UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIEROS INDUSTRIALES VALENCIA

TRABAJO FIN DE GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

DISEÑO DE UN CONTROL DE POSICIÓN X-Y DE UN QUADROTOR BASADO EN UN ALGORITMO DE FLUJO ÓPTICO

AUTOR: FRANCISCO MIGUEL AROCA JIMÉNEZ

TUTOR: PEDRO GARCÍA GIL

Curso Académico: 2014-15

Índice de documentos

- Memoria
- Presupuesto
- Anexo de programación

Índice de la memoria

Índice del presupuesto

Índice del anexo

Glosario de ilustraciones

Se cita aquí el origen de las imágenes tomadas de fuentes externas.

2015

Memoria

1. Introducción

1.1. Conceptos generales

Los vehículos aéreos no tripulados (VANT o UAV en inglés) también llamados drones, constituyen un grupo muy amplio de aeronaves de tipologías muy distintas, desde configuraciones similares a los aviones clásicos, hasta otras más próximas al sistema de funcionamiento de los helicópteros, con varias hélices que proporcionan la sustentación necesaria.

Ilustración 1. Dron MQ-9 para uso militar

Ilustración 2. Dron de 8 hélices para uso civil

Dentro de este último grupo, también podemos encontrar diferencias principalmente en el número de hélices y en el sistema de propulsión. Así, un mayor número de hélices proporciona un mayor control de la orientación a costa de una ley de control más compleja debido al aumento de los grados de libertad.

Por otra parte, en cuanto a los sistemas de propulsión, principalmente se utilizan 2: motores eléctricos y motores de combustión interna. El sistema más popular es el basado en motores eléctricos debido a la facilidad de regulación, poco peso y versatilidad aunque tiene inconvenientes como las limitaciones de potencia y autonomía debido a la baja potencia específica de los motores y la baja energía específica de las baterías que los alimentan. Estos problemas no aparecen con los motores de combustión, su mayor potencia permite elevar mayor peso y la densidad energética del combustible ofrece una autonomía superior a las baterías, aun así este tipo de motores son de difícil regulación e introducen vibraciones indeseables en el sistema que pueden interferir en los sensores.

Ilustración 3. Dron casero propulsado con MCIA único unido a un sistema de correas

En nuestro caso, el quadrotor sobre el que se ha trabajado se trata de un dron con 4 hélices configuradas en forma de cruz y movidas por pequeños motores eléctricos. Esta es una de las configuraciones más populares por su versatilidad y sencillez de control.

Ilustración 4. Quadrotor objeto del trabajo

1.2. Modelo teórico

Una vez definido el marco de trabajo, pasamos a introducir brevemente las variables y grados de libertad del quadrotor y el modelo dinámico del mismo.

Para plantear un sistema de control hacen falta variables que controlar. Las variables habituales utilizadas en este ámbito son los 3 ángulos de orientación (pitch, roll, yaw) relativos a un sistema de referencia ligado al quadrotor y las 3 coordenadas cartesianas globales del quadrotor (x, y, z).

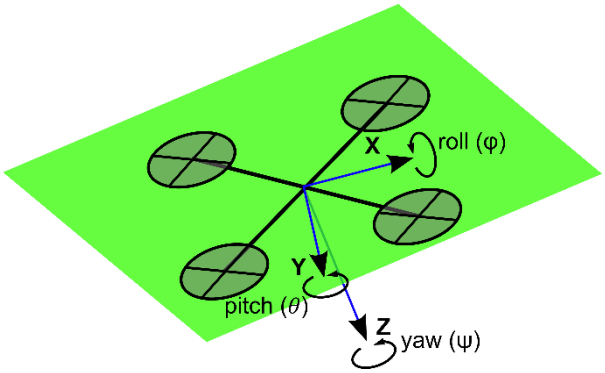

Ilustración 5. Angulos de orientación (φ, θ, ψ)

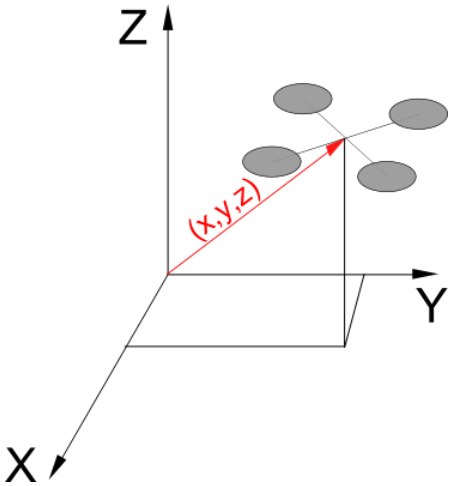

Ilustración 6. Coordenadas globales (x,y,z)

La posición y orientación del quadrotor se obtienen trasladando el origen de coordenadas del sistema global al centro del quadrotor, posteriormente efectuando el giro correspondiente a la medida yaw sobre el eje local Z, rotando después en torno al eje X del sistema de referencia resultante el ángulo roll, y finalmente, rotando de nuevo el ángulo pitch respecto al eje Y de forma análoga al giro anterior.

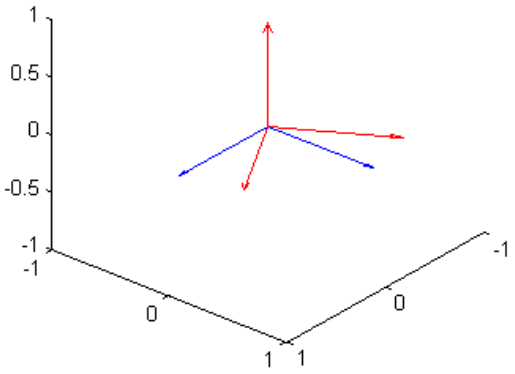

Ilustración 7. 1er giro (yaw)

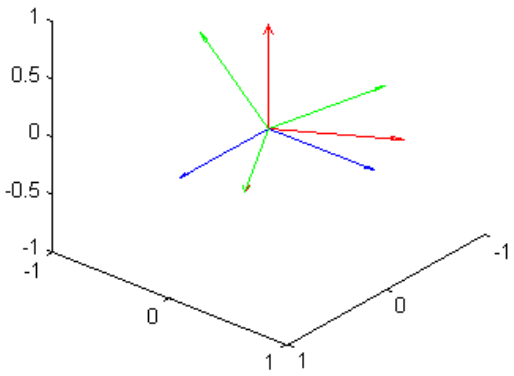

Ilustración 8. 2ndo giro (roll)

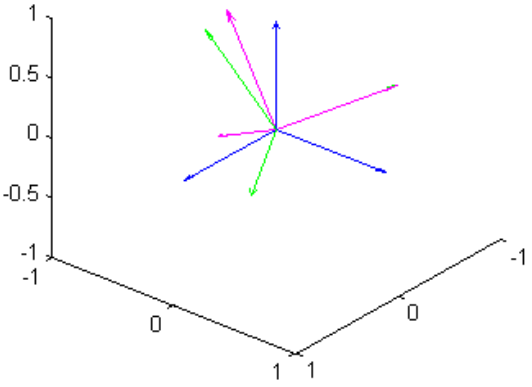

Ilustración 9. 3r giro (pitch)

Pasamos ahora a exponer el modelo dinámico del quadrotor [1], éste deriva de aplicar las ecuaciones de Euler-Lagrange a nuestro sistema incluyendo además la simplificación de que los ángulos pitch y roll son pequeños. El resultado es el siguiente sistema de ecuaciones:

$$
\begin{aligned}
m\ddot{x} &= -u\sin\theta \\
m\ddot{y} &= u\cos\theta\sin\varphi \\
m\ddot{z} &= u\cos\theta\cos\varphi - mg \\
\ddot{\psi} = u_{\psi} &= k_{\psi}\left(V_f - V_l + V_b + V_r\right) \\
\ddot{\theta} = u_{\theta} &= k_{\theta}\left(V_b - V_f\right) \\
\ddot{\varphi} = u_{\varphi} &= k_{\varphi}(V_l - V_r)
\end{aligned}
$$

Ecuación 1. Modelo matemático simplificado del quadrotor

Donde m es la masa del quadrotor, u es el empuje total de los motores, los parámetros k_{ψ}, k_{θ} y k_{φ} son constantes que engloban los momentos de inercia del quadrotor respecto a su centro, la relación entre voltaje aplicado al motor y velocidad de giro del mismo y la ley de las hélices que relaciona su velocidad de giro con la sustentación que proporcionan. Y las variables V_x son los voltajes aplicados a cada motor que constituirán la acción de control.

1.3. Motivación y justificación

La razón de ser de este trabajo, además de la finalización de los estudios de grado, es la resolución de los problemas que acarrea no controlar la posición del quadrotor que van desde la deriva excesiva y posible pérdida del aparato, hasta choques con obstáculos o personas que pueden ocasionar graves daños tanto al quadrotor cómo a aquello con lo que choca.

Procedemos ahora a analizar el modelo matemático para comprender la problemática del control en el plano X-Y. Tomemos como punto de partida la configuración inicial del quadrotor en el laboratorio, éste tiene implementado ya, gracias a trabajos anteriores de otros alumnos y personal investigador, un control de las variables de orientación (pitch, roll y yaw) y de la altura. Así pues para un vuelo "hover" (estático a una altura determinada) no hay más que ajustar las referencias de pitch y roll a 0º, consiguiendo así que el quadrotor permanezca paralelo al suelo y poner la referencia de la altura a un valor determinado constante z_0. Suponiendo que el control funciona correctamente y mantiene los ángulos de orientación en su referencia, analicemos que ocurre con las variables x e y de nuestro modelo teórico:

$$
\begin{matrix} m\ddot{x} = -u\sin\theta \\ m\ddot{y} = u\cos\theta\sin\varphi \end{matrix}
\xrightarrow[\varphi=0]{\theta=0}
\begin{matrix} \ddot{x} = 0 \\ \ddot{y} = 0 \end{matrix}
\longrightarrow
\begin{matrix} \dot{x} = v_x = C_x \\ \dot{y} = v_y = C_y \end{matrix}
$$

Ecuación 2. Problema de la deriva

Como podemos observar, de la ecuación 2 se extrae que el quadrotor se moverá con velocidad constante (no necesariamente nula) en el plano X-Y de forma análoga al movimiento que tendría un objeto en una pista de hielo. Este movimiento se comprueba de forma experimental al poner el quadrotor en vuelo, se observa como parece que se desplace en un plano imaginario paralelo al suelo y establecido a la altura de la referencia z_0.

El problema de deriva expuesto aquí, es un problema grave ya que, aunque no resulta crítico en un primer momento para la estabilidad en vuelo, la falta de control de la posición en X-Y sí que puede tener consecuencias negativas como las expuestas anteriormente. Además de estas razones de seguridad, hay razones de utilidad que motivan la implantación de un control de posición X-Y, como por ejemplo la capacidad hacer que el quadrotor siga rutas marcadas por el usuario.

1.4. Antecedentes y soluciones previas

Una vez establecido el problema a resolver, pasamos a examinar el campo de soluciones posibles. Los sistemas de posicionamiento en X-Y se pueden dividir en 2 grupos: sistemas autónomos y sistemas con dependencia externa.

Los sistemas con dependencia externa necesitan alguna referencia externa para calcular las variables de posición, destacan en este ámbito el sistema GPS, con un gran campo de aplicación en espacios exteriores pero con problemas de cobertura en interiores o el LPS (Local Positioning System) que permite calcular la posición del drone sin cobertura GPS utilizando un sistema de balizas en tierra para la triangulación.

Por otra parte, los sistemas autónomos se basan únicamente en la información que el propio drone pueda obtener de su entorno a través de sensores incluidos en el cuerpo del mismo. Estos sistemas tienen la ventaja de ser más versátiles y móviles a cambio de un aumento en el consumo y en la carga computacional del procesador. Como ejemplos podemos hablar de sistemas de posicionamiento por diodos láser como el Hokuyo, que calcula la distancia del drone a los posibles obstáculos que haya en su plano X-Y, el uso de este sistema está limitado a interiores por la necesidad de obstáculos para que el láser se refleje y así poder obtener medidas; otro sistema consiste en el cálculo de la posición a través de algoritmos de visión por computador, usando la información de una serie de imágenes del entorno tomadas durante el vuelo.

Este último sistema presenta varias ventajas que son las que nos han llevado a elegirlo como base para nuestro trabajo. El hecho de que sea un sistema autónomo da mayor independencia al quadrotor y la naturaleza de los sistemas basados en visión permite su uso tanto en exteriores como en interiores con el único requerimiento de que el ambiente esté debidamente iluminado.

1.5. Objetivos del trabajo

Tomando en consideración la información expuesta, determinamos como objetivo del presente trabajo el **diseño e implementación de un sistema de control de la posición de un quadrotor en el plano X-Y mediante un algoritmo de visión computacional**. La utilidad del sistema objetivo es inmediata ya que permite mover el quadrotor a través de coordenadas, solucionando además, por otra parte el problema de la deriva en el plano gracias a un sencillo control de posición.

Este objetivo global incluye una serie de objetivos intermedios que se pueden resumir en:

- Documentación sobre los fundamentos de la visión artificial y sobre los progresos actuales del quadrotor sobre el que se va a trabajar.
- Selección del material adecuado, cámara y placa.
- Instalación de la cámara en el conjunto del quadrotor.
- Preparación del entorno de programación para el tratamiento de imágenes.

- Implementación de un algoritmo computacional de cálculo y control de la posición en el plano X-Y.

Cabe mencionar que para cumplir estos objetivos, ha sido necesario un constante trabajo de aprendizaje transversal en campos tales como la programación en C++ orientada a objetos, el uso de las librerías de visión por computadora (OpenCV) o el trabajo con sistemas operativos Linux (Ubuntu), tanto en PC como embebido en placas computadoras.

Además ha resultado imprescindible aprender a trabajar en equipo con el resto de compañeros de desarrollo del quadrotor para la correcta coordinación de las tareas, así como para la toma de decisiones y para garantizar la compatibilidad e integración de los distintos trabajos individuales.

1.6. Normativa

En cuanto a la normativa aplicable, remitimos al lector al Boletín Oficial del Estado núm. 252 (17 de octubre de 2014) donde se establecen las condiciones necesarias para el vuelo de aeronaves no tripuladas.

2. Desarrollo del trabajo

2.1. Planteamiento general

El proceso de desarrollo del sistema objeto del trabajo ha consistido en la sucesiva consideración y descarte de soluciones para así llegar a la opción óptima con los recursos disponibles. Siguiendo esta premisa se pueden considerar los siguientes subapartados que, aunque expuestos por separado para mayor claridad, realmente están íntimamente interrelacionados:

1. **Documentación sobre los fundamentos de la visión artificial y el programa actual del quadrotor:**

La visión por computadora es un campo de la programación que no se trata en los contenidos del Grado en Tecnologías Industriales, por esta razón, antes de empezar a tomar decisiones sobre los materiales o software, resultó necesario un aprendizaje previo del uso de los algoritmos de OpenCV (librería de visión artificial libre) que se realizó durante las primeras semanas de forma simultánea con la consulta de material bibliográfico tanto físico como virtual, con esto se buscaba una mejor comprensión de los fundamentos de los algoritmos utilizados para así explotar mejor las capacidades de nuestro sistema.

De forma paralela se realizó también un análisis superficial del programa que se ejecuta en el quadrotor, prestando especial atención a los puntos críticos que pudiesen afectar o ser afectados por el algoritmo de visión a implementar. Más adelante, en la parte de desarrollo del algoritmo, se profundizaría más en este campo llegando a resultar determinante en el proyecto final.

2. **Selección del material: placa, cámara y sistema operativo**

Tras el estudio previo se pasó a seleccionar el hardware a utilizar, este no es un proceso trivial y consistió en una búsqueda de información combinada con un proceso de ensayo y error, uno de los principales problemas de este punto ha sido la compatibilidad entre los diferentes componentes, que ha limitado el campo de soluciones factibles.

El proceso se planteó del siguiente modo, primero compilamos un programa básico de cálculo del flujo óptico gracias a los conocimientos adquiridos en el apartado anterior, este programa ofrecía como resultados el tiempo de ejecución del algoritmo y los datos de flujo óptico (desplazamientos en el plano X-Y de la cámara), y probando este programa en las diferentes combinaciones de hardware (placa y cámara) y software se llegó a la solución adoptada desarrollada en apartados posteriores.

3. Implementación del algoritmo de flujo óptico para cálculo de la posición en el plano X-Y y diseño del controlador

Una vez elegida la plataforma física se ha pasado al refinamiento del algoritmo propiamente dicho, basándonos en un algoritmo anterior para luego actualizarlo y aplicar otras mejoras para reducir el tiempo de ejecución y conseguir una mayor precisión. Este proceso engloba mejoras de software así como mejoras en la propia ejecución del algoritmo buscando maximizar el rendimiento además de algunas correcciones necesarias por la naturaleza del sistema de visión y filtros para reducir el ruido presente en la medida.

2.2. Fundamentos de la visión por computador

Como primera aproximación al problema del cálculo del flujo óptico, se decidió consultar bibliografía al respecto, con este fin se buscó información tanto en el libro "Learning OpenCV" [2] como en la página web de documentación de OpenCV [3]. Exponemos aquí los conceptos más relevantes para nuestro propósito del trabajo, empezando por definir el flujo óptico a nivel cualitativo:

*"Optical flow or optic flow is the pattern of **apparent motion** of objects, surfaces, and edges in a visual scene caused by the **relative motion between an observer** (an eye or a camera) **and the scene**."*

(https://en.wikipedia.org/wiki/Optical_flow)

Analicemos la definición, en primer lugar, vemos que el flujo óptico como tal es un "patrón de movimiento aparente", no es una medida del movimiento real, el significado de esta afirmación se comprende fácilmente si se piensa en una cámara enfocando un plano o pared completamente blanca (o de cualquier otro color uniforme) iluminada con una luz igualmente uniforme, en estas condiciones, aunque la cámara se moviese, la imagen capturada no cambiaría obteniendo así una medida de movimiento aparente nula cuando en realidad sí que se está produciendo un movimiento. Pese a ser una posible limitación del sistema descartamos que este caso se produzca realmente por la dificultad de que se den las condiciones adecuadas para este fallo.

Por otra parte en la definición citada también se alude al carácter relativo de la medida, es decir, un sistema que sólo use como medida el flujo óptico interpretará de igual modo un movimiento de los objetos de la escena en una dirección y un movimiento de la cámara en la dirección contraria. En nuestro caso intentamos calcular el movimiento de la cámara (del quadrotor) así que este problema sí que nos podría afectar ya que necesitamos estar observando una escena estática para que así la única fuente de flujo óptico sea el movimiento de la propia cámara, esto es fácil de conseguir en vuelo a baja altura en espacios controlados, en un vuelo exterior a mayor altura resulta más complicado evitar las perturbaciones, por ejemplo, si un objeto cualquiera cruzase la escena (por ejemplo un coche o un

pájaro) el algoritmo interpretaría un movimiento en la dirección contraria a la del objeto obteniendo así una falsa medida de movimiento.

Cabe mencionar que la medida del flujo óptico será siempre la opuesta al movimiento real de nuevo por el carácter relativo de la medida que considera al quadrotor como sistema de referencia.

Definido ya el flujo óptico a nivel conceptual, pasemos a una definición matemática que nos permita obtener información útil, con este fin definimos el flujo óptico de un píxel (FO_p) como su cambio de posición entre dos imágenes:

$$FO_{p_i} = \begin{pmatrix} \Delta x_{p_i} \\ \Delta y_{p_i} \end{pmatrix} = \begin{pmatrix} x_{p_i} - x_{p_{i-1}} \\ y_{p_i} - y_{p_{i-1}} \end{pmatrix}$$

Ecuación 3. Definición del flujo óptico

En esta ecuación x_{p_i} e y_{p_i} son las coordenadas cartesianas del punto p en el instante i, asumimos ya que nuestro entorno de trabajo va a ser un computador y por lo tanto las medidas obtenidas van a tener un carácter discreto, es por eso que hablamos del instante i e $(i-1)$.

Así pues FO_{p_i} es una medida del desplazamiento de un píxel entre un instante i y el anterior. Además, sabiendo la posición inicial de un punto podemos saber su posición en el instante N usando la siguiente fórmula:

$$POS_{p_N} = \sum_{i=1}^{N} FO_{p_i} + \begin{pmatrix} x_{p_0} \\ y_{p_0} \end{pmatrix}$$

Ecuación 4. Definición de la posición de un píxel

Llegados a este punto el lector puede pensar que la notación utilizada para la posición es deliberadamente enrevesada, realmente la razón que hay detrás del uso de esta notación es la búsqueda de la semejanza con la implementación en el programa informático posterior ya que la medida que vamos a obtener de nuestro conjunto cámara-programa es exactamente FO_{p_i}. Más adelante en el apartado 2.4 retomaremos el uso de esta medida que requerirá un tratamiento de datos para resultar útil en el cálculo de la posición y evitar medidas falsas.

Pasemos ahora a aclarar un concepto que consideramos clave para la correcta comprensión del sistema de cálculo del flujo óptico de píxeles, ¿cómo se identifica un píxel para no confundirlo con otro al calcular el movimiento? Imaginemos una imagen en escala de grises, cada píxel tiene asignado un valor entero de tonalidad que va del 0 (blanco) al 255 (negro) con todos los tonos intermedios de gris. Supongamos que elegimos un píxel y queremos calcular su desplazamiento entre dos imágenes similares, como primera aproximación, podríamos almacenar el valor de tonalidad de ese píxel y buscarlo en la imagen siguiente, se puede ver fácilmente que este método muy probablemente nos daría como resultado un falso positivo ya que pueden haber más puntos en la imagen con el mismo valor de tonalidad.

En estas dos imágenes se puede detectar el fallo del sistema expuesto, si la figura de 3 píxeles del mismo tono mostrada en la imagen de la izquierda se mueve para pasar a la posición mostrada en la

imagen de la derecha, la medida de flujo óptico podría ser tanto los vectores rojos (medida incorrecta) como el vector azul (correcto) ya que los tres píxeles tienen el mismo valor de tonalidad.

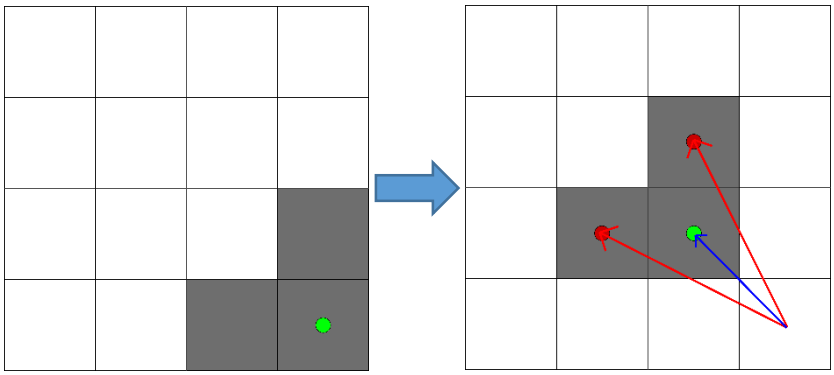

Ilustración 10. Medida de flujo óptico considerando un pixel

Como mejora de este sistema podemos pensar en considerar en lugar del valor del propio píxel, el valor de todos los píxeles en una ventana de X píxeles alrededor del píxel a seguir y buscar la coincidencia esto aumentaría la precisión de nuestro sistema considerablemente al reducir la cantidad de falsos positivos, de hecho, como más grande fuese la ventana X mejor precisión obtendríamos. Siguiendo con el ejemplo anterior:

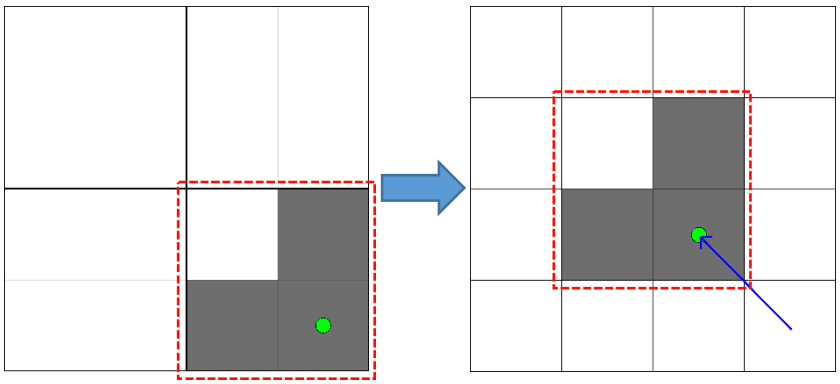

Ilustración 11. Medida de flujo óptico considerando una ventana de 2x2 píxeles

En la ilustración 8 podemos observar cómo al tomar una ventana más grande de píxeles, la probabilidad de error se reduce. Un posible error de este sistema de "ventana" se daría si alguno de los píxeles de la ventana cambiase de tonalidad entre una imagen y otra, el programa sería incapaz de encontrar la ventana completa y obtendríamos un nuevo error de medida.

Hay que tener en cuenta también que una ventana más grande de píxeles implica un mayor número de comprobaciones y, por lo tanto un mayor tiempo de computación, que queremos que sea pequeño para no interrumpir el resto de tareas del quadrotor, nos encontramos así ante un problema típico de precisión contra velocidad, más adelante en la implementación del algoritmo tendremos que tomar una decisión de compromiso entre estas dos variables.

Concluiremos este apartado con la clasificación y selección del algoritmo de flujo óptico a utilizar en el desarrollo del proyecto. Ya hemos visto cómo se calcula el flujo óptico de un píxel, ahora bien, ¿cómo

calculamos el flujo óptico de una imagen? Las opciones actuales son 2: flujo óptico denso o flujo óptico disperso. Expondremos brevemente las características y diferencias entre ambos:

- Flujo óptico denso: esta técnica consiste en intentar calcular el vector de flujo óptico para todos los píxeles de una imagen, el método usado es el de la "ventana" de píxeles expuesto anteriormente pero limitando el radio de búsqueda de esa ventana a las proximidades de la posición anterior del píxel para ahorrar tiempo de computación.
- Flujo óptico disperso: es una variante del flujo óptico denso que trata de calcular el vector de flujo óptico solo para unos determinados píxeles seleccionados previamente en lugar de hacerlo para todos.

Las diferencias entre ambos métodos son notables sobre todo en tiempo de computación, para ilustrar esto supongamos que queremos hacer el cálculo para una imagen de 480 x 320 píxeles, en el caso del flujo óptico denso habría que calcular el flujo óptico de 153600 píxeles mientras que con el flujo disperso podríamos elegir el número que quisiéramos, por ejemplo 100 píxeles es una cantidad suficiente, consiguiendo un tiempo 1536 veces menor. La característica favorable que tiene el flujo óptico denso es que no requiere ningún preprocesamiento de la imagen mientras que el flujo disperso necesita un paso previo de selección de píxeles a seguir que se puede realizar de forma manual o con algoritmos diseñados a propósito para seleccionar puntos de interés.

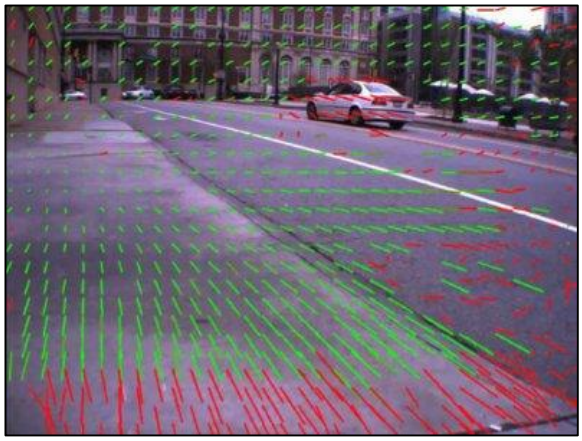

Ilustración 12. Ejemplo de flujo óptico denso

Ilustración 13. Ejemplo de flujo óptico disperso

Para decidirnos por uno u otro se compilaron unos programas de prueba de cálculo de flujo óptico por ambos métodos y, aunque en un PC de sobremesa la diferencia no era notable, probablemente por el uso de una GPU (unidad de procesamiento gráfico), en los sistemas embebidos utilizados sin GPU la diferencia sí que resultaba crítica, mientras que el flujo disperso consumía unos 80-100 ms para el cálculo, el flujo denso necesitaba entre 5 y 8 segundos, obteniendo medidas de precisión similar.

Siendo pues el tiempo de ejecución un punto crítico, nos decantamos por utilizar un algoritmo de flujo óptico disperso por utilizar menos píxeles para el cálculo permitiendo al procesador realizar otras tareas. Tras elegir esta opción se compiló un programa de cálculo del flujo óptico disperso simple que

busca una serie de puntos e intenta rastrearlos en la imagen siguiente y que sería utilizado como herramienta de “benchmarking” para ayudar a elegir el material a utilizar (hardware y software).

2.3. Selección del material (hardware y software)

En este punto se empezó con la selección del software base para el algoritmo de flujo óptico, de entre las diversas librerías de visión artificial disponibles se eligió OpenCV por diversas razones: se trata de software libre, multiplataforma con una gran comunidad de usuarios y una amplia documentación sobre su uso e instalación. Además, versiones anteriores de estas librerías habían sido utilizadas ya por otros alumnos e investigadores en otros proyectos similares con buenos resultados. Los programas diseñados en el apartado anterior fueron compilados con estas librerías. Más adelante, en el punto 2.4 se profundizará en el contenido y funciones de estas librerías.

Pasemos ahora a la selección de la plataforma hardware a utilizar, el conjunto placa-cámara, hablamos de estos dos elementos como conjunto porque, pese a tratarse de partes físicamente independientes, es imprescindible que sean compatibles para el correcto funcionamiento del sistema.

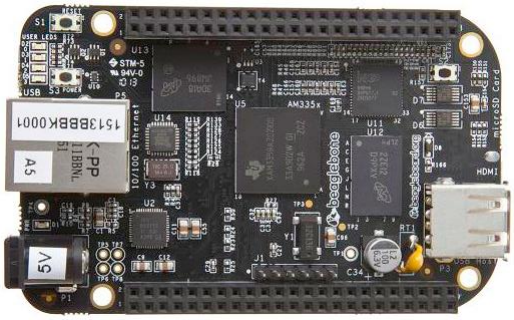

Ilustración 14. BeagleBone Black

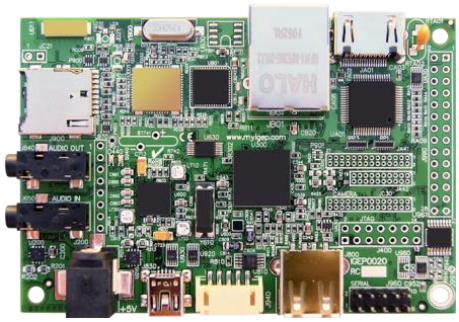

Ilustración 15. IGEPv2

Empezamos este proceso de selección comparando dos placas disponibles en el laboratorio, la IGEPv2 (actualmente en uso en el quadrotor) y la Beaglebone Black (BBB en adelante). Las especificaciones técnicas de ambas placas son similares tal y como exponemos en la siguiente tabla:

	IGEPv2	BBB
Procesador	DM3730 ARM Cortex A8 1Ghz	AM335x 1GHz ARM® Cortex-A8
Memoria RAM	512 MB	512MB
WiFi	Incluido en la placa	Con adaptador USB WiFi
Conexiones disponibles	1 x USB 2.0 1 x Ethernet 3 x UART (2 x RS232 y 1 x RS485) 3 x SPI 2 x I2C	1 x USB 2.0 1 x Ethernet 1 x UART
Precio	179 €	51.99 €

Tabla 1. Comparación de las especificaciones de las placas

La IGEPv2, al estar ya instalada en el quadrotor, estaba ya casi preparada para ejecutar nuestro programa de prueba, a falta de la instalación de las librerías OpenCV, que fueron compiladas en cruzado desde un ordenador de sobremesa para la arquitectura de la placa (ARM) dejando así la IGEPv2 lista para su uso.

La BBB, por otra parte, estaba prácticamente recién salida de fábrica, con lo cual fue necesario un proceso de preparación del entorno de programación instalando un sistema operativo basado en Linux (Debian) y aplicando un parche (Xenomai) al kernel de Linux que permite la ejecución de programas en "tiempo real", las tareas ejecutadas en tiempo real tienen máxima prioridad dentro del sistema, se les puede asignar una prioridad incluso mayor a la de las propias tareas del sistema operativo evitando así cualquier "cuelgue" del programa de vuelo que provocaría la caída del quadrotor.

El sistema operativo seleccionado para la BBB ha sido el disponible en la siguiente página web bajo licencia libre:

http://www.machinekit.net/deb/rootfs/wheezy/BBB-eMMC-flasher-debian-7.4-machinekit-armhf-2014-05-19-4gb.img.xz

Se ha elegido este sistema concreto por estar diseñado para el manejo de máquinas de fabricación por control numérico, las necesidades de este campo son similares a las nuestras y esto se ve en el software incluido en este sistema operativo dedicado que ya contiene el kernel con Xenomai y las librerías de OpenCV.

Una vez preparados los respectivos entornos de programación, se procede a ejecutar el programa de prueba que habíamos preparado en el apartado anterior. El experimento se ejecuta en las siguientes condiciones:

- Cámara inmóvil y entorno invariante.
- El programa de la cámara es el único programa en ejecución en la placa.
- Los resultados son impresos a fichero para no comprometer la velocidad del programa.

En las condiciones expuestas los resultados esperados son: una medida de desplazamiento nula y un tiempo de ejecución solo dependiente de las características de la placa y del sistema operativo.

Los resultados obtenidos son los siguientes:

	IGEPv2	BBB
Desplazamiento máximo (pix)	0.01	0.01
Tiempo de ejecución (mín – med – máx) (ms)	30 – 50 – 120	50 – 80 – 200

Tabla 2. Comparación del rendimiento del programa de prueba en las placas

Pasemos ahora al análisis de los resultados, a primera vista, sorprende que el desplazamiento máximo de píxeles sea un número decimal y no entero, este resultado se explica por la naturaleza del programa utilizado, que, tras obtener el valor individual de flujo óptico para cada píxel, pasa a calcular la media aritmética de todos estos valores que es lo que consideramos como flujo óptico total de la imagen.

Dicho esto, cabe remarcar que un “error” de 0.01 píxeles es un buen resultado que podemos asociar a pequeños cambios de iluminación o incluso al ruido presente en la imagen. Otra conclusión importante que extraemos de este resultado es que la precisión de la medida no depende de la placa utilizada sino más bien de la cámara en uso.

Por otra parte sí que encontramos diferencias en el tiempo de ejecución de la tarea, pese a que estamos trabajando en un entorno de tiempo real (Xenomai) el programa compilado no hace uso de estas funciones, lo que hace que pueda ser pospuesta su ejecución en favor de tareas internas del sistema, es por eso que no obtenemos un tiempo de ejecución fijo. Cabe mencionar que tras detectar este problema se intentó implementar el mismo programa como tarea de tiempo real pero sin éxito, al parecer las librerías de OpenCV no son compatibles con las tareas de Xenomai.

El problema expuesto no compromete la validez del experimento ya que se realizó del mismo modo en ambas placas, en los resultados se observa que la placa BBB tiene un tiempo de ejecución ligeramente más alto, creemos que el origen de esta diferencia es la interfaz de usuario del sistema operativo de la BBB, en la IGEPv2 no se ejecuta ningún entorno gráfico lo cual evita cargar innecesariamente el procesador.

Pese a que se podría haber desactivado la interfaz gráfica de la BBB para mejorar la velocidad de procesamiento, se encontró otro problema que llevó a tomar la IGEPv2 como placa de trabajo: los puertos USB. Para el funcionamiento autónomo del quadrotor es necesaria tanto la conexión WiFi para comunicarse con el PC en tierra como la cámara para hacer el control por flujo óptico. En el caso de la IGEPv2 esto no es un problema porque el WiFi está integrado en la propia placa y tiene un puerto USB para conectar la cámara, pero la BBB tiene un solo puerto USB al que se deberían conectar tanto la cámara como el dispositivo de WiFi, se probó un ampliador de puertos USB pero daba problemas con la alimentación así que se apartó por el momento la idea de utilizar la BBB y se tomó como placa de trabajo la IGEPv2 que además estaba ya instalada en el quadrotor y tenía los puertos configurados para las conexiones del resto de sensores.

Pasemos ahora a la selección de la cámara, las opciones consideradas son las 2 siguientes: Eye Toy Namtai y PlayStation Eye, ambas son cámaras diseñadas con fines de entretenimiento en juegos basados en la detección del movimiento del jugador, se ajustan pues a nuestras necesidades de detección de movimiento.

Ilustración 16. Eye Toy Namtai

Ilustración 17. PlayStation Eye

Las especificaciones técnicas de estas cámaras no se han encontrado y se ha procedido a su determinación empírica, en concreto nos interesan dos parámetros: la velocidad de captura o FPS, y

la resolución, que debe ser lo suficientemente alta como para detectar movimiento, pero sin llegar a valores demasiado altos porque una alta resolución de imagen aumenta exponencialmente el tiempo de cálculo del flujo óptico.

Para poder utilizar las cámaras es necesario instalar primero los drivers correctos, así pues se buscó el driver correspondiente a ambas cámaras y se comprobó si estaba instalado en la IGEPv2, con resultado satisfactorio. Una vez comprobado esto, se pasó a la extracción de parámetros característicos de las cámaras, con este fin se compiló un pequeño programa que toma fotos de forma cíclica a la máxima velocidad posible y comprueba el tamaño de las imágenes. Los resultados se expresan en la tabla siguiente:

	Eye Toy Namtai	PS eye
Velocidad (FPS)	60	180 ó 70
Resolución (pix x pix)	320 x 240 (0.0768 Mpx)	320 x 240 (0.0768 Mpx) ó 640 x 480 (0.3 Mpx)

Tabla 3. Características de las cámaras

Como podemos ver los FPS son mayores en el PS eye, sobre todo a baja resolución, esta diferencia se debe principalmente al driver instalado, modificado por la comunidad de usuarios de Linux para aumentar al máximo posible la velocidad de muestreo de la cámara.

La resolución de 320x240 píxel hemos comprobado en el laboratorio que es más que suficiente para nuestro propósito de detección de movimiento, además con esta resolución obtenemos un tiempo de cálculo mucho menor (4 veces menor) que con la resolución de 640x480 píxel, en el caso de querer utilizar la cámara para otros propósitos como grabación de vídeo o mapeado mediante imagen utilizaríamos la máxima resolución pero en este caso es prioritaria la velocidad de cálculo.

Con los datos que tenemos concluimos que la cámara PS eye se ajusta mejor a nuestros propósitos y es por eso que la elegimos como cámara a utilizar en el trabajo.

Con el objetivo de reducir peso y mejorar la integración física de la cámara en el conjunto del quadrotor, le quitamos todos los elementos de protección innecesarios e instalamos la cámara en la parte inferior, tal y como se muestra en las siguientes imágenes:

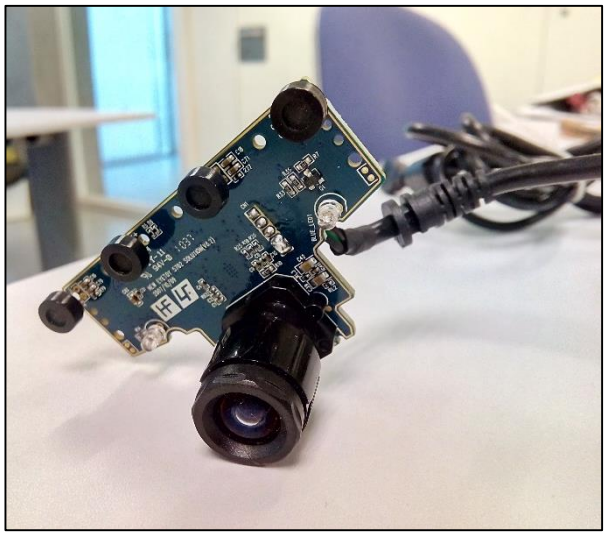

Ilustración 18. Cámara PS eye sin carcasa

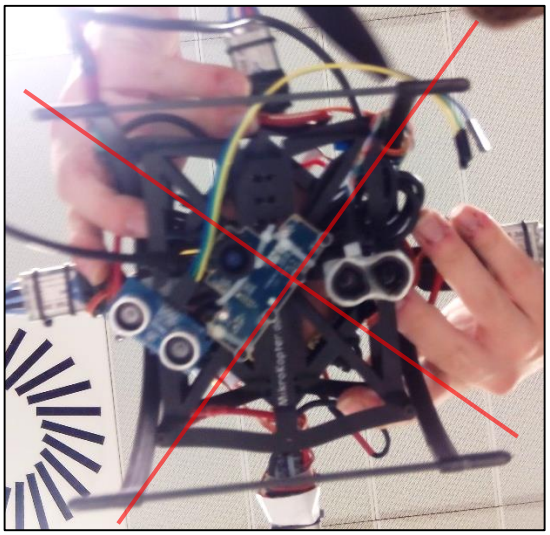

Ilustración 19. Detalle del montaje de la cámara (líneas rojas = ejes X-Y quadrotor)

2.4. Desarrollo del algoritmo de medición y control

El desarrollo del algoritmo de control se puede dividir en tres etapas:

- Medición
- Tratamiento de la información (filtrado)
- Control

Procedemos a explicar el desarrollo de cada una de ellas.

2.4.1. Algoritmos de medición

Antes de explicar el desarrollo y funcionamiento de los algoritmos de medición cabe mencionar que se han desarrollado paralelamente dos algoritmos: un primer algoritmo lo más general posible que pudiese funcionar en cualquier entorno, y otro más específico para entornos visuales semi-controlados. Estos dos algoritmos serán explicados de forma independiente puesto que son muy diferentes tanto en contenido como en base teórica, coincidiendo solo en el resultado que ofrecen, una medida de posición. Se desarrolló también una variante del algoritmo de círculos para detectar un color determinado, aunque el funcionamiento es muy similar.

Por otra parte, debemos notar que, para una mejor comprensión de los procesos que se ejecutan en el algoritmo, en este documento hemos incluido diversas imágenes mostradas por pantalla pero que en la ejecución final del programa no se muestran para no ralentizar la ejecución del programa.

Además, no hemos de olvidar la naturaleza indirecta de esta medida de posición, nuestro sensor es una cámara de la cual obtenemos imágenes y no es hasta después de un procesamiento interno de estas imágenes que se obtiene la medida de posición, esto hace que los periodos de muestreo obtenidos al aplicar estos métodos sean bastante superiores a los obtenidos con sensores directos como la IMU.

Por último, decir que se ha intentado en la medida de lo posible que este apartado sea lo más claro posible reduciendo al mínimo necesario la inclusión de fragmentos de código.

Pasemos sin más demora al desarrollo del primero de los algoritmos:

Algoritmo general

Este programa parte de las siguientes suposiciones:

- No tenemos ninguna información previa del entorno visual en el que se mueve el quadrotor
- La diferencia de posición entre dos frames es pequeña (esto se puede conseguir con una baja velocidad de movimiento del quadrotor o con una alta frecuencia de muestreo de la cámara).
- En la imagen hay elementos "característicos" de fácil seguimiento (un elemento característico es cualquier parte de la imagen que se diferencie del resto, por ejemplo los bordes de objetos, más adelante se aportarán ejemplos visuales de estos elementos)

El proceso de medición de la posición se divide en varios subprocesos encadenados que aquí exponemos en el mismo orden en el que se ejecutan en el programa diseñado inicialmente, incluimos en este punto un diagrama de flujo con la intención de proporcionar una visión global del programa como conjunto (recuperaremos esta figura con fin de clarificar los sub-procesos y modificaciones sucesivas):

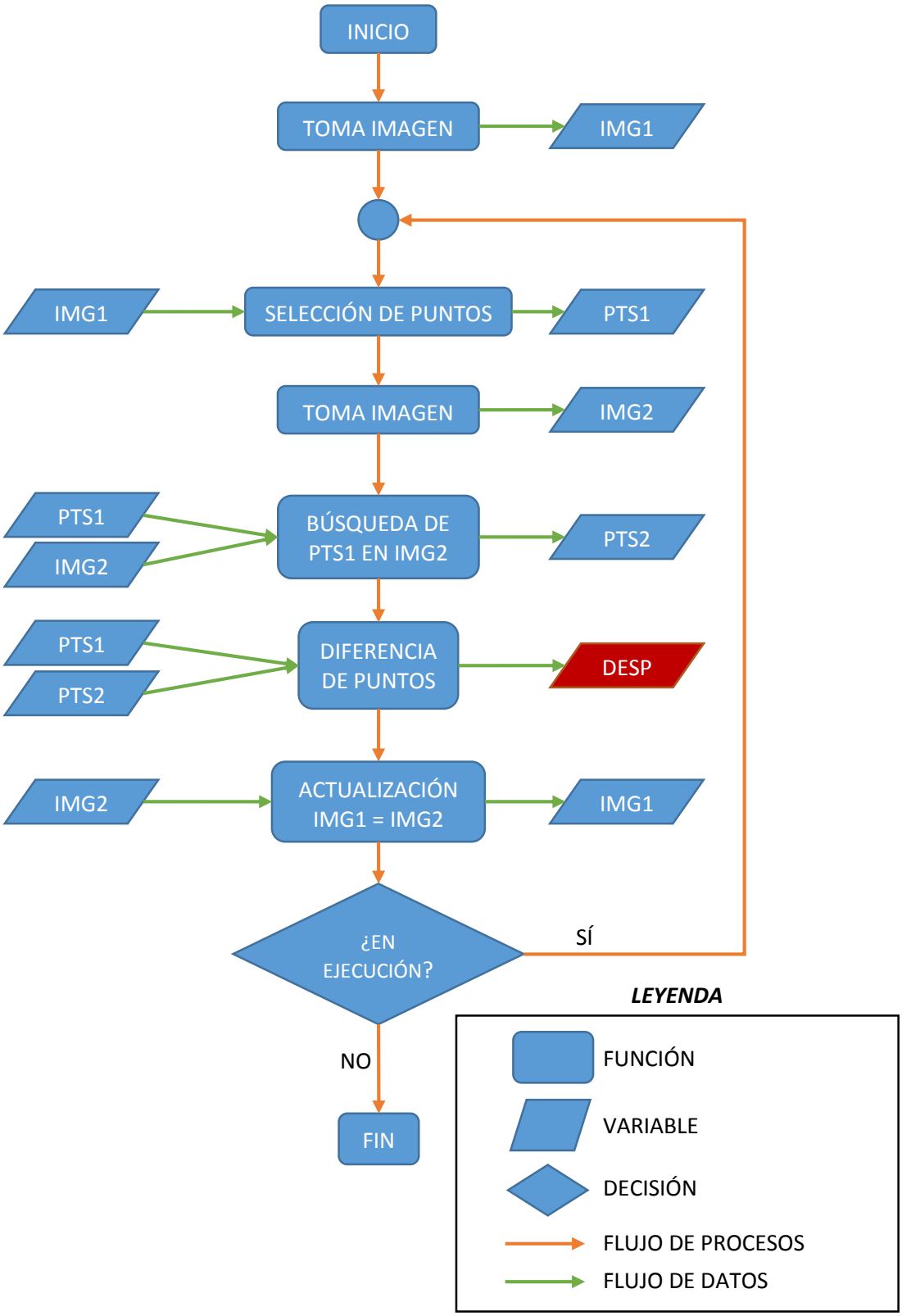

Ilustración 20. Diagrama de flujo del algoritmo general inicial

1. Selección de puntos

Considerando que tenemos la cámara y su driver correspondiente correctamente configurados, el primer paso de nuestro algoritmo es la selección de puntos. A grandes rasgos la función de selección de puntos debe tomar como entrada una imagen y devolver una serie de puntos de interés seleccionados en esa imagen:

Ilustración 21. Detalle del diagrama de flujo (selección de puntos)

Descartamos directamente la opción de seleccionar manualmente los puntos por la lentitud del proceso y porque el objetivo es automatizar el algoritmo.

Pasamos pues a considerar las diferentes opciones que nos ofrece OpenCV para la selección de puntos:

En primer lugar se probó a seleccionar puntos a intervalos regulares de píxeles de la imagen, a modo de cuadrícula, éste método más tarde probaría ser el más rápido (ya que no requiere operaciones internas de selección) pero también el más impreciso ya que los puntos seleccionados no tenían ninguna garantía de ser óptimos para el seguimiento.

Posteriormente se probaron tres funciones internas de OpenCV diseñadas específicamente para la selección de puntos: GoodFeaturesToTrack, FAST y ORB. Los fundamentos de estas tres funciones son similares, tratan de buscar, mediante un proceso iterativo, los puntos que más contrastan con su entorno, asignando a cada punto una puntuación, luego son seleccionados y almacenados los puntos con mayor puntuación. La diferencia entre ellos está en los métodos matemáticos internos que ejecuta cada función que hacen variar los resultados notablemente.

Para elegir uno de los cuatro algoritmos propuestos, tomamos como variables importantes las siguientes: el tiempo que consume la función y los puntos que no se han encontrado (puntos "perdidos") entre dos frames. Esta última variable se puede considerar un indicador de la "calidad" de los puntos encontrados, si los puntos seleccionados son buenos candidatos para el seguimiento, deberían ser fácilmente encontrados en el frame siguiente, del mismo modo, si los puntos elegidos no son los adecuados, éstos no se encontrarán.

El experimento de selección se realizó primero con la cámara en reposo y luego con un ligero movimiento hacia un lado desplazando la imagen unos 100 píxel en horizontal en un período de aproximadamente 10 segundos, es decir, a una velocidad de más o menos 10 píxel/segundo que no debería ser problema para unas funciones que se ejecutan en el orden de los milisegundos. Como condición extra se limitó la cantidad de puntos a encontrar a 50 puntos, una cantidad suficiente para efectuar el cálculo de posición sin comprometer la velocidad del procesador.

Los resultados de la comparación se muestran en el gráfico siguiente:

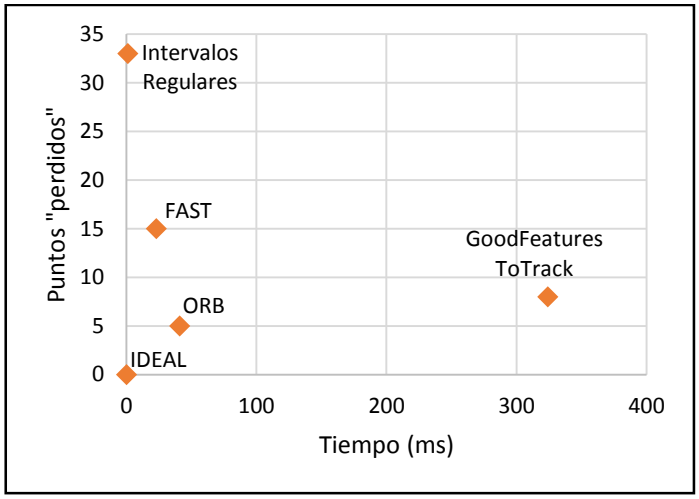

Ilustración 22. Gráfica de comparación de algoritmos de selección de puntos

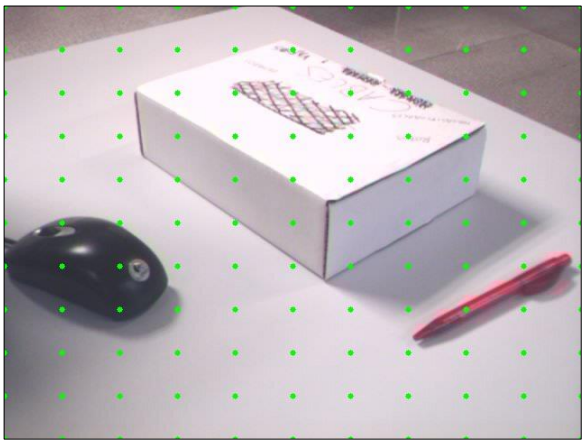

Ilustración 23. Selección de puntos a intervalos regulares

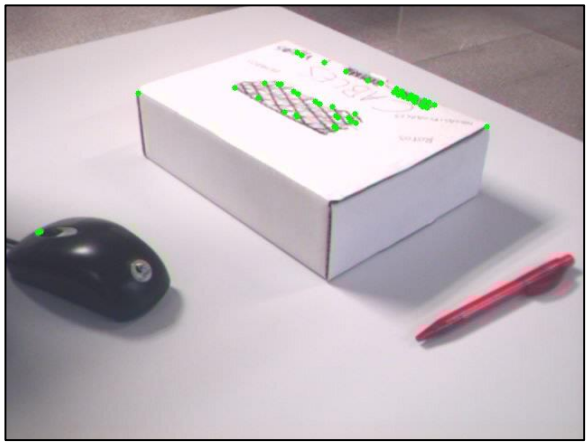

Ilustración 24. Selección de puntos con GoodFeatures

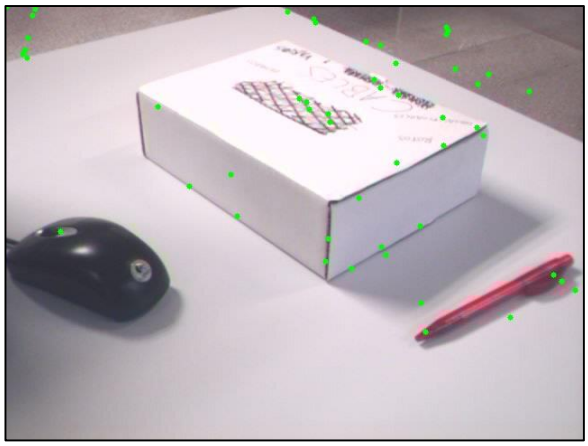

Ilustración 25. Selección de puntos con FAST

Ilustración 26. Selección de puntos con ORB

En la gráfica anterior (Ilustración 19) podemos ver que de los algoritmos expuestos, el que más se aproxima al algoritmo ideal es el ORB que ofrece la mejor relación calidad/tiempo. Es por eso que incluimos esta función en nuestro programa para la selección de puntos.

Cabe mencionar que cada una de las funciones anteriores admite una serie de parámetros relativos al proceso de selección de puntos que también fueron muestreados para obtener la máxima eficiencia y que aquí sólo se muestran los mejores resultados de cada función.

2. Seguimiento de puntos

Una vez elegidos los puntos a seguir, pasamos a buscarlos en una nueva imagen para calcular el desplazamiento que han sufrido. La función que buscamos debe funcionar del modo siguiente:

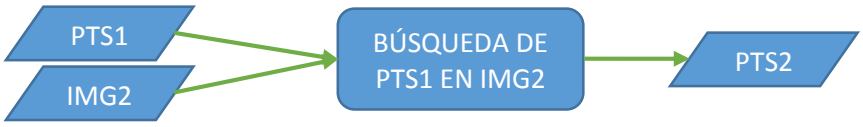

Ilustración 27. Detalle del diagrama de flujo (seguimiento de puntos)

Tomando como argumentos la imagen siguiente y los puntos encontrados en la imagen anterior, debe determinar la nueva posición de los puntos.

Consultando la documentación online de OpenCV la opción sugerida más popular es el algoritmo de flujo óptico desarrollado por Lucas-Kanade basado en pirámides (aproximaciones sucesivas reduciendo la resolución de la imagen hasta llegar a la imagen original), implementado en la función "calcOpticalFlowPyrLK". Analicemos ahora el funcionamiento de este algoritmo, como argumentos admite varios parámetros pero los más importantes son:

- Tamaño de la ventana de búsqueda: determina el ancho y alto de la ventana alrededor del punto original en la cual se va a buscar el punto desplazado, un tamaño grande mejora la precisión pero puede comprometer la velocidad de cálculo. Tomamos como solución de compromiso un tamaño de 15x15 píxel.
- Número de pirámides: cantidad de veces que se reduce la resolución de la imagen para los sucesivos cálculos del flujo óptico. Un valor de 3 mejora el rendimiento del algoritmo respecto a no aplicar reducción, valores superiores no mejoran la eficiencia e incluso llegan a reducirla, es por esto que asignamos 3 niveles de pirámides a nuestro algoritmo.
- Criterios de terminación: determinan cuando la función debe parar de buscar el flujo óptico para un punto determinado, en nuestro caso hemos dejado los valores por defecto porque funcionan correctamente, estos hacen que el algoritmo pare si el error de la solución (diferencia entre una iteración y la anterior) es menor a 0.3 píxel o si se han hecho más de 20 iteraciones.

De estos tres argumentos el más notable a nivel de rendimiento es el tamaño de la ventana, mostramos aquí dos ejemplos de cálculo del flujo óptico tomando tamaños de ventana diferentes, el flujo óptico se calcula entre la imagen mostrada en los ejemplos de selección de puntos y otra versión de esa misma imagen, desplazada un poco hacia la izquierda y hacia arriba usando un programa de edición de imágenes:

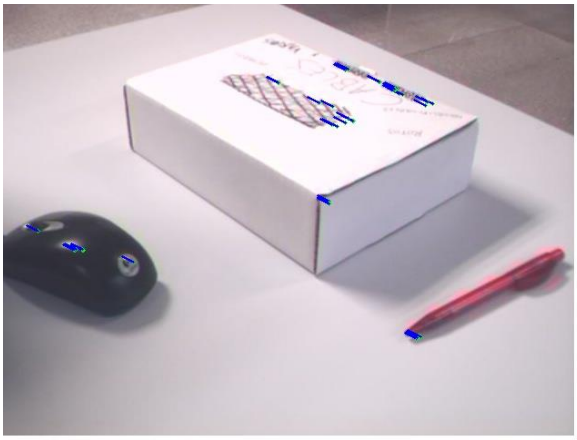

Ilustración 28. Flujo óptico con tamaño de ventana 10x10

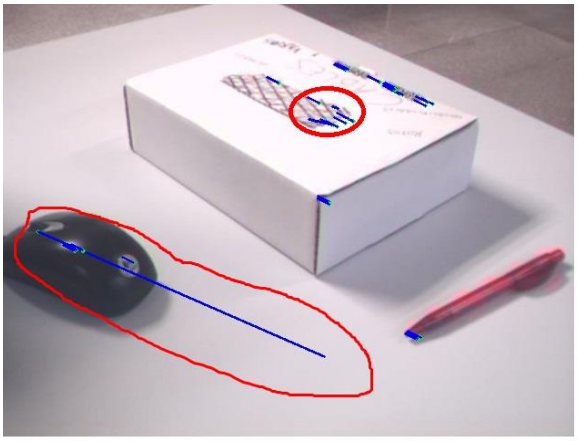

Ilustración 29. Flujo óptico con tamaño de ventana 3x3

Tal y como se aprecia en la Ilustración 26, una reducción excesiva del tamaño de la ventana de búsqueda da como resultado medidas erróneas.

Una vez calculado el flujo óptico de cada punto seleccionado, hemos de pasar de dos conjuntos de puntos, a una única medida que asignaremos como flujo óptico de la escena:

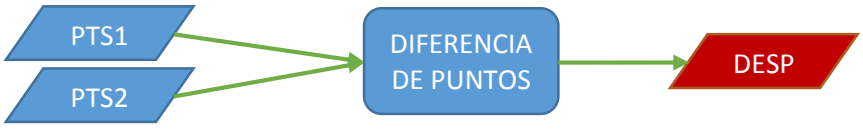

Ilustración 30. Detalle del diagrama de flujo (cálculo desplazamiento)

Lo que hacemos en este punto es tomar el vector (x_2, y_2) (coordenadas nuevas) de cada punto y restarle el vector (x_1, y_1) (coordenadas anteriores) para así encontrar el vector FO_p. Una vez obtenido el vector FO_p para cada punto seleccionado, se hace la media aritmética de estos vectores y este es el valor que se toma cómo flujo óptico de la imagen:

$$FO_{im_i} = \frac{\sum_{p=1}^{N} FO_{p_i}}{N}$$

Ecuación 5. Obtención de la medida de flujo óptico de la imagen

3. Análisis del algoritmo y mejoras

Una vez finalizada la etapa 2 del algoritmo, tenemos la medida de flujo óptico de la imagen, que es una medida indirecta del cambio de posición del quadrotor, acumulando esta medida (Ecuación 4. Definición de la posición de un píxel) lo que obtenemos es una medida de la posición relativa respecto al punto donde se inició el algoritmo.

Analizando el tiempo de ejecución total del proceso, observamos que la etapa que más tiempo consume es la de selección de puntos, así pues, con el fin de evitar el retardo excesivo del programa, efectuamos los siguientes cambios en el algoritmo:

- Realimentación: los puntos calculados por el algoritmo de seguimiento para los cuales se ha encontrado el valor de flujo óptico (no se han "perdido") son realimentados al proceso, puenteando así el proceso de selección de puntos.

- Reinicio de puntos: se establece un margen de puntos “perdidos”, si se pierden demasiados puntos, se ejecuta de nuevo la función de selección. En este caso reiniciamos los puntos cuando se ha perdido un 75% de los puntos iniciales.

Con estos cambios, el nuevo algoritmo quedaría del siguiente modo:

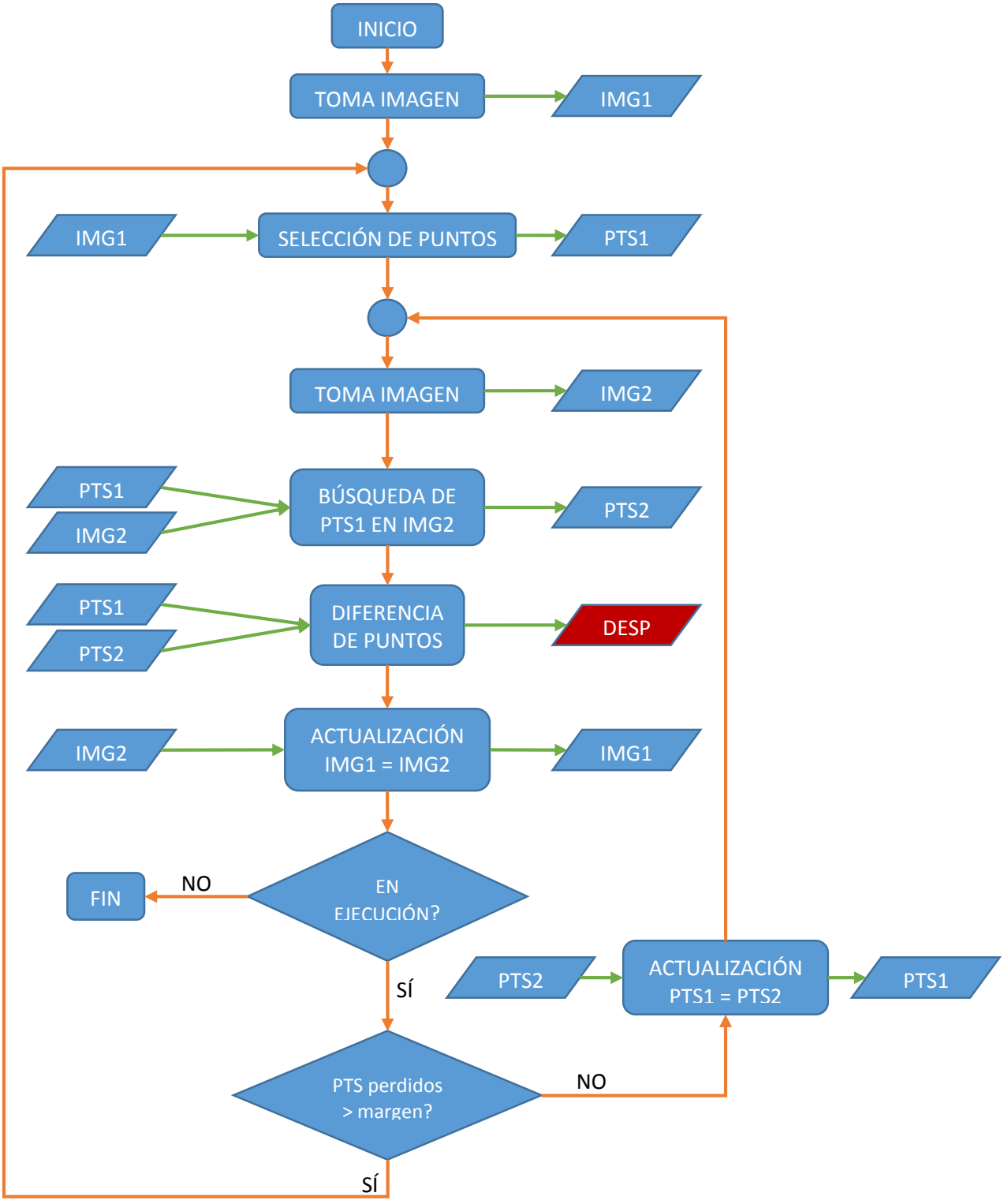

Ilustración 31. Diagrama de flujo del algoritmo general modificado

Esta sencilla modificación mejora el tiempo de ejecución medio en gran medida, mostramos aquí una gráfica que compara el tiempo de ejecución por iteración del algoritmo inicial (rojo) con el del algoritmo modificado (en esta comparación se utilizó la función de selección de puntos GoodFeaturesToTrack):

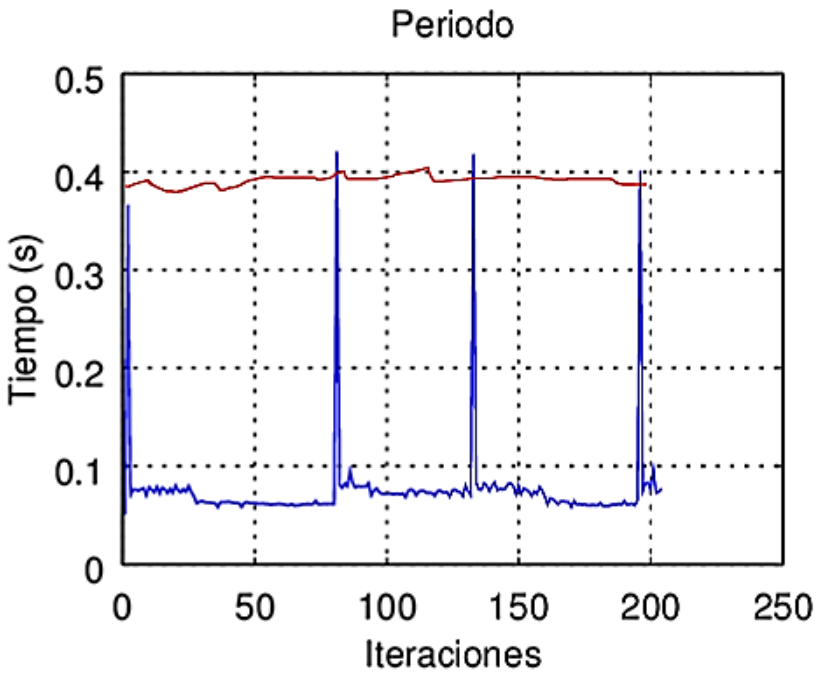

Ilustración 32. Gráfica comparativa entre algoritmos de medida

Incluimos aquí los resultados de un sencillo experimento para comprobar el correcto funcionamiento del algoritmo. El experimento consiste en arrancar el programa y luego cogiendo el quadrotor a una altura de 0.72 m, efectuar un movimiento de aproximadamente 20 cm en el sentido negativo del eje X y luego 20 cm en el sentido positivo del eje Y, para observar los resultados de forma cualitativa.

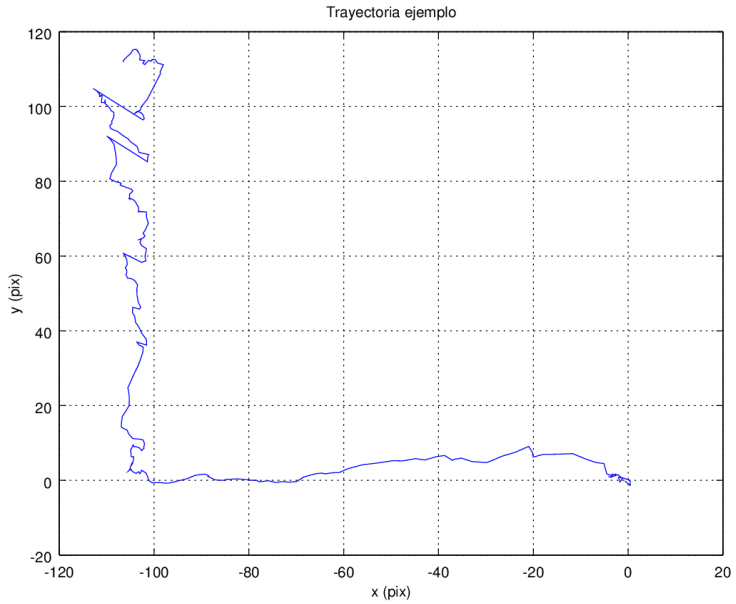

Ilustración 33. Gráfica de trayectoria ejemplo

Se puede apreciar que detecta movimiento en ambos ejes, con magnitudes similares, aunque con ruido de magnitud no despreciable tal y como se esperaba.

Algoritmo específico (círculos)

Paralelamente al desarrollo del algoritmo general, se desarrolló otro con fin de obtener una mayor precisión a costa de la necesidad de tener un control del entorno. En concreto desarrollamos un algoritmo basado en la detección de círculos mediante la transformada de Hough.

Este algoritmo es mucho más simple a nivel de programación que el anterior, exponemos aquí el diagrama de flujo que explica el funcionamiento del programa:

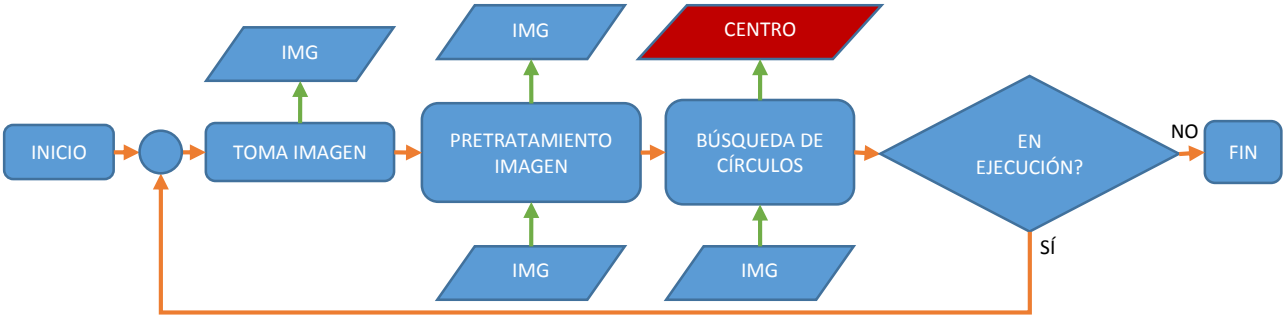

Ilustración 34. Diagrama de flujo del algoritmo específico

No entraremos tanto en detalle en este algoritmo por ser más cerrado y menos apto para modificaciones pero aun así explicaremos los aspectos generales del mismo.

En primer lugar, es notable que para la determinación de la posición en este algoritmo, es suficiente con una sola imagen, esto se debe a la naturaleza extrínseca de la referencia de posición, en este caso, un círculo impreso en papel. Así conseguimos que la medida se vea menos influida por vibraciones o ruido en la imagen.

El algoritmo consta de dos pasos:

- Primero se hace un pretratamiento de la imagen capturada para mejorar el funcionamiento de la función que aplica la transformada de Hough. Este pretratamiento consiste en un desenfoque de la imagen y una detección de bordes (algoritmo Canny)
- Posteriormente se ejecuta la función que aplica la transformada de Hough, detectando así los círculos presentes en la imagen y su centro.

De forma visual el proceso ejecutado es el siguiente:

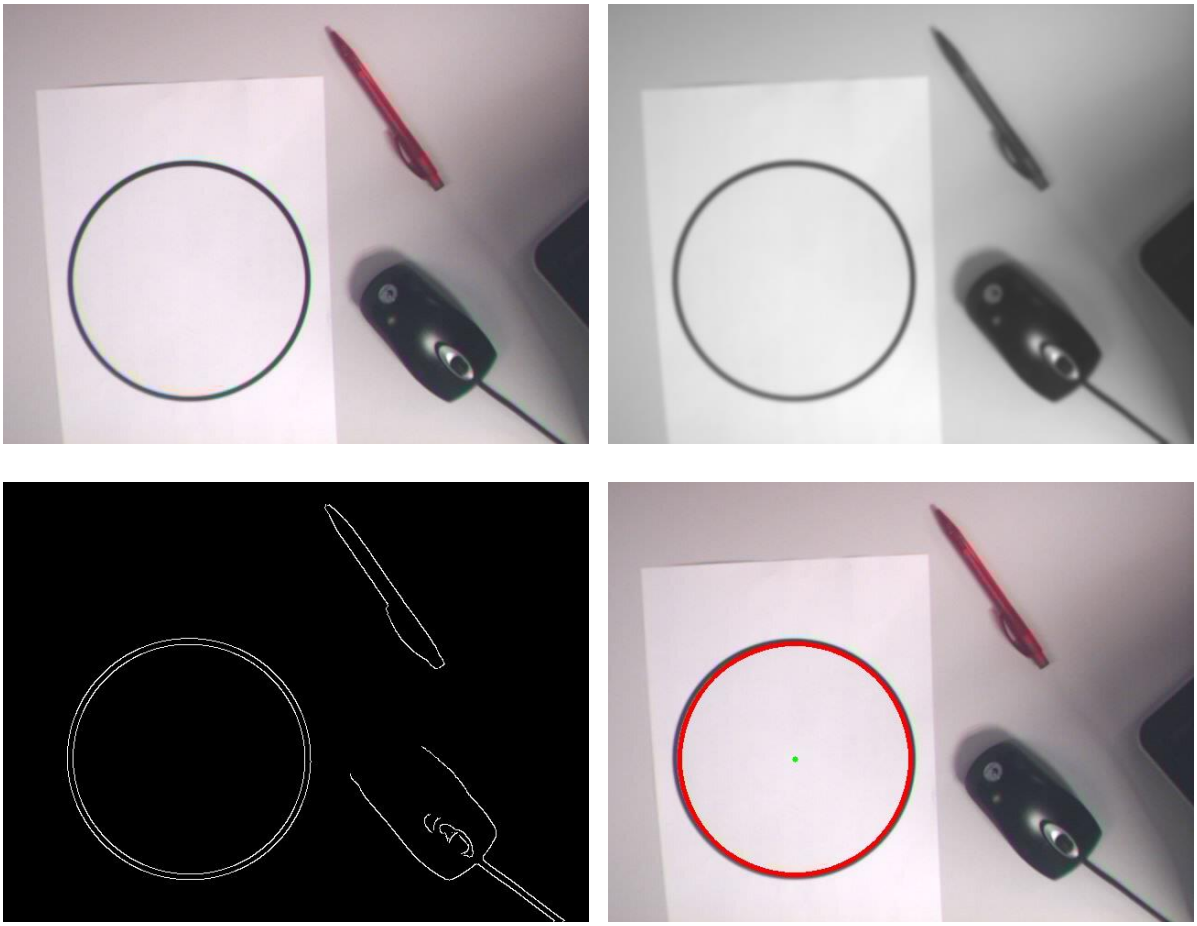

Ilustración 35. Procedimiento ejecutado por el algoritmo específico

El pequeño punto verde de la cuarta imagen indica el centro del círculo. Este centro indica la posición relativa del quadrotor respecto al círculo. Esta medida tiene un carácter más directo y con menor error que el algoritmo general, aunque es necesario tener el círculo en la escena.

Posibles aplicaciones de este algoritmo incluirían el vuelo estático sin deriva sobre el círculo (“hover”) o el seguimiento de objetos con formas características, por ejemplo automóviles.

Como modificación a este algoritmo específico se ha probado a calcular, además de la posición, el ángulo de rotación yaw, pese a que el objetivo de este trabajo es el control de posición, la obtención del ángulo yaw mediante este algoritmo es tan sencilla que se ha decidido implementarla para complementar a la medida de la IMU.

Para obtener la orientación simplemente se añade otro círculo a la escena y se calcula la inclinación de la recta que une sus centros respecto a la horizontal, este ángulo es directamente el yaw o su complementario, el funcionamiento se puede observar en la siguiente imagen.

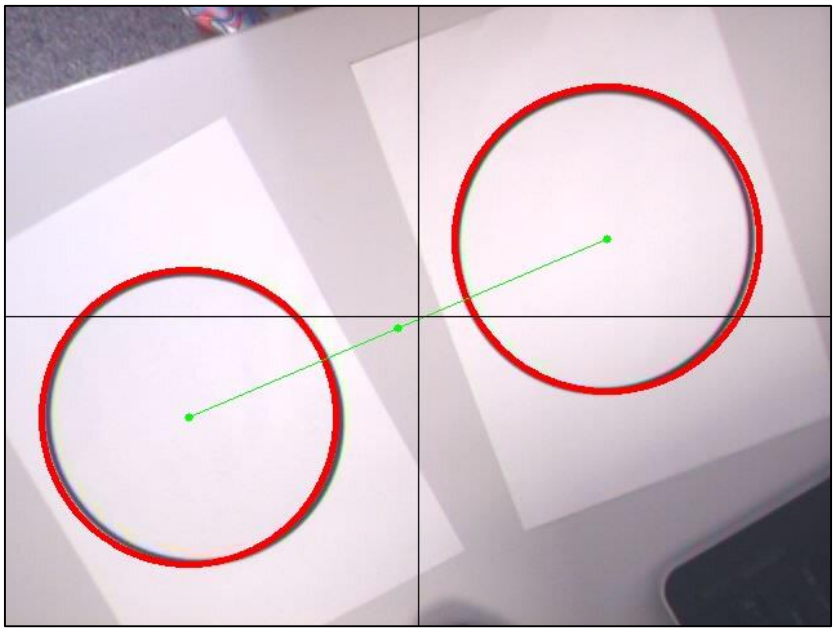

Ilustración 36. Resultado del algoritmo utilizando 2 círculos

El punto medio que une los centros sería la medida de posición x-y, y el ángulo yaw se mediría calculando la inclinación de la recta. En la Ilustración 36, por ejemplo, los resultados son:

Posición centro (píxel)	(-16,-9)
Yaw (º)	23.07

Tabla 4. Resultados algoritmo 2 círculos

Para validar la utilidad de este algoritmo se realizó un experimento igual al realizado para el algoritmo anterior los resultados se exponen aquí:

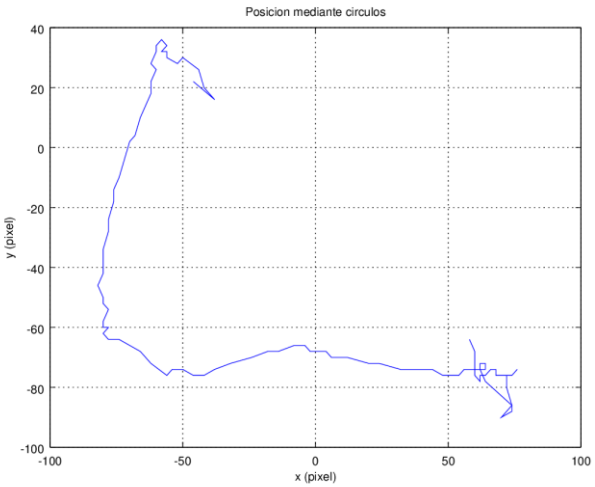

Ilustración 37. Gráfica resultado experimento círculos

Algoritmo específico (color)

Se ha diseñado también una tercera opción de medida similar al algoritmo de los círculos que elimina algunos de los problemas de éste a costa de un ligero aumento del tiempo de computación por utilizar la imagen en color en lugar de en escala de grises.

Este algoritmo funciona exactamente igual que el anterior pero en lugar de buscar un círculo, busca algún elemento de un color determinado, lo aísla y calcula su centro geométrico que se toma como medida de posición. La búsqueda de color se realiza haciendo una simple comparación de los valores de los tres canales de color (HSV) asignando un 1 a un píxel si sus colores se encuentran dentro del rango deseado, y un 0 si no se cumple esta condición.

Después a la imagen binaria obtenida, se le hace un tratamiento para eliminar el posible ruido y se le calculan los momentos de la imagen con los cuales se obtiene el centro del objeto de color.

El proceso se puede ver en las siguientes imágenes (se ha elegido el color verde para la búsqueda):

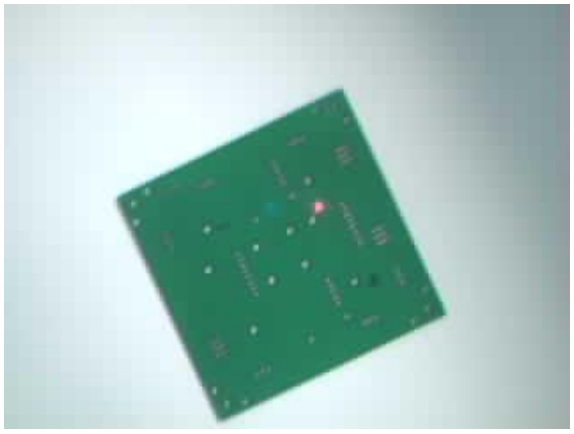

Ilustración 38. Imagen original

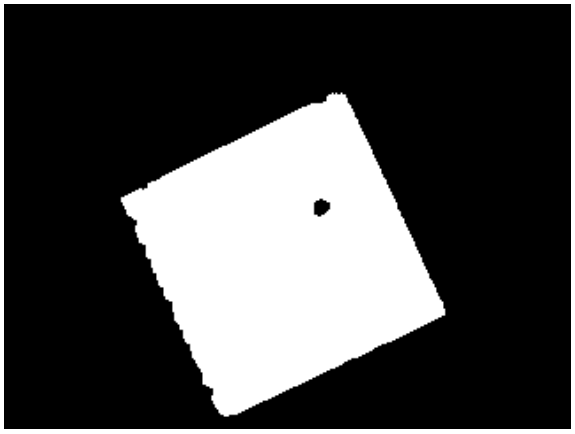

Ilustración 39. Imagen tras comparación y tratamiento

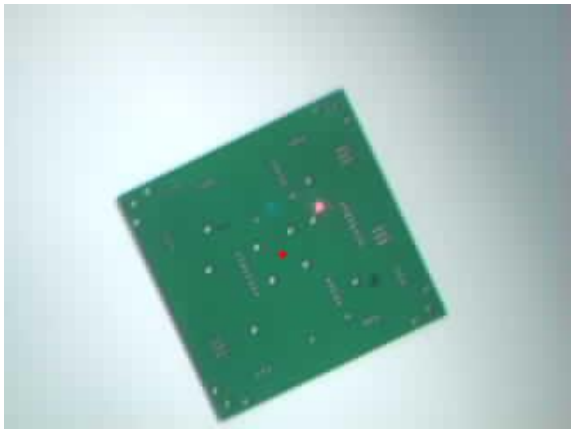

Ilustración 40. Resultado (punto rojo = centro)

2.4.2. Tratamiento de la medida

La medida obtenida con los algoritmos anteriores no se puede utilizar para hacer el control de forma directa ya que presenta ruido, interferencias provenientes de otras variables, y no está en las unidades adecuadas. Estos factores justifican la necesidad de un tratamiento de la medida.

En primer lugar, nuestra medida se encuentra en píxel, lo que buscamos es una medida del desplazamiento en metros (o centímetros), con este fin es necesario calibrar la cámara obteniendo la correspondencia entre distancias en la imagen y distancias reales. Desgraciadamente la calibración implementada en OpenCV solo permite obtener esta correspondencia para una posición estática de la cámara, las condiciones de posición y orientación de nuestra cámara están ligadas a las del quadrotor y por lo tanto son muy variables, por lo tanto la calibración de OpenCV resulta inviable así que pasamos a hacer una calibración sencilla utilizando un modelo de cámara básico.

Para calibrar la cámara tomamos el modelo "pin-hole" que considera la cámara como un punto del espacio con un ángulo de visión constante, con este ángulo y la altura de la cámara como únicos parámetros, podemos (aplicando algunas simplificaciones) calcular el factor de conversión de medidas imagen-realidad. Procedemos a exponer el experimento de calibración:

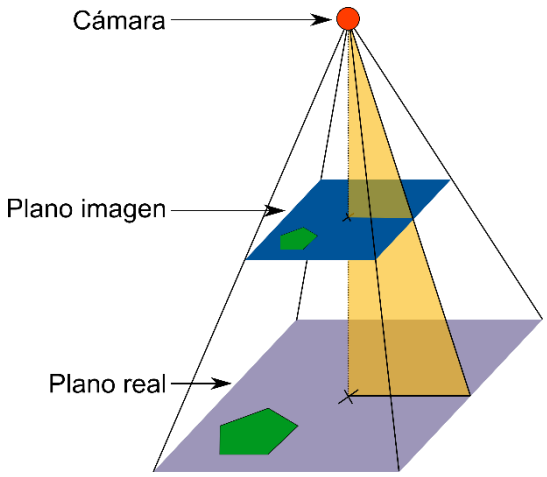

Ilustración 41. Modelo teórico de la cámara

Ilustración 42. Imagen de ejemplo para la calibración

La simplificación que consideramos para esta calibración es que la cámara está situada de forma que el plano de la imagen (ver Ilustración 41) es paralelo al suelo, esto hace que las medidas tomadas sean directamente proporcionales con las medidas reales. La simplificación es razonable siempre que los ángulos de rotación sean pequeños, esta hipótesis se ha considerado ya en la deducción del modelo teórico y se ha comprobado que es correcta.

La ejecución del experimento consiste en tomar una imagen de un objeto de longitud conocida, a una distancia también conocida y cumpliendo la simplificación expuesta anteriormente. Consideramos el triángulo naranja del modelo de la Ilustración 41, que ampliamos aquí:

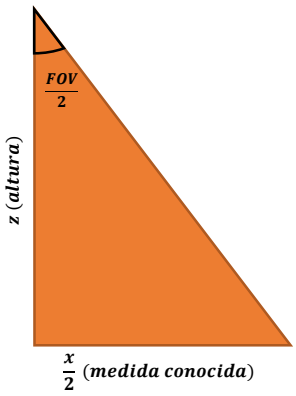

Ilustración 43. Triángulo extraído del modelo

Mediante trigonometría simple, establecemos una relación entre las 3 variables importantes:

$$\tan\frac{FOV}{2} = \frac{x}{2 * z}$$

Ecuación 6. Ecuación modelo 1

Sabiendo además que la medida x está representada en la imagen por 320 píxel podemos establecer la relación metros/píxel mediante la siguiente ecuación:

$$\frac{m}{píx} = 2 * \frac{\tan\frac{FOV}{2}}{320} * z$$

Ecuación 7. Ecuación modelo 2

Queda así establecida la relación entre metros reales y píxeles de imagen que solo depende de la altura z, por último hemos de determinar el ángulo de visión (FOV) que es una característica de la cámara. Para esto tomamos varias imágenes en las condiciones especificadas y calculamos el valor:

$$FOV = 2 * \arctan\left(\frac{x}{2 * z}\right) \approx 62.5\ º$$

Ecuación 8. Ecuación modelo 3

Hecho esto tenemos un modelo sencillo y con la precisión suficiente para aplicarlo al quadrotor.

A modo de ejemplo y validación mostramos aquí el resultado de aplicar el modelo calculado al ejemplo de trayectoria mostrado en la Ilustración 33, las características físicas del experimento fueron registradas en el momento de su ejecución con objeto de aplicar más tarde el modelo expuesto:

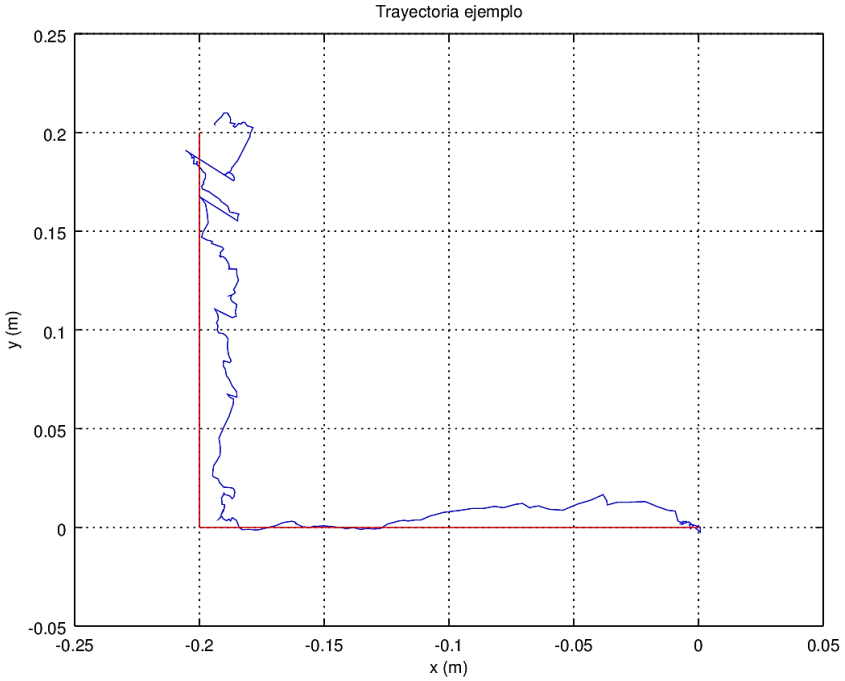

Ilustración 44. Validación del modelo con trayectoria ejemplo

En rojo aparece la trayectoria ideal y en azul la trayectoria medida y escalada, al ser el experimento realizado a una altura determinada, el modelo únicamente realiza un escalado de los ejes para ajustarlos a las dimensiones reales.

Durante la validación del modelo previamente expuesto, se detectó otro problema que afectaba a la medida: la influencia de la velocidad angular.

La cámara se puede considerar como un observador ligado al sistema físico del quadrotor, por lo tanto mide la velocidad relativa del suelo respecto a sí misma, esta medida es una medida compuesta, incluye tanto la velocidad real en x o y del quadrotor, como la velocidad correspondiente al giro.

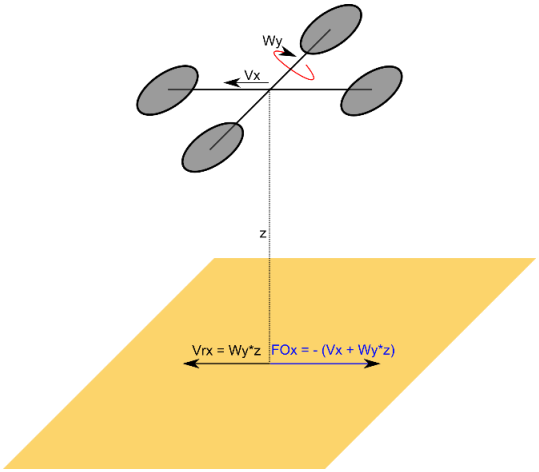

Ilustración 45. Composición de velocidades en X

$$FO_x = -(v_x + \omega_y * z)$$

Ecuación 9. Composición de velocidades en X

Para evitar este error simplemente corregimos la medida del flujo óptico con las medidas de velocidad angular y altura provenientes de otros sensores (IMU y ultrasonidos o barómetro). En un primer

momento se intentó corregir con la medida de velocidad angular que proporciona la IMU, el resultado no fue bueno y por eso se pasó a utilizar la medida de orientación (ángulo) corrigiendo sus cambios.

La extensión de esta corrección a puntos que no se encuentren inmediatamente debajo del quadrotor es directa ya que, para un giro Wy, todos los puntos que se encuentren en el plano del suelo tendrán una componente de velocidad en X del mismo módulo, y esta componente es lo que medimos con FO.

Con las correcciones aplicadas podríamos, con el algoritmo general, obtener la posición del quadrotor respecto a un origen arbitrario (punto de arranque del programa) suponiendo un ángulo yaw constante mediante la ecuación de la posición expuesta anteriormente (Ecuación 4). Esto no tiene por qué ser así ya que el ángulo yaw puede variar, así pues modificamos la ecuación de la posición para adaptarla a un ángulo yaw variable del siguiente modo:

$$POS_N = \sum_{i=1}^{N} FO(corr)_i + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \xrightarrow{\psi\ variable} POS'_N = \sum_{i=1}^{N} \begin{pmatrix} \cos\psi_i & \sin\psi_i \\ -\sin\psi_i & \cos\psi_i \end{pmatrix} * FO(corr)_i + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

Ecuación 10. Ecuación modificada de la posición (yaw variable)

En esta ecuación simplemente se ha aplicado una rotación del sistema de referencia para que el resultado sea siempre la posición en coordenadas globales.

Una vez aplicadas las correcciones descritas, podemos afirmar que la medida que obtenemos es una medida de posición absoluta en x-y del quadrotor. Ahora bien, como toda medida, presenta ruido, al cual además le hemos añadido el ruido presente en las medidas de altura y orientación al usar sus valores para las correcciones anteriores.

Es por esto que la medida requiere un filtrado previo a su uso, con este fin implementamos un filtro paso bajo para intentar rechazar el ruido mediante la siguiente ecuación:

$$x_k = z_k * \alpha + x_{k-1} * (1 - \alpha)$$

Ecuación 11. Filtro paso bajo

Donde x_k es la estimación de la posición en el instante k, z_k es la medida de posición del sensor y α es el parámetro del filtro, que puede variar entre 0 y 1, valores cercanos al 0 implican un filtrado muy fuerte, ignorando prácticamente al sensor, y valores cercanos al 1 anulan el filtrado dejando sin modificar la medida proveniente del sensor. Mostramos aquí medidas de posición y velocidad obtenidas mediante este filtrado, las medidas están en píxeles para tratar de forma independiente al filtrado y a las correcciones anteriores:

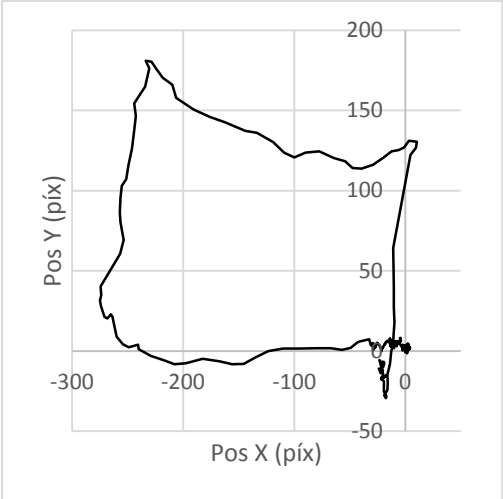

Ilustración 46. Gráfica de la posición X-Y (sin filtrar)

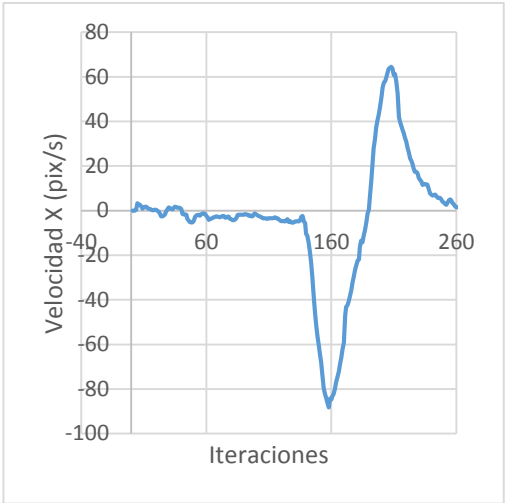

Ilustración 47. Gráfica de velocidad filtrada en X ($\alpha = 0.05$)

Hemos adaptado en nuestro programa también un filtro de Kalman diseñado en trabajos anteriores por los compañeros del laboratorio. El fundamento de este algoritmo se basa en generar un modelo de nuestro sistema que utilizamos para predecir la siguiente medida y, una vez tomada, la medida real se utiliza para corregir el modelo generado.

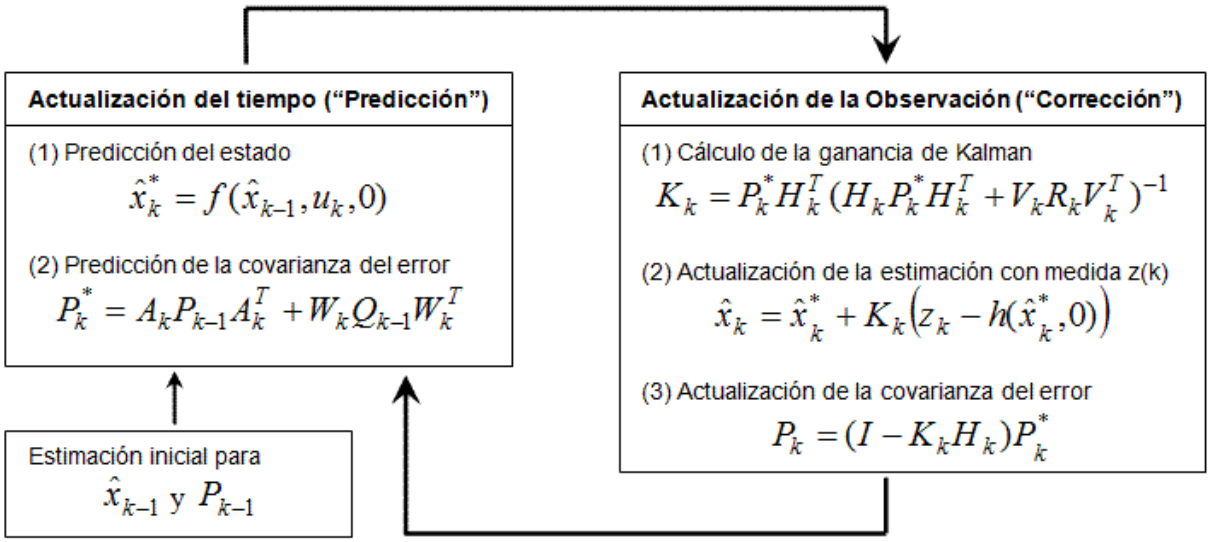

Ilustración 48. Diagrama del filtro de Kalman

En la imagen aparece el funcionamiento del filtro, los parámetros que interesan son las entradas y salidas. La entrada z_k es la medida proveniente del sensor, y la salida $\hat{x}_k$ es la estimación del estado de la variable intentando minimizar el error de medida. Las entradas Q_k y R_k son las matrices de covarianzas de los ruidos presentes en el proceso y en el sensor respectivamente, y determinan el peso que se le dará a las diferentes variables y mediciones en el modelo creado, a mayor covarianza, menor peso.

2.4.3. Desarrollo del algoritmo de control

Una vez tomada la medida de posición, se pasa al desarrollo del control, para poder controlar la posición, acudimos al modelo teórico (Ecuación 1) y observamos que no tenemos un control directo de la posición (o de la aceleración) sino que hemos de diseñarlo pasando por el control de orientación.

Explicaremos el algoritmo de control para el eje X y el ángulo pitch pero la extensión al eje Y y el ángulo roll sería inmediata (si consideramos ángulos pequeños). Partimos del control de la orientación, que tiene la siguiente forma:

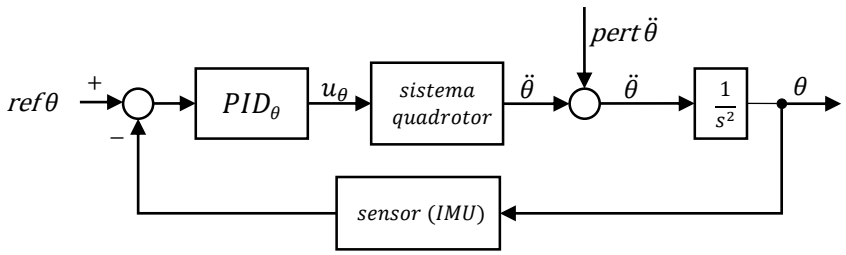

Ilustración 49. Diagrama de flujo del control de orientación

Donde ref θ es la referencia de pitch, u_θ es la acción aplicada (tal y como aparece en el modelo teórico) y $\ddot{\theta}$ es la respuesta dinámica del sistema, a la cual añadimos posibles perturbaciones no presentes en el modelo teórico (viento, choques,…) y que luego es captada (tras un doble integrador) por el sensor y retroalimentada al sistema de control.

El control que proponemos para la posición X es el siguiente:

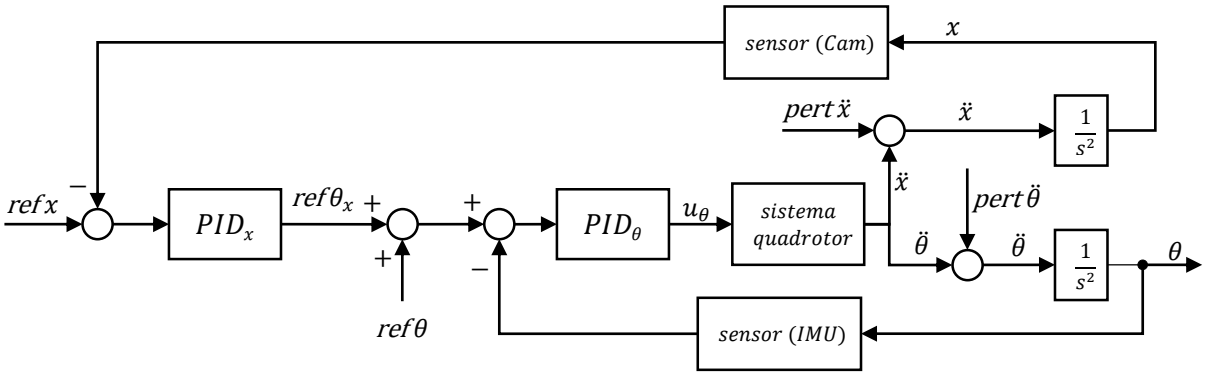

Ilustración 50. Diagrama de flujo del control de la posición en X

Como se observa en el diagrama, se mantiene el control de orientación original al cual se superpone un control de posición a través de la variable $ref\,\theta_x$ que hace la función de "actuación" indirecta sobre el sistema quadrotor.

En las pruebas realizadas con el control de posición, los resultados obtenidos no fueron buenos, el control era inestable, y el quadrotor se movía de forma errática, reaccionando a las medidas de posición pero sin lograr controlar estas variables, se llegó a la conclusión de que un control de posición actuaba con demasiado retardo.

También se observó que el quadrotor empezaba a tener sustentación suficiente para volar con los motores a un 75% de su potencia máxima, esto hacía que el control no fuese fiable porque se llegaba

al límite de potencia al sumar la acción de control, así que se cambiaron los motores por otros que permitían sustentar el quadrotor a sólo un 60% de potencia, dando margen suficiente para la acción de control.

En base a los resultados del control de posición, se decidió pasar a un control de velocidad utilizando como medida el flujo óptico dividido entre el periodo entre imágenes. Se ha fusionado la medida de velocidad proveniente de la cámara con la velocidad integrada de la aceleración de la IMU en un filtro de Kalman.

Mostramos aquí una medida tomada en vuelo con el algoritmo de control de la velocidad, en azul se muestra la velocidad en el eje X, y en rojo, la referencia de pitch enviada al control de orientación:

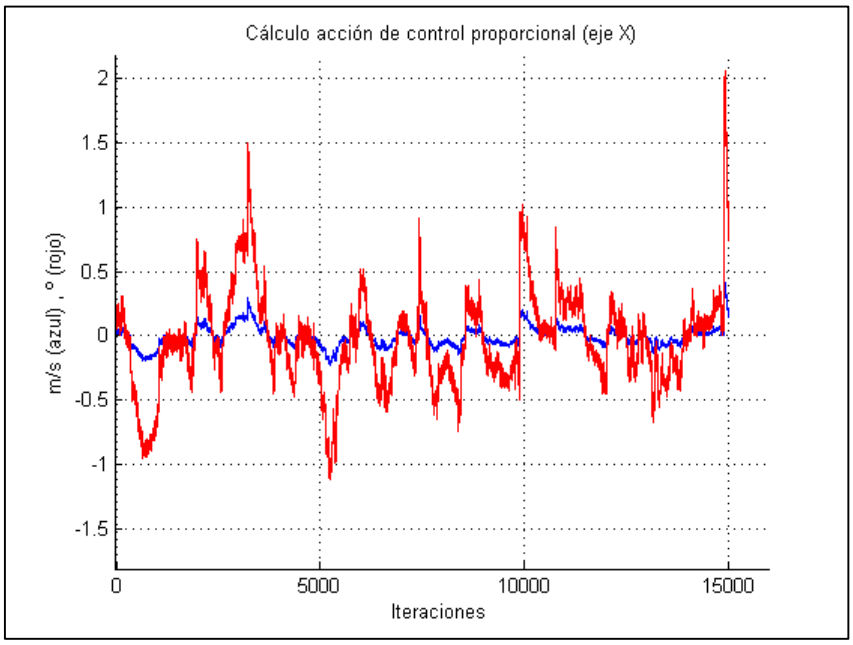

Ilustración 51. Resultados de la acción proporcional en vuelo

En el vuelo mostrado en la gráfica sólo se aplicó un control proporcional para comprobar el correcto comportamiento y respuesta a la medida. Los resultados son satisfactorios ya que se consigue mantener la velocidad en X del quadrotor en un rango de ±0.3 m/s.

La acción derivativa se implementó más tarde pues la medida de aceleración de la IMU presentaba mucho ruido, mostramos aquí el resultado del filtro paso bajo aplicado a la aceleración:

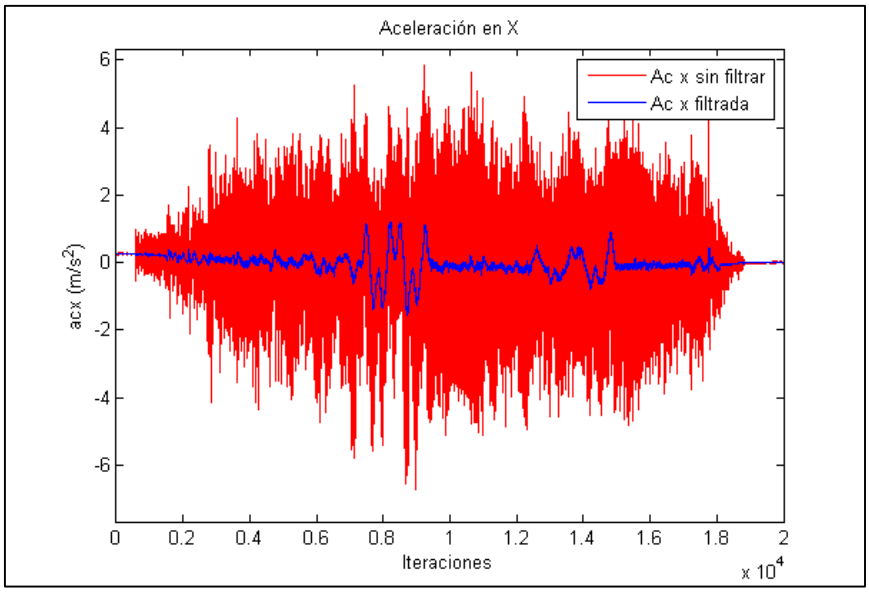

Ilustración 52. Filtrado de la aceleración

Aun así, tras el filtrado, la acción derivativa desestabilizaba el quadrotor y se intentó implementar un control PI, con antiwindup, con la finalidad de evitar la deriva a velocidades lentas. Que tampoco funcionó por la misma razón, creaba inestabilidad en el quadrotor.

Se observó durante estas pruebas también un grave retardo (ver gráfica inferior) en la medida obtenida de la cámara a intervalos regulares (partes planas de la gráfica azul). Esto dificultaba gravemente el control por dejar de tener medida por periodos de 2-3 segundos, lo cual distorsionaba mucho el modelo elaborado por el filtro de Kalman (gráfica roja). Una vez solucionado este cuello de botella relacionado con la escritura en ficheros se procedió a volver a intentar realizar el control.

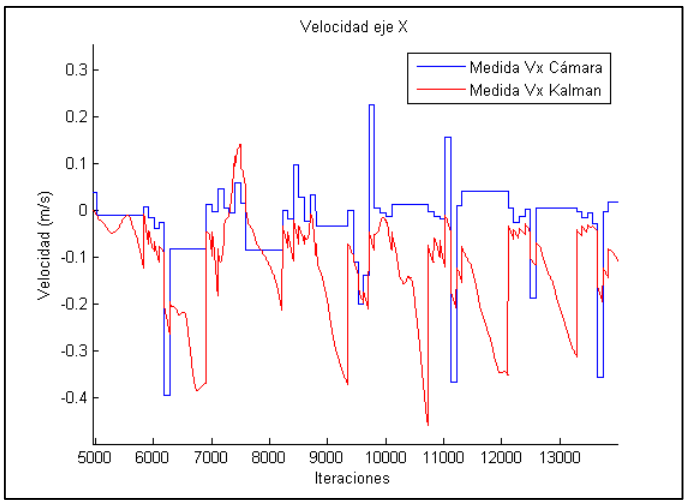

Ilustración 53. Retardo en la medida de visión

Los resultados del control aplicado finalmente se muestran en el apartado siguiente.

3. Resultados y conclusiones

Los resultados obtenidos son los siguientes:

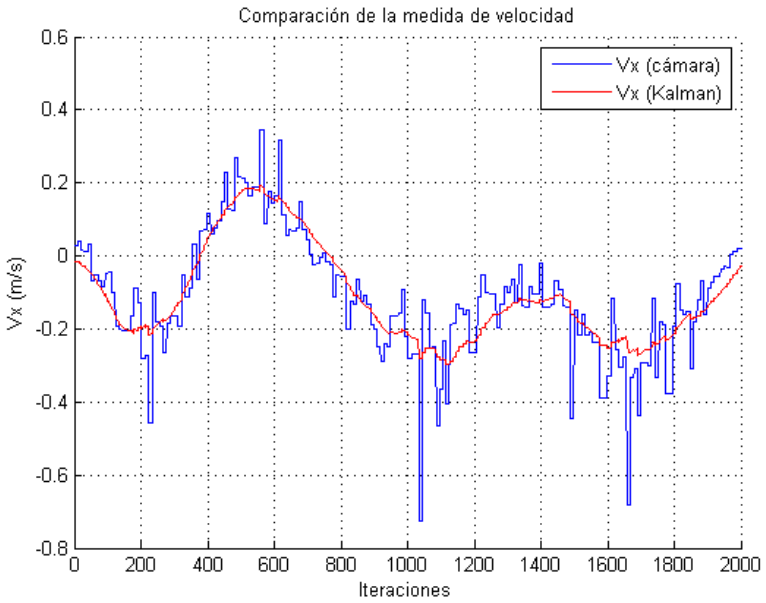

Ilustración 54. Medida final de velocidad

La gráfica es el resultado de un control de velocidad proporcional pero sin retardo en la medida de visión, las acciones proporcional y derivativa no se pudieron probar por falta de tiempo. Como se ve, el sistema tiende a la velocidad 0 aunque con oscilaciones por la simplicidad del control proporcional.

Los experimentos verifican que el sistema funciona aceptablemente en interiores y que podría ser aplicado como mecanismo de emergencia (en caso de fallo de la señal gps) en exteriores para evitar la pérdida del quadrotor

Tras la realización de este trabajo, concluimos que la visión artificial es una herramienta potente y polivalente tal y como hemos visto puesto que permite calcular posiciones, velocidades e incluso la orientación del quadrotor. Pero esta polivalencia tiene un precio, que pagamos en las dificultades de obtener una medida precisa y aislada, debido a los parámetros de resolución, velocidad y ruido de la cámara y a las múltiples interacciones con otras variables causadas por la naturaleza del sistema de visión empleado.

Mencionamos aquí algunas de las potenciales mejoras que se podrían aplicar a este sistema, las razones de que no se hayan aplicado en la ejecución del trabajo son principalmente la limitación de recursos y tiempo:

- Inclusión de una segunda cámara (estereovisión) para dejar de depender de sensores externos.
- Utilizar una cámara (o más) externa al quadrotor, eliminando así el carácter relativo de la medida.
- Para el sistema con la cámara integrada en el quadrotor, añadir un sistema de estabilización dinámica para evitar interacciones con la velocidad angular.

Bibliografía

[1] *Víctor Manuel Peñacoba Hornillos. (2013). Plataforma de desarrollo para la configuración del control empotrado en helicópteros cuatri-rotor. Valencia: UPV.*

[2] *Gary Bradski & Adrian Kaehler. (2011). Learning OpenCV. Sebastopol: O'reilly.*

[3] OpenCV documentation. (n.d.). Obtenido de: http://docs.opencv.org/

2015

Presupuesto

El presupuesto que procedemos a detallar, considera sólo el coste de diseño e instalación del sistema de control por visión en un quadrotor operativo, quedando fuera de éste los costes de construcción y puesta a punto del quadrotor. Básicamente se supone que se tiene ya la estructura soporte y los motores con sus drivers correspondientes y funcionales.

Se dividirá el trabajo realizado en unidades de obra semi-independientes, de modo que si ya se contase con alguna de ellas de manera previa, ésta sería deducible del importe total.

Cabe mencionar también que no se han contabilizado aquí las horas necesarias para el aprendizaje de los procedimientos a realizar en cada una de las unidades de obra.

Unidades de obra

UD 0. Preparación de la plataforma base				
	Descripción	Rendimiento	Precio	Importe
h	Graduado en tecnologías industriales	50	25	1250
ud	Placa IGEPv2	1	179	179
ud	Arduino DUE	1	38.08	38.08
ud	IMU 10DOF v2 Drotek	1	18.9	18.9
ud	PING Ultrasonic Distance Sensor (Parallax)	1	26.69	26.69
			Coste directo	1512.67
			Costes directos complementarios (2%)	30.25
			Costes indirecto (2%)	30.86
			Coste total	1,573.78 €

UD 1. Instalación de la cámara en el quadrotor				
	Descripción	Rendimiento	Precio	Importe
ud	Cámara PlayStation® Eye	1	40	40
h	Graduado en tecnologías industriales	0.5	25	12.5
			Coste directo	52.5
			Costes directos complementarios (1%)	0.53
			Coste indirecto (0%)	0
			Coste total	53.03 €

UD 2. Diseño de los algoritmos de medición, calibración e integración en el software del quadrotor.				
	Descripción	Rendimiento	Precio	Importe
h	Graduado en tecnologías industriales	40	25	1000
ud	Flexómetro (5 m)	1	2.5	2.5
			Coste directo	1002.5
			Costes directos complementarios (0.5%)	5.01
			Coste indirecto (2%)	20.15
			Coste total	1027.66 €

UD 3. Validación de las mediciones y ajuste del control				
	Descripción	Rendimiento	Precio	Importe
h	Graduado en tecnologías industriales	15	25	375
ud	Fuente de alimentación Maas SPS-9250 40 A	1	129	129
			Coste directo	504
			Costes directos complementarios (1%)	5.04
			Coste indirecto (2%)	10.18
			Coste total	519.22 €

Presupuesto final

Id.	Unidad de obra	Importe
UD 0	Preparación de la plataforma base	1,573.78 €
UD 1	Instalación de la cámara en el quadrotor	53.03 €
UD 2	Diseño de los algoritmos de medición, calibración e integración en el software del quadrotor.	1027.66 €
UD 3	Validación de las mediciones y ajuste del control	519.22 €
Presupuesto de ejecución material (PEM)		**3173.69 €**
Gastos generales (12%)		380.84 €
Beneficio industrial (6%)		190.42 €
Presupuesto de ejecución por contrata		**3744.95 €**
IVA (21%)		786.44 €
Presupuesto base de licitación		**4531.39 €**

El presente presupuesto asciende a la cantidad de: CUATRO MIL QUINIENTOS TREINTA Y UN EUROS CON TREINTA Y NUEVE CÉNTIMOS.

2015

Anexo de programación

El propósito de este anexo es mostrar el código desarrollado para garantizar la reproducibilidad del proyecto, se añaden además anotaciones sobre las funciones utilizadas y sus parámetros importantes para facilitar la comprensión y la implementación de posibles mejoras.

El código ha sido escrito en lenguaje C++ para su posterior integración en el programa que controla el resto del quadrotor, que también está escrito en este lenguaje.

En esta explicación se omiten las declaraciones de variables a no ser que se trate de variables específicas de OpenCV.

Código del algoritmo general de medición

Inicialización del programa:

```
Videocapture cam;       //Esta clase de variable constituye un puntero a un
                        //dispositivo de captura (cámara)

Mat imagen1;            //La clase de variable Mat es muy utilizada en OpenCV
                        //por su versatilidad, contiene matrices, pero
                        //puede contener imágenes en varios formatos

//Se establece la conexión con la cámara y la resolución de captura, también
//se capta la primera imagen y se transforma a escala de grises.

if(cam.open(0) == 0){
         printf("Error en la apertura de la camara.\n");
         return -1;
    }

    cam.set(CV_CAP_PROP_FRAME_WIDTH,320);
    cam.set(CV_CAP_PROP_FRAME_HEIGHT,240);

    cam >> imagen1;

    gettimeofday(&tvs, NULL);       //aquí almacenamos el instante en el que
                                    //se captura la imagen

    cvtColor(imagen1,imagen1, CV_BGR2GRAY);
```

Bucle principal:

Para mayor claridad, los bucles anidados o condicionales se han tabulado de acuerdo a su nivel de "anidación"

```
While(1){

cam >> imagen2;              //tomamos la segunda imagen
```

```
gettimeofday(&tve, NULL);     //calculamos la diferencia de tiempos entre
timersub(&tve, &tvs, &tvd);   //capturas, esta parte se puede ejecutar de
gettimeofday(&tvs, NULL);     //forma cíclica
periodo=((float)tvd.tv_sec+1e-6*(float)tvd.tv_usec);

cvtColor(imagen2,imagen2,CV_BGR2GRAY);//pasamos la imagen a escala de grises

//dentro de este "if" se seleccionan los puntos de interés, "reseteador" es
//una variable que se utiliza para decidir si se deben buscar puntos o no

if (reseteador==1){     //reseteador = 1 indica que se deben buscar puntos
      corners1.clear(); //se limpian los puntos anteriores

      //se buscan los puntos y se convierten a un fomato adecuado para su
      //uso (la función ORB devuelve más datos de los necesarios)

      OrbFeatureDetector det(100,1.2f,8,31,0,2,0,31);
      det.detect(imagen1,keypoints)
      for (i=0;i<int(keypoints.size());i++){
            corners1.push_back(keypoints[i].pt);
      }

}

if (corners1.size()>0){
      reseteador = 0;   //si se han encontrado puntos (el vector corners1 no
                        //está vacío), se indica mediante reseteador que no
                        //hace falta buscar puntos

      //Función de cálculo del flujo óptico
      calcOpticalFlowPyrLK(imagen1,imagen2,    //imágenes previa y actual
      corners1,corners2,      //vectores de puntos previos y actuales,
                              //corners2 es la salida de esta función
      status,err,             //status indica los puntos encontrados
                              //err indica el error de posición
      Size(15,15),3,          //Size(x,y) indica el tamaño de la ventana
                              //de búsqueda y el 3 es el número de pirámides
      //criterios de terminación: 20 iteraciones o error inferior a 0.3
      TermCriteria(TermCriteria::COUNT+TermCriteria::EPS,20,0.3));




      ofx=0;            //inicializamos variables de desplazamiento a 0
```

```
  ofy=0;
  nb_depl=0;

  //este bucle calcula la media de los desplazamientos de los píxel

    for(i=0;i<int(corners1.size());i++)
    {
        if(status[i]!=0)
        {
            ofx = ofx + (corners2[i].x-corners1[i].x);
            ofy = ofy + (corners2[i].y-corners1[i].y);
            nb_depl++;
        }
    }
  //aquí se comprueba si no se han perdido todo los puntos

    if (nb_depl>0){
    ofx=ofx/nb_depl;
    ofy=ofy/nb_depl;
    }

  //calculamos la velocidad (derivada de la posición) filtrada para
  //la acción derivativa del control

  velx = (1-0.05)*velx + 0.05*(ofx/periodo);
  vely = (1-0.05)*vely + 0.05*(ofy/periodo);


  //condición de reset, que se pierda más de un 75% de los puntos

if(float(nb_depl)/float(corners1.size())<0.25)
reseteador=1;

  //actualización de imágenes y vectores

imagen2.copyTo(imagen1);
std::swap(corners2, corners1);

  //acumulación del desplazamiento para obtener posición

suma[0] += ofx;
suma[1] += ofy;

  //impresión a fichero
```

```
    fflush(fichflujo);
    fprintf(fichflujo,"%.6f\t %+.5f\t%+.5f\n",periodo,suma[0],suma[1]);
}


else
imagen2.copyTo(imagen1);      //en caso que no se encuentren puntos,
                              //simplemente se actualiza la imagen
}
```

Código del algoritmo específico de medición (algoritmo de círculos)

Este código está basado en el código de ejemplo presente en la página web:

http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/hough_circle/hough_circle.html

Inicialización:

```
VideoCapture cam(0);
cam.set(CV_CAP_PROP_FRAME_HEIGHT,320);
cam.set(CV_CAP_PROP_FRAME_WIDTH,240);
```

Bucle principal:

```
while(1){

//toma de imagen y tratamiento de la misma (escala de grises y desenfoque)
cam >> src;
cvtColor( src, src_gray, CV_BGR2GRAY );
GaussianBlur( src_gray, src_gray, Size(9, 9), 2, 2 );

//aplicación de la transformada de Hough para encontrar círculos

HoughCircles( src_gray, //imagen
circles,                //vector de centros y radios
CV_HOUGH_GRADIENT,      //método utilizado para la detección
1,                      //proporción inversa de resolución
src_gray.rows/8,        //distancia mínima entre centros
200,                    //margen para detección de círculos
100,                    //margen para el filtro canny (detección de bordes)
0,0);                   //radios minimo y máximo de los círculos

   //adaptación de los datos al formato de uso
   if(int(circles.size())>0){
      for(i=0;i<int(circles.size());i++){
            Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
            centros[i]=center;
      }
      //si se encuentran 2 círculos, calcular la inclinación y el centro
```

```
    if (circles.size()==2){
          float yaw;
          yaw=atan(float(centros[1].y-centros[0].y)/float(centros[1].x-
          centros[0].x));
          fflush(fichflujo);
          fprintf(fichflujo,"%f %f %f\n",
          (centros[1].x+centros[0].x)*0.5-160,
          -(centros[1].y+centros[0].y)*0.5+120,
          yaw*180.0/PI);
     }

     //si se encuentra solo un círculo, calcular la posición de este
     if (circles.size()==1){
         fflush(fichflujo);
         fprintf(fichflujo,"%d %d\n",centros[0].x-160,-centros[0].y+120);
     }

  }
```

Estos algoritmos han sido implementados como threads que se lanzan desde un objeto dedicado para adecuarse a la estructura del programa principal del quadrotor.

Código del algoritmo específico de medición (algoritmo de color)

Este código está basado en el código de ejemplo presente en la página web:

http://opencv-srf.blogspot.com.es/2010/09/object-detection-using-color-seperation.html

Por ser muy similar al algoritmo de círculos se explicarán solo aquellas funciones que no hayan sido explicadas ya.

Inicialización:

```
VideoCapture cam(0);
cam.set(CV_CAP_PROP_FRAME_HEIGHT,320);
cam.set(CV_CAP_PROP_FRAME_WIDTH,240);
```

Bucle principal:

```
while(1){

    cam >> src;
    gettimeofday(&tve, NULL); timersub(&tve, &tvs, &tvd);
    gettimeofday(&tvs, NULL);

    mediaper=((float)tvd.tv_sec+1e-6*(float)tvd.tv_usec);

    cvtColor(src, src, COLOR_BGR2HSV); //pasamos la imagen a HSV
      inRange(src,                              //imagen origen
      Scalar(iLowH, iLowS, iLowV),              //límite inferior del rango
```

```
      Scalar(iHighH, iHighS, iHighV),          //límite superior del rango
      src);                                    //imagen destino (binaria)

      //tratamiento de la imagen (eliminar ruido)
    erode(src, src, getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );
    dilate( src, src, getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );
    dilate( src, src, getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );
    erode(src, src, getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );

      //cálculo de momentos
    Moments oMoments = moments(src);
    double dM01 = oMoments.m01;
    double dM10 = oMoments.m10;
    double dArea = oMoments.m00;

      //si se encuentra algún elemento de área suficiente, calcular su centro
    if (dArea > 100)
    {
        posX = -(dM10 / dArea)+160;
        posY = (dM01 / dArea)-120;
    }
}
```

Estos algoritmos han sido implementados como threads que se lanzan desde un objeto dedicado para adecuarse a la estructura del programa principal del quadrotor.

Modificaciones para adaptar la medida

```
//adaptación de píxel a metros (coefficient = 0.00379259)
MeasureInMetres = MeasureInPixels*coefficient*altura;

//corrección para rechazar medidas falsas por giros
Measure = Measure – sin(Angle)*altura;
```

Filtrado

```
//implementación del filtro paso bajo
MeasureFiltered = alfa*Measure + (1-alfa)*PreviousMeasure;
```

No se incluye el código del filtro de Kalman porque ha sido desarrollado principalmente por Alberto Castillo, compañero del laboratorio, y no procede anexarlo aquí. Solo mencionar que este filtro ejecuta una fusión de datos combinando las medidas de la cámara y la IMU, consiguiendo así una medida más fiable y precisa.

Control

El control PID, ya sea de posición o velocidad se ha implementado del siguiente modo:

```
accPx = kPx * (refx – x);                //proporcional x
accDx = kDx * (-dx);                     //derivativa x
accIx = accIx + kIx * (refx – x);        //integral x

refPitchX = accP + accD + accI;          //acción total x

accPy = kPy * (refy – y);                //proporcional y
accDy = kDy * (-dy);                     //derivativa y
accIy = accIy + kIy * (refy – y);        //integral y

refRollY = accPy + accDy + accIy;        //acción total y

refPitchX = saturacion(refPitchX,-5,5);  //saturamos las acciones tanto en
refRollY = saturacion(refRollY,-5,5);    //x como en y para evitar peligro
```