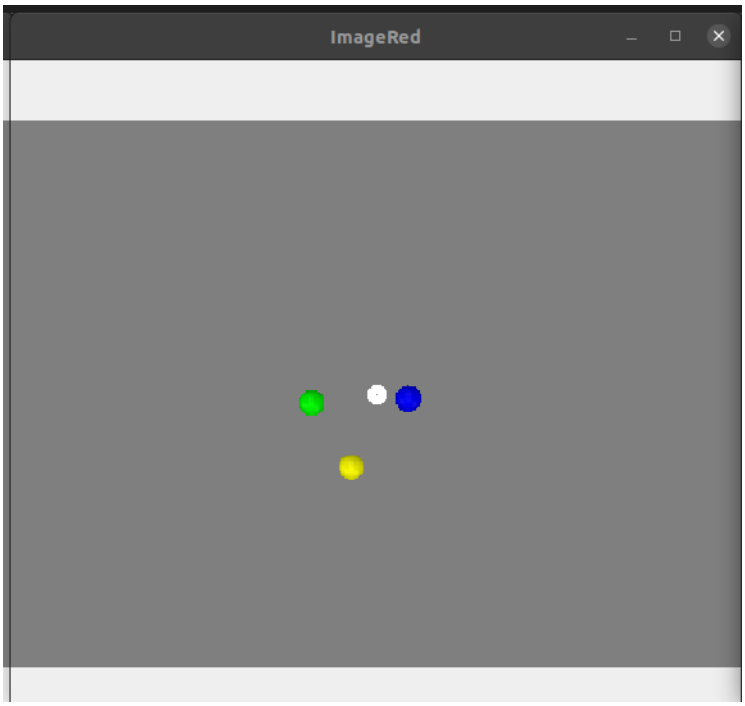


Figura 7-25. Apuntamiento aproximado a cierto grupo de objetivos a partir de información de cámara gran angular.

En esta imagen, se tiene ya apuntado al grupo con cierto error y sin aplicar zoom, luego se facilita la tarea para que la cámara orientable calcule la distancia focal para encuadrar adecuadamente a este grupo, aplicarlo y tratar de centrar el nuevo centro del *convex-hull* una vez aplicado dicho zoom.

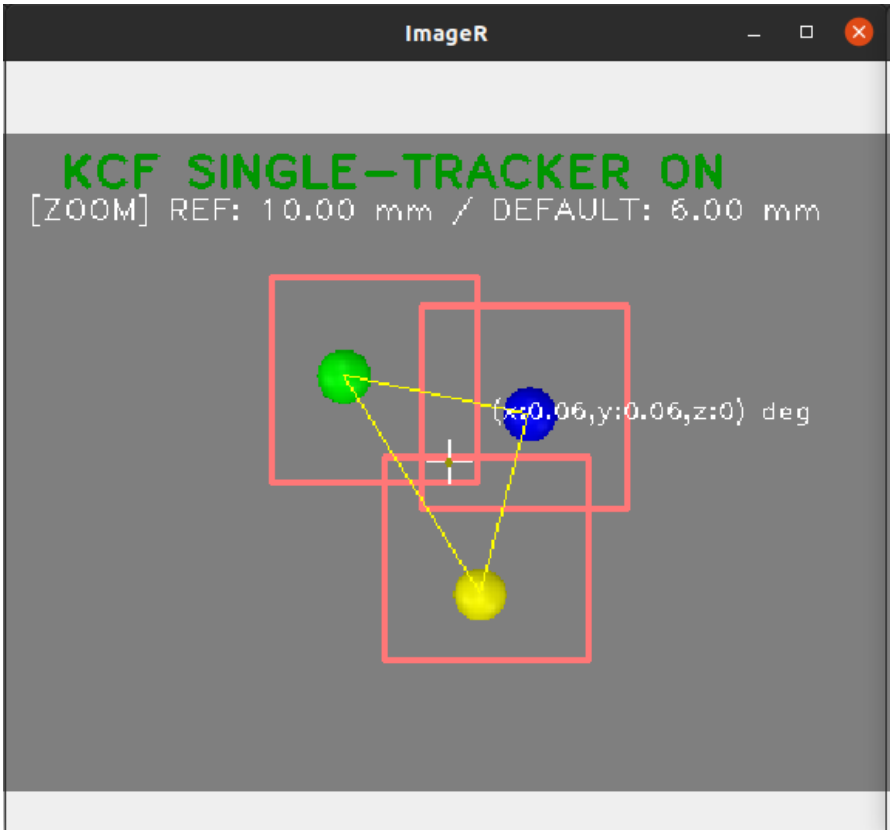


Figura 7-26. Ajuste del zoom y apuntamiento a un grupo de objetivos mediante control de regulación de una cámara orientable.

Una vez identificados los objetivos del grupo, se inicia un algoritmo KCF de *tracking* para cada uno de ellos, para mantener la identidad y posición de cada uno de ellos durante su movimiento, teniendo así al cluster identificado en todo momento para el control de regulación de la cámara orientable.

El nivel de zoom dado por la distancia focal se muestra en la parte superior de la imagen. El valor default corresponde al valor mínimo de distancia focal, que es el que se aplicaba en la **Figura 7-25**, y sirve como referencia para ver cuánto zoom se ha aplicado en esta fase respecto al paso anterior.

Tiene sentido que inicialmente, el zoom tome el valor mínimo para facilitar la localización del grupo en la imagen, y una vez localizado, ya se aplique el zoom correspondiente para tener una imagen más detallada.

También es interesante mencionar que, tanto para el modo de objetivos individuales como de agrupaciones, si en este control de regulación se producen pérdidas mantenidas en el tiempo, se considerará que ha habido error. Esto implicaría que se volvería a recurrir a la información de la cámara gran angular y repetir el proceso de apuntamiento previo con cámara angular (con zoom al mínimo) para luego pasar a control de regulación de nuevo.

7.5 Funcionamiento del sistema en su conjunto

Este apartado se plantea para recoger un resumen que recoja de manera ordenada cómo funcionaría el sistema de manera ordenada, resumiendo toda la implementación explicada previamente.

Los pasos generales a la hora de realizar un arranque ordenado de la simulación son:

1. Una vez se utilice uno de los archivos lanzadores tipo “bash” desde un terminal, el script *master_tracking.py* iniciará la simulación de CoppeliaSim que esté ya abierta, es decir, la escena debe haber sido manualmente abierta para poder comenzar. El resto de nodos esperarán unos 2s para que este lance la simulación de CoppeliaSim.
2. En primer lugar, se activa el control de posicionamiento del UAV y se le deja unos 10 segundos de margen actuando y centrando al UAV antes de lanzar el resto de tareas.
3. Transcurrido ese tiempo, se realiza la asignación de objetivos, se inician los algoritmos de seguimiento visual en la cámara gran angular y se generan los ángulos de orientación de referencia para cada gimbal.
4. Los gimbals reciben los ángulos de referencia y modifican su orientaciónde acuerdo al control de posición angular absoluta (etapa 1), logrando que las cámaras apunten aproximadamente a su objetivo o grupo de objetivos asignado.
5. Tras una serie de *frames* realizando este apuntamiento aproximado, se calcula el zoom necesario, con un algoritmo diferente si es objetivo individual o un grupo de objetivos, y se aplica dicho zoom.
6. Una vez modificado el zoom, se procede a identificar al objetivo asignado e iniciar algoritmos de seguimiento visual, que nos permitirán obtener los puntos de interés para calcular los errores angulares necesarios para el control de regulación.
7. Se conmuta finalmente al control de regulación y se procura llevar esos errores angulares a cero, buscando así un apuntamiento del objetivo o grupo de objetivos asignados a cierta cámara orientable.

En cuanto a la posibilidad de recuperación de errores, pueden darse dos escenarios problemáticos:

- Pérdida de los algoritmos de *tracking* en la cámara gran angular. Esto puede darse si por ejemplo un objetivo que estaba siendo seguido se sale de la imagen, o bien si el propio algoritmo deja de mantener el seguimiento. Esto implica que no se puede conocer qué objetivo está siendo seguido por qué cámara. Esto es una situación problemática y habrá que volver a aplicar todo desde los pasos 3 a 7.

- Errores o pérdidas de los algoritmos de *tracking* en alguna de las cámaras orientables durante cierto tiempo (es ideal dejarle cierto margen, ya que un fallo ante oclusión parcial, por ejemplo, puede rescatarse si se le deja algo de margen). En este caso, por como está programado, se opta por repetir los pasos 4-7 para todas las cámaras orientables. Con esto, se mantiene la asignación inicial dada por la cámara gran angular, pero se permite que la cámara que ha perdido a su objetivo lo vuelva a apuntar de forma adecuada.

7.6 Experimentos

Todos los archivos desarrollados pueden descargarse o visualizarse desde el siguiente enlace:

TFG_FcoJavierRomanCortes_CoppeliaSim

Para poner a prueba todo lo implementado, se desarrollan una serie de escenas y lanzadores.

Escenas:

- **MOT_4_cameras_movobjScript.ttt**: Escena diseñada para que el sistema funcione con el movimiento de los objetivos siendo comandado por script externo.
- **MOT_4_cameras_movTrayIndiv.ttt**: Escena diseñada para que la misma contenga programadas internamente un conjunto de trayectorias para objetivos individuales que se entrelazan.

Lanzadores:

- **Launch.sh:** con primer argumento el modo de funcionamiento (clustering o individual) y segundo el número de cámaras orientables. Este lanzador incluye al nodo que mueve a los objetivos de manera remota, está diseñado para la primera de las dos escenas anteriores.
- **LaunchTrayIndiv.sh**: con primer argumento el modo de funcionamiento (clustering o individual) y segundo el número de cámaras orientables. Este lanzador excluye al nodo que mueve a los objetivos de manera remota, luego está diseñado para la segunda de las dos escenas anteriores.

Para poder replicar las simulaciones, se parte de que se usa como sistema operativo *Ubuntu 20.04***.** Debería funcionar en *Windows* pero no se ha comprobado ahí el sistema completo y generalmente el rendimiento de CoppeliaSim es peor en dicho sistema operativo.

Habría que instalar ciertas librerías necesarias para los programas de *Python* como *pyzmq, numpy, OpenCV* versión 4.7.0. y *matplotlib.* Esto se puede hacer típicamente con "*pip install <libreria>"*, excepto para el caso de *OpenCV* que sería con "*pip install opencv-python==4.7.0*".

Con esto, descargando el zip completo del proyecto y extrayéndolo, este ya incluye el programa CoppeliaSim para Ubuntu 20.04.

Hay ciertas carpetas que no son de interés por contener códigos o escenas de prueba utilizadas durante el desarrollo.

De esta manera, una vez se ha descomprimido el zip del proyecto completo, habría que realizar una serie de pasos.

Abrir un terminal y navegar hasta el directorio del programa CoppeliaSim dentro del directorio descomprimido con un comando equivalente al siguiente según donde se encuentre dicho directorio:

cd ~Path_to_Downloads_or_where_the_zip_was_unfolded/CoppeliaSim_Edu_V4_5_1_rev2_Ubuntu20_04/

Tras ello, ya se puede ejecutar CoppeliaSim para abrir una de las dos escenas mencionadas anteriormente:

./coppeliaSim.sh /home/javier/Desktop/TFG/MOT_4_cameras_movobjScript.ttt

./coppeliaSim.sh /home/javier/Desktop/TFG/MOT_4_cameras_movTrayIndiv.ttt

Lógicamente, habría que modificar la ruta de los comandos anteriores por la del ordenador en que se esté tratando de ejecutar.

Con esto, se abriría CoppeliaSim con la escena en cuestión ya cargada.

Ahora, en otra pestaña del mismo terminal, o bien una ventana nueva, habría que navegar de nuevo hasta el directorio del TFG:

cd ~Path_to_Downloads_or_where_the_zip_was_unfolded/

Tras ello, ya se puede ejecutar los lanzadores, para lo cual se haría de la siguiente manera:

bash launch.sh individual n_cams_orientables

bash launch.sh clustering n_cams_orientables

bash launchTrayIndiv.sh individual n_cams_orientables

bash launchTrayIndiv.sh clustering n_cams_orientables

Donde "n_cams_orientables" sólo tiene como valores admitidos 2 o 4. Con esto ya se ejecutaría la simulación completa junto a todos los nodos ya explicados.

Las simulaciones realizadas y con las que se obtiene el vídeo demostrativo proceden de las siguientes combinaciones de escenas y lanzadores:

MOT_4_cameras_movobjScript.ttt + bash launch.sh clustering 2: Plantea el caso en que los objetivos de la escena se mueven formando 2 agrupaciones y el programa planteará el uso de 2 cámaras orientables para seguir cada agrupación.

MOT_4_cameras_movobjScript.ttt + bash launch.sh clustering 4: Igual que el caso anterior pero generando un mayor número de objetivos y hasta 4 agrupaciones, una para cada una de las 4 cámaras orientables que ahora estarán funcionando.

Para esta escena también se puede combinar con **bash launch.sh individual 2/4**, donde cada cámara seguirá un objetivo, ya sean del mismo cluster o diferentes, eso sí nunca repitiendo el objetivo de otra cámara.

MOT_4_cameras_ LaunchTrayIndiv.ttt + bash launchTrayIndiv.sh individual 2: Carga una simulación con movimientos inconexos y que se cruzan entre sí. Permite comprobar ciertas características como reinicio ante errores, robustez frente a oclusión parcial, reacción ante pérdida de objetivos entre otros.

MOT_4_cameras_ LaunchTrayIndiv.ttt + bash launchTrayIndiv.sh individual 4: Lo mismo que la anterior, pero con todas las cámaras orientables.

En la carpeta "VideosSimulaciones" se encontrarán diferentes videos demostrando el funcionamiento del sistema, sin embargo, no está de más dejar una serie de capturas para diferentes modos de funcionamiento y diferentes escenas.

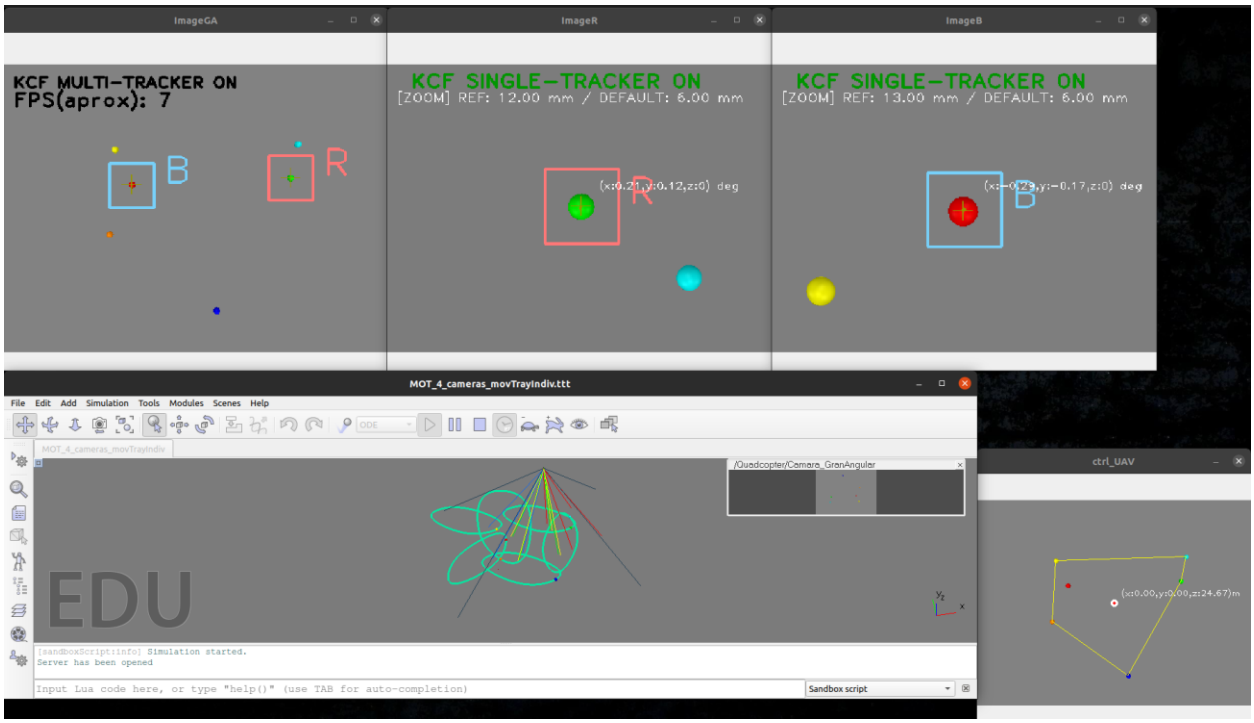


Figura 7-27. Experimento para sistema de 2 cámaras orientables siguiendo a objetivos individuales.

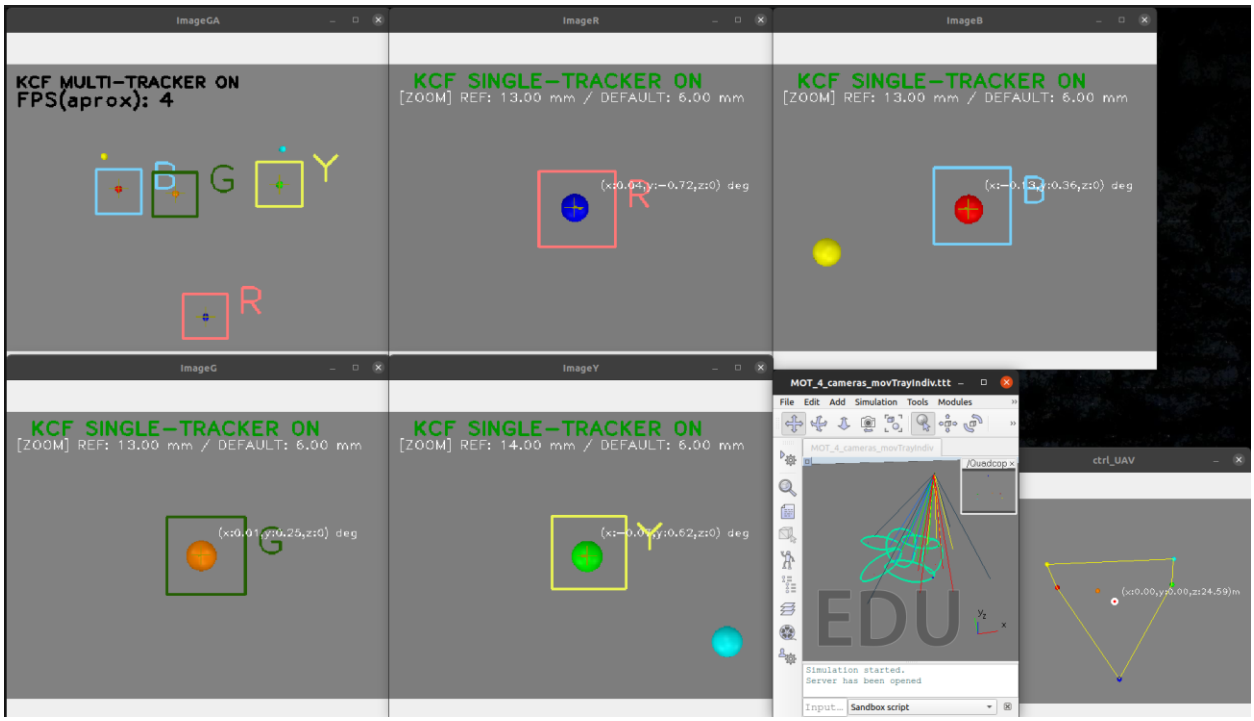


Figura 7-28. Experimento para sistema de 4 cámaras orientables siguiendo a objetivos individuales.

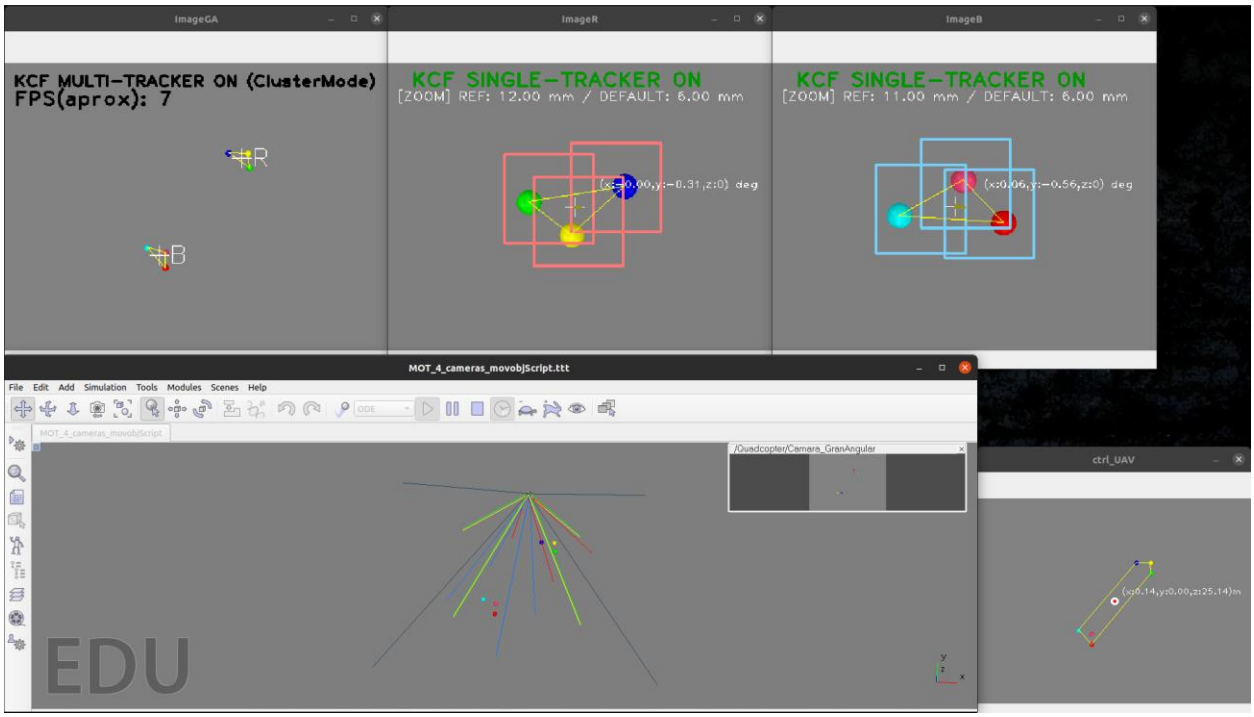


Figura 7-29. Experimento para sistema de 2 cámaras orientables siguiendo a agrupaciones de objetivos.

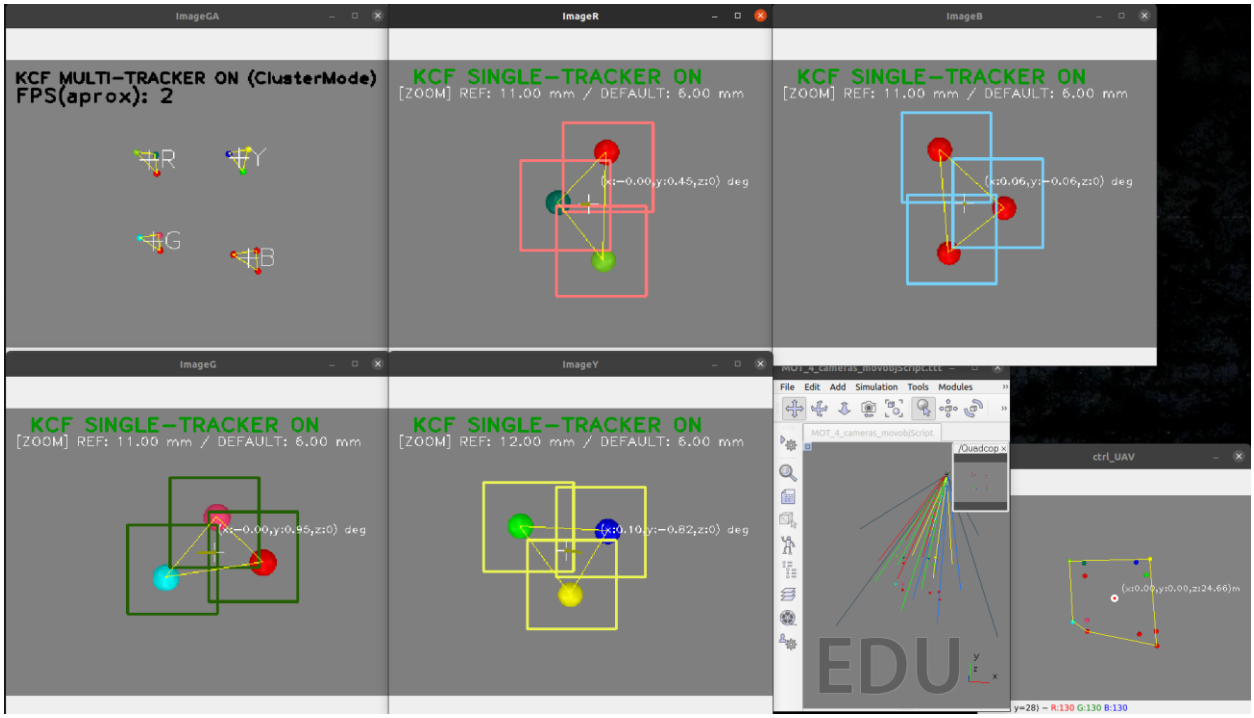


Figura 7-30. Experimento para sistema de 4 cámaras orientables siguiendo a agrupaciones de objetivos.

Nuevamente, los vídeos se pueden acceder a través de: Videos_Simulaciones

Donde basta con darle al botón verde “Code” y “Download Zip” para descargar el proyecto, descomprimirlo y dichos videos estarán en la carpeta VideosSimulaciones.

8 CONCLUSIONES Y LINEAS DE AMPLIACIÓN

A lo largo del proyecto, se ha logrado resolver los diferentes objetivos planteados inicialmente, que originan de forma natural problemas a resolver durante el desarrollo de los diferentes subsistemas que conforman un sistema completo complejo.

La gran novedad de este proyecto es introducir el simulador CoppeliaSim y verificar que es funcional a la hora de simular sistemas del grado de dificultad del planteado en este proyecto. Dentro de ciertas limitaciones en los detalles de modelado o en el rendimiento de las simulaciones por falta de información al no ser tan popular y tener tanto soporte como ROS, el foro es una herramienta vital a la hora de plantear y resolver dudas de implementación y modelado y recomiendo personalmente hacer uso del mismo si se tiene que trabajar con CoppeliaSim.

Además de plantear y explicar toda la base teórica que se aplica en la implementación del sistema, se presenta también lógicamente el esquema software que valida dicha teoría y simula un sistema que podría llevarse a tareas prácticas reales, como podrían ser detección de objetivos no esperados en tareas de vigilancia, tareas de agricultura, inspección de instalaciones, entre otras.

Respecto a la idea de llevarlo a un sistema real, considerando que para incluir este tipo de algoritmia se necesitaría algún microcontrolador potente, estos típicamente llevan programas compilados escritos en lenguaje C o C++, luego habría que reformatear el código que interese a alguno de estos lenguajes, lo cual debería ser una tarea asequible. Considerando que a la hora de trabajar con el sistema real habría mucha más flexibilidad en cuanto a rendimiento, se podrían mejorar los algoritmos y obtener mejores resultados. Luego este proyecto sienta una posible base teórica y programática a una posible implementación real, teniendo en cuenta claramente que, en el mundo real, los escenarios serían mucho más heterogéneos y requirirían probablemente algoritmos de segmentación y seguimiento más complejos y potentes.

Por otro lado, en cuanto a las posibles líneas de mejora, se podría plantear la idea de modelar una escena más compleja con entidades como personas, otros robots, vehículos, etc., y según cual interese, ser capaz de identificarlo respecto del resto, siendo necesario para ello explorar algoritmos de inteligencia artificial que sean capaces de realizar esta tarea.

Otra potencial fuente de mejora sería el plantear el esquema de nodos similar al de este proyecto pero en un lenguaje compilado, lo cual, conocida la gran mejora de rendimiento de estos lenguajes frente a los interpretados (*Python*, por ejemplo), podría implicar una notable mejora en el rendimiento de la simulación, eso junto a tal vez una posible aceleración del código de OpenCV mediante GPU, si es que el ordenador tuviese una potente y dedicada (no integrada como en mi caso, que está muy limitada en cuanto a potencia de cómputo).

APÉNDICE A: CÓDIGOS DE MATLAB

Código A.1 Función para discretizar en formato de espacio de estados una función de transferencia continua.

```
% Cálculo de las matrices en espacio de estados discreto
% dada una función de transferencia:

function [G,H,C,D] = calculoGHCD(G_sist, Tm, str)

% ----- CÁLCULO DE LAS MATRICES ---------------
[num, den] = tfdata(G_sist, 'v');

% Pasar a espacio de estados
[A,B,C,D] = tf2ss(num, den)

% Discretizar:
[G,H] = c2d(A,B,Tm);    % Caso general
% [G,H,C,D] = c2dm(A,B,C,D,Tm,'tustin');  % Posible version para discretizar controlador UAV

% ---------------- COMPROBACIÓN ---------------

N = 50;
t = 0:Tm:(N-1)*Tm;

% Escalón
u = zeros(1,N);
u(1:N) = 1*ones(1,N);

% Sistema
n = size(A);
n = n(1,2);
x = zeros(n,N);
y = zeros(1,N);

% Iteraciones
for k = 1:N
    x(:, k+1) = G*x(:,k) + H*u(k);
    y(k) = C*x(:,k) + D*u(k);
end

figure();
plot(t,y,'*r');
hold on;
step(G_sist, t);
grid();
axis([0, N*Tm, 0, max(y)*1.01]);
title("Step Response");

% ---------------- GUARDAMOS CSV ---------------
mkdir(str)
writematrix(G,str + "/G.csv");
writematrix(H,str + "/H.csv");
writematrix(C,str + "/C.csv");
writematrix(D,str + "/D.csv");

disp("Descripcion interna discreta")
disp("----------------------")
disp("[G]")
disp(G)
disp("[H]")
disp(H)
disp("[C]")
disp(C)
disp("[D]")
disp(D)
disp("end")


end
```

Código A.2 Código para discretizar usando Euler II (comprobación de discretización manual).

```
%% Sistema y controlador continuos:
% Función de transferencia del drone para traslación en cada dirección:
tau = 0.3;
N = 1;
D = [tau,1,0];

% Controlador:
tauC = 0.8296;
Kp = 1.741;
Nc = Kp * conv([tauC,1],[tauC,1]);
Dc = [1,0];

% Parámetros del controlador en forma de PID estándar:
Ti = 2*tauC
Td = Ti/4
Kc = Kp*Ti
Nc2 = Kc*[Ti*Td,Ti,1];
Dc2 = [Ti,0];

% Función de transferencia en BA:
Nba = conv(Nc,N);
Dba = conv(Dc,D);

Nba2 = conv(Nc2,N);
Dba2 = conv(Dc2,D);

% Función de transferencia en BC:
[Nbc,Dbc] = cloop(Nba,Dba,-1);
[Nbc2,Dbc2] = cloop(Nba2,Dba2,-1);

% Superponemos ambas respuestas, para comprobar que son iguales:
figure(1); step(Nbc,Dbc); grid on;
hold on;
step(Nbc2,Dbc2);
hold off;



%% Discretización:
Tm = 0.01;
% Discretización del sistema mediante mantenedor de orden cero:
[Nd,Dd] = c2dm (N,D,Tm,'zoh')

% Discretización del controlador por Euler II:
q0 = Kc * (1+Tm/Ti+Td/Tm)
q1 = Kc * (-1-2*Td/Tm)
q2 = Kc * Td/Tm
Ncd = [q0,q1,q2]
Dcd = [1,-1,0]

% Función de transferencia en BA:
Nbad = conv(Nd,Ncd);
Dbad = conv(Dd,Dcd);

% Función de transferencia en BC:
[Nbcd,Dbcd] = cloop(Nbad,Dbad,-1);

% Simulamos respuesta ante escalón para la misma duración (7s -> 700 muestras):
nMuestras = 700;
figure(2); dstep(Nbcd,Dbcd,nMuestras); grid;

% Si queremos podemos mostrar la respueta con escalones frente al tiempo en segundos:
tDiscr = Tm * [0:nMuestras-1]';
yd = dstep(Nbcd,Dbcd,nMuestras);

figure(3); stairs(tDiscr,yd); grid;
```

Código A.3 Simulación de las ecuaciones que modelan la dinámica del zoom óptico.

```
% Prueba de las ecuaciones que discretizan una dinámica de segundo orden
% correspondiente a la dinamica del zoom optico
%%%%%%%%%%%%%%%%%%%%%%%%%
% Datos del modelo (G(s))
tau1 = 0.1;
tau2 = tau1/3;
K = 1;
% Valores de saturacion
ypMin = -20;
ypMax = 20;
% Datos simulación
n = 2000;
T = 0.01;
% Variables
u = 20*ones(1,n);
u(1,(n/2+1):n) = 40*ones(1,n/2);
y = zeros(1,n);
yp = zeros(1,n);
% Constantes
C1 = K/(tau1*tau2);
C2 = (tau1+tau2)/(tau1*tau2);
C3 = 1/(tau1*tau2);
%%%%%%%%%%%%%%%%%%%%%%%%%
% Simulación
for k = 2:n
% Primer Integrador
yp(k) = yp(k-1)*(1 - T*C2) + u(k-1)*C1*T - y(k-1)*T*C3;
% Saturador
if yp(k-1) > ypMax
ypsat1 = ypMax;
elseif yp(k-1) < ypMin
ypsat1 = ypMin;
else
ypsat1 = yp(k-1);
end
% Segundo Integrador
y(k) = T*ypsat1 + y(k-1);
end
tdisc = 0:T:(n-1)*T;
figure();
plot(tdisc,y,'.','Color',[1,0,0]);hold on;
plot(tdisc,u,'Color',[0,0,1]);
xlim([0,20]);
ylim([0,45]);
ylabel('f (mm)');
xlabel('t (s)');
title('Dinamica discretizada del zoom optico frente a referencia');
legend('Referencia de distancia focal', 'Distancia focal dinamica');grid;

% Linea para plotear la salida de simulink si se ha ejecutado:

% plot(y_simulink.time, y_simulink.signals.values, 'Color', [0,1,0])
```

APÉNDICE B: CÓDIGOS IMPLEMENTADOS

Ante la gran extensión (aproximadamente 70-80 páginas) que supondría apendizar en esta memoria el conjunto de todos los códigos, se opta por dejar de nuevo un enlace directo a la carpeta del repositorio donde se encuentran todas las versiones finales de los códigos que implementan la funcionalidad final descrita del sistema y con comentarios explicativos cada función usada:

Código

REFERENCIAS

[1] Wikipedia, «Payload,» 2020. [En línea]. Available: https://en.wikipedia.org/wiki/Payload. [Último acceso: 2023].

[2] David Daly, «A-Not-So-Short History of Unmanned Aerial Vehicles(UAV),» nd. [En línea]. Available: https://consortiq.com/uas-resources/short-history-unmanned-aerial-vehicles-uavs. [Último acceso: 2023].

[3] Château de Versailles, «The first hot air balloon flight,», nd. [En línea]. Available: https://en.chateauversailles.fr/discover/history/key-dates/first-hot-air-balloon-flight#:~:text=The%20first%20'aerostatic'%20flight%20in,in%20the%20history%20of%20humanity. [Último acceso: 2023].

[4] Tony Reichhardt, «Alfred Nobel's rocket camera,», 2009, [En línea]. Available: https://www.smithsonianmag.com/air-space-magazine/alfred-nobels-rocket-camera-117825125/#:~:text=Nobel%20wasn't%20the%20first%20to%20think%20of%20launching%20a,parachuting%20back%20to%20the%20ground. [Último acceso: 2023] .

[5] National Museum of the USAF, «Kettering Aerial Torpedo "Bug",», nd. [En línea]. Available: https://www.nationalmuseum.af.mil/Visit/Museum-Exhibits/Fact-Sheets/Display/Article/198095/kettering-aerial-torpedo-bug/. [Último acceso: 2023]

[6] Seth J. Frantzman, «How Israel Became a Leader in Drone Technology,», 2019, [En línea]. Available: https://www.meforum.org/58937/how-israel-became-a-leader-in-drone-technology. [Último acceso: 2023].

[7] Richard Whittle, «The Man Who Invented the Predator,», 2013, [En línea]. Available: https://www.smithsonianmag.com/air-space-magazine/the-man-who-invented-the-predator-3970502/. [Último acceso: 2023].

[8] Wikipedia, «Gimbal,» nd. [En línea]. Available: https://en.wikipedia.org/wiki/Gimbal. [Último acceso: 2023].

[9] Yongkun Zhou, Bin Rao, Wei Wang, «UAV Swarm Intelligence: Recent Advances and Future Trends,» 2020. [En línea]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9214446. [Último acceso: 2023]

[10] D. T. Ruiz, «Seguimiento de Múltiples Objetivos Móviles Mediante Vehículo Aéreo No Tripulado,» 2021. [Copy on File]. [Último acceso: 2023]

[11] H. Grabner, M. Grabner y H. Bischof, «Real-time tracking via on-line boosting,», *Bmvc,* vol. 1, nº 5, p.6, 2006.

[12] B. Babenko, M. H. Yang y S. Belongie, «Visual tracking with online multiple instance learning,», *IEEE Conference on computer vision and Pattern Recognition*, 2009.

[13] Z. Kalal, K.Mikolajczyk y J. Matas, «Forward-backward error: Automatic detection of tracking failures,», *20th International Conference on Pattern Recognition*, 2010.

[14] D. S. Bolme, J. R. Beveridge, B. A. Draper y Y. M. Lui, «Visual object tracking using adaptive correlation filters,», *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010.

[15] Z. Kalal, K.Mikolajczyk y J. Matas, «Tracking-learning-detection,», *IEEE transactions on pattern analysis and machine intelligence,* vol 34, p. 1409-1422, 2011.

[16] J. F. Henriques, R. Caseiro, P. Martins y J. Batista, «High-speed tracking with kernelized correlation filters,», *IEEE transactions on pattern analysis and machine intelligence,* vol 37, p. 583-596, 2014.

[17] A. Lukezic, T. Vojir, L. C. Zajc, J. Matas y M. Kristan, «Discriminative correlation filter with channel and spatial reliability,», *IEEE Conference on Computer Vision and Pattern Recognition,* 2017.

[18] OpenCV, «Optical Flow,» 2020. [En línea]. Available: https://docs.opencv.org/4.5.0/d4/dee/tutorial_optical_flow.html. [Último acceso: 2023].

[19] OpenCV, «Meanshift and Camshift,» 2020. [En línea]. Available: https://docs.opencv.org/4.5.0/d7/d00/tutorial_meanshift.html. [Último acceso: 2023].

[20] Srishti Yadav y Shahram Payandeh, «Understanding Tracking Methodology of Kernelized Correlation Filter,» 2018. [En línea]. Available: https://ieeexplore.ieee.org/document/8614990. [Último acceso: 2023].

[21] Henriques, J. F., Caseiro, R., Martins, P. and Batista, J. (2012). Exploiting the circulant 2 structure of tracking-by-detection with kernels. In Computer Vision ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part IV 12 (pp. 702-715). Springer Berlin Heidelberg.

[22] Chen, Z., Hong, Z. and Tao, D. (2015). An experimental survey on correlation filter-based tracking. arXiv preprint arXiv:1509.05520

[23] Aishwarya Singh, «A Valuable Introduction to the HOG Feature Descriptor,», 2023. [En línea]. Available: https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/, [Último acceso: 2023].

[24] Satya Mallick, «MultiTracker : Multiple Object Tracking using OpenCV (C++/Python),» 2018. [En línea]. Available: https://learnopencv.com/multitracker-multiple-object-tracking-using-opencv-c-python/. [Último acceso: 2023].

[25] Wikipedia, «Modelo de color HSV,», nd. [En línea]. Available: https://es.wikipedia.org/wiki/Modelo_de_color_HSV, [Último acceso: 2023].

[26] Z. W. Zhong, «Altitude errors for ultrasonic sensor in altitude control,», 2013. [En línea]. Available: https://www.researchgate.net/figure/Altitude-errors-for-ultrasonic-sensor-in-altitude-control_fig13_257346980 [Último acceso: 2023].

[27] Wikipedia, «Convex hull,», nd. [En línea]. Available: https://en.wikipedia.org/wiki/Convex_hull, [Último acceso: 2023].

[28] Wikipedia, «Gift wrapping algorithm,», nd. [En línea]. Available: https://en.wikipedia.org/wiki/Gift_wrapping_algorithm, [Último acceso: 2023].

[29] Wikipedia, «Centroide,», nd. [En línea]. Available: https://es.wikipedia.org/wiki/Centroide, [Último acceso: 2023].

[30] Wikipedia, «Shoelace formula,», nd. [En línea]. Available: https://en.wikipedia.org/wiki/Shoelace_formula, [Último acceso: 2023].

[31] Wikipedia, «K-means clustering,», nd. [En línea]. Available: https://en.wikipedia.org/wiki/K-means_clustering, [Último acceso: 2023].

[32] Wikipedia, «K-means++,», nd. [En línea]. Available: https://en.wikipedia.org/wiki/K-means%2B%2B, [Último acceso: 2023].

[33] Van Kreveld, «Smallest enclosing circles and more,», 2020. [En línea]. Available: https://www.cise.ufl.edu/~sitharam/COURSES/CG/kreveldnbhd.pdf,

Implementation: https://www.nayuki.io/page/smallest-enclosing-circle, [Último acceso: 2023].

[34] F. M. B. Prado, «Modelado y Control de un Quadrotor con Gimbal mediante la perspectiva Fly-The-Camera,» 2020. [Copy on File]. [Último acceso: 2023].

[35] StackOverflow, «Relationship between focal length and FOV,», nd. [En línea]. Available: https://stackoverflow.com/questions/73371549/calculate-fov-from-video-files, [Último acceso: 2023].