

# Trabajo de Fin de Master

## Ingeniería Electrónica, Robótica y Automática

Posicionamiento de un UAV Mediante los  
Marcadores Visuales *Aruco* y el Estimador  
de Estados de *PX4*

Autor: Isidro Jesús Arias Sánchez

Tutor: Manuel Vargas Villanueva

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

Sevilla, 2020





Trabajo de Fin de Master  
Ingeniería Electrónica, Robótica y Automática

# **Posicionamiento de un UAV Mediante los Marcadores Visuales Aruco y el Estimador de Estados de *PX4***

Autor:  
Isidro Jesús Arias Sánchez

Tutor:  
Manuel Vargas Villanueva  
Profesor Titular

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

Sevilla, 2020



Trabajo de Fin de Master: Posicionamiento de un UAV Mediante los Marcadores Visuales Aruco y el Estimador de Estados de *PX4*

Autor: Isidro Jesús Arias Sánchez  
Tutor: Manuel Vargas Villanueva

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:



A mi abuela Mercedes



# Resumen

---

**E**n este trabajo se busca conseguir el posicionamiento preciso de un quadrotor intentando no usar componentes demasiado costosos. Para ello, se han usado unos marcadores impresos y se ha dotado al vehículo de una cámara. Del proceso seguido hasta conseguirlo, se detallará tanto su programación como los componentes físicos elegidos.

Por otra parte, se analiza una de las mejoras a un estimador de estados convencional que se le aplican en la práctica, tomando como ejemplo el autopiloto de código abierto *PX4*. Una mejora que resuelve la duda de cómo afrontar el retraso que tienen algunos sensores, que se considera nulo en muchos estudios, pero que pueden impactar negativamente en el desempeño si no se tratan adecuadamente. Para demostrar esta idea, se elabora una simulación que pone de manifiesto su utilidad.



# Abstract

---

The aim of this work is to achieve the precise positioning of a quadrotor attempting not to use expensive components. To that end, printed markers has been used and the vehicle has been provided with a camera. Both programming and the chosen physical components will be detailed.

Moreover, it is analyzed an improvement over standard state estimators, that is carried out in practice, specifically in the open source autopilot *PX4*. This enhancement handle the delay that some sensors have, which is not considered in many studies, but they could have a negative impact in the performance if they are not faced appropriately. This idea is tested in a simulation that illustrate his usefulness.



# Índice

---

<i>Resumen</i>	III
<i>Abstract</i>	V
<i>Notación</i>	IX
<i>Acrónimos</i>	XI
<b>1 Introducción</b>	<b>1</b>
<b>2 Estimador de estados</b>	<b>3</b>
2.1 Manejo de medidas retrasadas	3
2.1.1 Detalles de implementación	5
2.2 Modelo bidimensional de un quadrotor para el EKF	6
2.3 Simulación del quadrotor y del estimador	8
2.3.1 Resultados	9
<b>3 Posicionamiento mediante marcadores visuales</b>	<b>13</b>
3.1 Componentes	13
3.2 Programa ejecutado en el ordenador embebido	15
3.3 Detección de marcadores Aruco	18
3.4 Metodología de la experimentación	20
3.4.1 Iteración de los parámetros	21
3.4.2 Registro de resultados ( <i>logging</i> )	21
3.5 Resultados experimentales	22
<b>4 Conclusiones</b>	<b>27</b>
<b>5 Trabajos futuros</b>	<b>29</b>
<b>Apéndice A Simulador del estimador de estados</b>	<b>31</b>
<b>Apéndice B Detector de marcadores visuales</b>	<b>41</b>
B.1 main.cpp	41
B.2 vision_params.yml	44
B.3 marker_vision.h	44
B.4 mavlink_helper.h	51
<i>Índice de Figuras</i>	55
<i>Índice de Códigos</i>	57
<i>Bibliografía</i>	59



# Notación

---

$P$	Posición del vehículo en ejes inerciales
$V$	Velocidad del vehículo en ejes inerciales
$T$	Empuje aplicado en ejes cuerpo
$T_{rot}$	Empuje aplicado en ejes inerciales
$\theta$	Inclinación del quadrotor
$\tau$	Par aplicado al quadrotor
$\omega$	Medida del giróscopo
$a$	Medida del acelerómetro
$R^{UAV \rightarrow Marcador}$	Orientación del marcador vista desde el vehículo
$X$	Vector de estados



# Acrónimos

---

<i>UAV</i>	Vehículo aéreo no tripulado
<i>GCS</i>	Estación de control terrestre
<i>NED</i>	Ejes norte, este y abajo
<i>GNSS</i>	Sistema Global de Navegación por Satélite
<i>EKF</i>	Filtro de Kalman extendido



# 1 Introducción

---

Actualmente, los vehículos autónomos no están al alcance de cualquiera. Estos no deben de confundirse con los vehículos no tripulados, como los UAVs (Vehículos aéreos no tripulados) que si están más extendidos, llegando a utilizarse para el ocio o en negocios, estando al alcance del bolsillo de cada vez más gente. La mayoría tienen una autonomía parcial y necesitan la supervisión de un piloto en algunas de las fases del vuelo, por ejemplo, en el aterrizaje o cuando se navega cerca de obstáculos. Además, muchos de ellos solo pueden volar en exteriores donde le llega la señal de los satélites. La causa de todas estas limitaciones está en que no pueden determinar dónde están ubicados con exactitud, ya que la precisión que suelen tener es de varios metros. Si se mejorase ese aspecto, el número de aplicaciones en las que se podría utilizar sería enorme. Para empezar, todas las tareas que se realizan actualmente con un piloto, como el traslado urgente de muestras para detectar el coronavirus [1] o la detección mediante cámaras térmicas de koalas que están heridos por los incendios [2].

El interés en realizar estas tareas de forma autónoma no solo está en que cualquiera pueda hacer uso del vehículo, sin necesitar licencia ni habilidades especiales, sino que también hace el sistema más escalable. Si se quisiera, por ejemplo, instalar un sistema de reparto de paquetes mediante vehículos aéreos, y su flota se hace cada vez más grande, llegará un momento que sea difícil que todos los pilotos se pongan de acuerdo compartiendo el mismo espacio aéreo. Si esta planificación la hace en su lugar un ordenador, posiblemente los vuelos estarán más organizados y se pueda operar con más vehículos a la vez. Como última ventaja de la automatización se podría decir que el vehículo podría estar **disponible de forma inmediata** en cualquier momento, y no obligaría a los pilotos a estar trabajando en horas de descanso. Aprovechando esta ventaja, se podría utilizar en una aplicación de seguridad, en la que se programe un vuelo periódico para vigilar una propiedad o que buscase un intruso cuando se detecte una alarma. Frente a la opción de tener muchas cámaras instaladas, de esta otra forma se protegería más la privacidad del usuario.

Realmente, si se disponen de los suficientes recursos, ya existe la tecnología para conseguir ese nivel de automatización. Por ejemplo, mediante balizas sonoras, que son usadas en los interiores de fábricas, o si se opera en el exterior, existe la posibilidad de utilizar la *navegación cinética satelital en tiempo real* (RTK). Otra solución que no necesita de instalación en el entorno, son las cámaras o lídars, que con el uso de unos algoritmos llamados SLAM, pueden crear un mapa del entorno y ubicarse en él. El problema de estos es que



**Figura 1.1** Estación de carga de *Flytbase*.

o bien sus sensores son caros, o bien tienen una alta carga computacional que obliga a instalar grandes y costosos ordenadores a bordo del vehículo.

Lo que se busca en este trabajo es una **alternativa más barata** que consiga un posicionamiento centimétrico del vehículo. En concreto se ha hecho uso de unos marcadores visuales planos, que contienen figuras que son fácilmente detectables mediante el procesamiento automático de la imagen. Este método tiene mucha menos carga computacional que los anteriores, de hecho, para su procesamiento se usará uno de los ordenadores embebidos más baratos del mercado. Ya están a la venta algunas soluciones que utilizan esta tecnología para el aterrizaje automático. En particular, algunas empresas como *Everdrone* [3] o *Flytbase* (ver figura 1.1) ofrecen estaciones de carga con los mismos marcadores utilizados aquí.

Existen varios artículos en los que ya han utilizado marcadores visuales para mejorar el posicionamiento de un UAV, la mayoría teniendo como objetivo el aterrizaje automático. En [4] se consigue que un quadrotor *Parrot AR.Drone* aterrice sobre un marcador Aruco, incluso cuando se pierde la vista al marcador brevemente, gracias a las medidas de la IMU. En [5] además de la IMU, se utiliza un sensor de flujo óptico para conseguir aterrizar el vehículo. En ninguno de los dos trabajos se utiliza un autopiloto de código abierto, sino que se limitan a utilizar las interfaces de programación que aportan Parrot y DJI respectivamente, para tomar las medidas de los sensores y comandarle una postura de referencia. Además, la falta de detalles de implementación que suele haber en este tipo de artículos, hace que su trabajo sea más difícil de ser replicado por el lector. En este proyecto, lo que se quiere aportar es otro enfoque en el que se hace uso del estimador de estados del autopiloto, el mismo que se utiliza para estimar la orientación y al que le llegan las medidas de la IMU, magnetómetro, barómetro, etc., para obtener una fuente precisa de posición.

En el presente documento, en primer lugar, en el capítulo 2 se hace un estudio de una parte clave para el posicionamiento con marcadores, que es el estimador de estados. Además, se elabora y se muestran los resultados de una simulación en la que se pone a prueba este. En el capítulo 3 se explica cómo se ha implementado un posicionamiento basado en marcadores visuales, mostrando los componentes utilizados y explicando el programa creado. Finalmente, en ese mismo capítulo se enseñan los resultados experimentales a los que se ha llegado.

## 2 Estimador de estados

---

En este capítulo se analizará uno de los elementos que se utilizarán en el siguiente, que es el estimador de estados. En concreto se analiza el que incorpora el autopiloto de código abierto PX4, ya que será este el que se utilizará para la implementación. Previamente, en [6] se hizo una explicación parcial de este, pero se dejaron atrás algunos detalles como el manejo de las medidas retrasadas. Aquí se explicará esto y además se realizará una simulación que demuestre la eficacia del algoritmo.

### 2.1 Manejo de medidas retrasadas

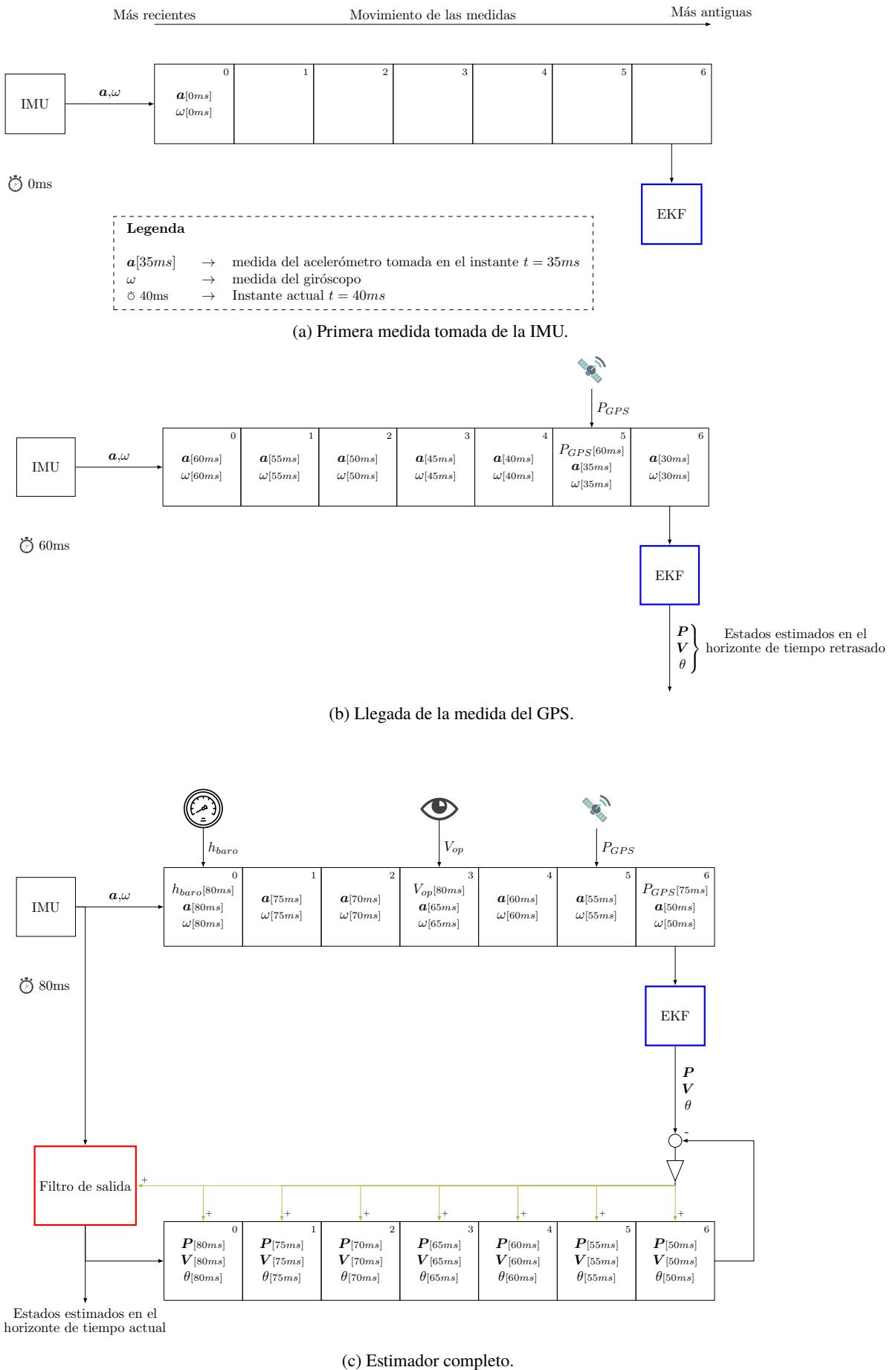
En muchas ocasiones se tienen sensores con unos retrasos muy diferentes entre ellos, por ejemplo una IMU es mucho más rápida que el procesamiento de la imagen de una cámara o el GNSS. PX4 lo soluciona añadiendo más elementos a la estructura original de un estimador de estados. Uno de sus elementos es un *Filtro de Kalman Extendido* (EKF). Este no usa las medidas más nuevas que le llegan, sino que las almacena y utiliza las que llegaron hace un determinado tiempo. Corriendo en paralelo, existe un estimador llamado *Filtro de Salida*, el cual sí que utiliza la última medida del acelerómetro y del giróscopo.

Supongamos que se tiene un sistema que se mueve en el espacio y del que se quiere conocer sus estados, en concreto, su posición, su velocidad y su orientación. Para este objetivo el sistema está dotado de numerosos sensores como son un acelerómetro, un giróscopo, un barómetro, un GNSS o un sensor de flujo óptico. Cada uno de ellos tiene diferentes propiedades en cuanto a retraso, ruido, precisión, etc. Por ejemplo, la medida aportada por el GNSS es la única fuente de posición absoluta, sin embargo, tiene un gran retraso y las medidas que genera, se refieren a la posición que se tenía hace un tiempo (generalmente decenas de milisegundos).

Para explicar un método de cómo afrontar este problema, se va a poner un ejemplo de la ejecución paso a paso del estimador de estados con diferentes sensores. Supongamos que en la primera ejecución del estimador, se toma la primera medida de la IMU (acelerómetro y giróscopo). El EKF todavía no la utiliza, si no que la guarda en su buffer (figura 2.1a). Conforme llegan nuevas medidas, que ocurre cada 5 ms, estas se introducen en la posición de más a la izquierda del buffer y las que ya estaban se van desplazando hacia la derecha, hasta que llegan a la última celda. Las medidas de esta celda situada más a la derecha, son las que son usadas por el EKF. Los estados que este genera y las medidas utilizadas para estimarlos se refieren al *horizonte de tiempo retrasado*. Como se muestra en la figura, el buffer tiene una longitud de 7 celdas, por lo tanto las medidas de la IMU que llegan al EKF siempre serán las que se recogieron hace 30 ms.

Pasan algunos ciclos más hasta que en el instante 60ms llega la primera medida del GPS, pero esta no se coloca en el extremo izquierdo del buffer junto con las medidas más recientes de la IMU, si no que se lleva directamente a la celda número 5 (ver figura 2.1b). En esta también se encuentran las medidas de la IMU tomadas en el instante 35ms, es decir las que fueron tomadas hace 25 ms, que coincide con el retraso que tiene la posición del GPS con respecto a la IMU o lo que es lo mismo, la medida del GPS, corresponde a la posición que tenía el vehículo hace 25 ms. De esta manera se agrupan las medidas que se refieren al mismo instante físico, es decir, el instante en el que llegaron pero **compensándose su retraso**.

De forma paralela se ejecuta el *filtro de salida*, que es otro estimador de estados y para esta explicación se va a suponer que su funcionamiento interno es exactamente igual al del EKF, la única diferencia es que solamente utiliza las medidas de la IMU, en este caso las que se generan más recientemente. Estos estados se refieren al *horizonte de tiempo actual* y son los únicos que se usan para las otras tareas que tenga vehículo, como por ejemplo para alimentar al controlador de orientación, por esta razón se le denomina filtro de salida.



**Figura 2.1** Manejo de medidas retrasadas.

El problema que tiene este es que desaprovecha todos los demás sensores que tiene el vehículo, por lo que se le aplica un **mecanismo de corrección**.

Este mecanismo está compuesto otro buffer llamado *buffer de salida*, que se comporta de la misma manera que el primero, pero en lugar de guardar medidas, almacena los estados del filtro de salida. Estos estados se van desplazando hacia la derecha hasta que llegan a la última posición del buffer. En esta posición están los estados que se estimaron por el filtro de salida hace 30 ms, que coincide con el retraso que tienen las medidas del la IMU que entran al EKF. Si al EKF solo se le hubiese suministrado las medidas de la IMU, al igual que al filtro de salida, los estados del EKF y los que hay almacenados en esta última celda del buffer de salida coincidirían. Sin embargo, lo que está ocurriendo es que el EKF recoge medidas de otros sensores y por lo tanto no coincidirán. Para realizar la corrección, se calcula la diferencia entre ellos. Esta diferencia se atenúa y se le suma a todos los elementos del buffer de salida, además de al propio filtro de salida.

En la figura 2.1c se ha ilustrado este mecanismo de corrección además de incluir más medidas: la velocidad proporcionada por flujo óptico ( $V_{op}$ ), la cual tiene un retraso de 15ms, y la altura que proporciona el barómetro, que se ha supuesto que no tiene ningún retraso.

### 2.1.1 Detalles de implementación

En este apartado se presentan algunos trozos del código de PX4 que implementan lo anteriormente descrito, además de algunos detalles que se han omitido en el apartado anterior para que fuese más fácil su comprensión.

En el apartado anterior, se explicó que el error entre los estados del EKF y los del filtro de salida se atenúan (multiplicar por una ganancia menor que 1, mostrado en la figura 2.1c como un triángulo) y se le suma a todo el buffer de salida. En el siguiente código se puede comprobar cómo se ha implementado esto. Se puede ver que la corrección de la velocidad y la posición no solo se calcula a partir del error, sino también a partir de la integral del error, por lo tanto aquí se tiene un control proporcional-integral, que tiene como señal de control la corrección al buffer de salida. De esta manera, los estados en el horizonte de tiempo actual, se igualarán a los del horizonte de tiempo retrasado en régimen permanente.

**Código 2.1: Corrección del buffer de salida. Ubicado en la línea 488 del archivo *Firmware/src/lib/ecl/EKF/ekf.cpp***

```

488 void Ekf::applyCorrectionToOutputBuffer(float vel_gain, float pos_gain){
489     // calculate velocity and position tracking errors
490     const Vector3f vel_err(_state.vel - _output_sample_delayed.vel);
491     const Vector3f pos_err(_state.pos - _output_sample_delayed.pos);
492
493     _output_tracking_error(1) = vel_err.norm();
494     _output_tracking_error(2) = pos_err.norm();
495
496     // calculate a velocity correction that will be applied to the output state
497     // history
498     _vel_err_integ += vel_err;
499     const Vector3f vel_correction = vel_err * vel_gain + _vel_err_integ *
500     // sq(vel_gain) * 0.1f;
501
502     // calculate a position correction that will be applied to the output state
503     // history
504     _pos_err_integ += pos_err;
505     const Vector3f pos_correction = pos_err * pos_gain + _pos_err_integ *
506     // sq(pos_gain) * 0.1f;
507
508     // loop through the output filter state history and apply the corrections to the
509     // velocity and position states
510     for (uint8_t index = 0; index < _output_buffer.get_length(); index++) {
511         // a constant velocity correction is applied
512         _output_buffer[index].vel += vel_correction;
513
514         // a constant position correction is applied
515         _output_buffer[index].pos += pos_correction;
516     }
517
518     // update output state to corrected values
519     _output_new = _output_buffer.get_newest();
520 }
```

En el código anterior no ha aparecido la corrección de la orientación, y esto es porque requiere que sea tratada a parte. En este estimador, la orientación se expresa en cuaternios y la operación de la corrección no es simplemente una suma como ocurría en el caso de la velocidad, sino que es más complicada y se tardaría demasiado en aplicarla a todos los elementos del buffer. En su lugar, únicamente se aplica una corrección a la orientación estimada en el horizonte de tiempo actual.

**Código 2.2: Corrección de la orientación. Ubicado en el archivo [Firmware/src/lib/ecl/EKF/ekf.cpp](#)**

En la línea 323 se corrige la orientación:

```
323     // Apply corrections to the delta angle required to track the quaternion states at the
     ↵ EKF fusion time horizon
324     const Vector3f delta_angle(imu.delta_ang - _state.delta_ang_bias * dt_scale_correction
     ↵ + _delta_angle_corr);
```

En la línea 411 se calcula la ganancia del control

```
411     // calculate a gain that provides tight tracking of the estimator attitude states
     ↵ and
412     // adjust for changes in time delay to maintain consistent damping ratio of ~0.7
413     const float time_delay = fmaxf((imu.time_us - _imu_sample_delayed.time_us) * 1e-6f,
     ↵ _dt_imu_avg);
414     const float att_gain = 0.5f * _dt_imu_avg / time_delay;
415
416     // calculate a correction to the delta angle
417     // that will cause the INS to track the EKF quaternions
418     _delta_angle_corr = delta_ang_error * att_gain;
```

Tampoco se ha hablado de la longitud de los buffers, la cual hay que determinar antes de empezar a estimar. En el siguiente código se puede ver cómo se calcula la longitud del buffer de medidas de la IMU. Esta se determina de manera que el EKF se ejecutará con un retraso igual al sensor que más retraso tiene.

**Código 2.3: Cálculo del tamaño del buffer. Ubicado en el archivo [Firmware/src/lib/ecl/EKF/estimator\\_interface.cpp](#)**

```
512     // find the maximum time delay the buffers are required to handle
513     const uint16_t max_time_delay_ms = math::max(_params.mag_delay_ms,
514                                                 math::max(_params.range_delay_ms,
515                                                       math::max(_params.gps_delay_ms,
516                                                       math::max(_params.flow_delay_ms,
517                                                       math::max(_params.ev_delay_ms,
518                                                       math::max(_params.auxvel_delay_ms,
519                                                       math::max(_params.min_delay_ms,
520                                                       math::max(_params.airspeed_delay_ms,
521                                                       _params.baro_delay_ms))))));
522
523     // calculate the IMU buffer length required to accomodate the maximum delay with some
     ↵ allowance for jitter
524     _imu_buffer_length = (max_time_delay_ms / FILTER_UPDATE_PERIOD_MS) + 1;
```

## 2.2 Modelo bidimensional de un quadrotor para el EKF

El objetivo de esta sección, es el de realizar los cálculos necesarios para montar la simulación que vendrá más adelante. Dicha simulación tiene el fin de probar el manejo de medidas retrasadas explicadas anteriormente, no se buscará que sea especialmente realista, añadiendo todos los sensores que se utilizan en la práctica, sino que sea lo más simple posible para centrar la atención en el aspecto que se quiere explicar. Dicho esto, un quadrotor moviéndose en el plano, cuyos únicos sensores son el acelerómetro, giróscopo y GNSS servirá para probar el estimador de estados.

Para realizar el filtro, se tomará prestada la idea de PX4 de suponer que no se tiene conocimiento de ningún aspecto dinámico del vehículo: pares, fuerzas, masa e inercias. En su lugar, se propone un **modelo cinemático** para la fase de predicción del filtro, en el que las entradas son la aceleración lineal y la velocidad angular, que aportan el acelerómetro y giróscopo respectivamente.

Ya que se utilizará un filtro de Kalman discreto, se necesita un modelo discreto en el espacio de estados:

$$\mathbf{X}_{k+1} = \begin{bmatrix} x \\ y \\ V_x \\ V_y \\ \theta \end{bmatrix}_{k+1} = f(\mathbf{X}_k, \mathbf{u}_k) + \mathbf{w}_k \quad (2.1)$$

$$\mathbf{z}_k = \begin{bmatrix} P_x^{\text{GNSS}} \\ P_y^{\text{GNSS}} \end{bmatrix}_k = h(\mathbf{X}_k) + \mathbf{v}_k \quad (2.2)$$

Siendo  $\mathbf{X}_k$  los estados estimados en la iteración anterior, que están compuestos por la posición, velocidad e inclinación estimadas;  $\mathbf{X}_{k+1}$  los estados predichos,  $\mathbf{u}_k = [a_x \ a_y \ \omega]^T$  el vector de entradas formado por las dos medidas del acelerómetro en los ejes  $x$  y  $y$  del cuerpo y la medida del giroscopio,  $f()$  el modelo de predicción,  $\mathbf{z}_k$  el vector de medidas que en este caso solo contiene la posición del GNSS y  $h()$  el modelo de observación. Por último,  $\mathbf{w}_k$  es el ruido de predicción y  $\mathbf{v}_k$  es el de observación, ambos se desarrollarán más adelante.

Para obtener el modelo de predicción, se realiza la integral discreta a la velocidad,

$$\begin{bmatrix} x \\ y \end{bmatrix}_{k+1} = \begin{bmatrix} x \\ y \end{bmatrix}_k + \begin{bmatrix} V_x \\ V_y \end{bmatrix}_k \Delta t \quad (2.3)$$

también se hace con la velocidad angular,

$$\theta_{k+1} = \theta_k + \Delta t \omega \quad (2.4)$$

y por último se integra la aceleración en ejes iniciales compensando la medida de la aceleración gravitatoria:

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix}_{k+1} = \begin{bmatrix} V_x \\ V_y \end{bmatrix}_k + \Delta t \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \mathbf{a} + \begin{bmatrix} 0 \\ -g \end{bmatrix} \Delta t \quad (2.5)$$

Viendo este modelo de predicción  $f()$  formado por las tres últimas ecuaciones, se puede comprobar que no es lineal con respecto a los estados (debido a las funciones trigonométricas) y por lo tanto no se puede utilizar un filtro de Kalman convencional, sino uno extendido. Este se basa en realizar una aproximación de primer orden del modelo no lineal:

$$\Delta f(\Delta \mathbf{X}_k) \approx \mathbf{F}_k \Delta \mathbf{X}_k \quad (2.6)$$

Siendo  $F_k$  el jacobiano del modelo de predicción con respecto a los estados, que para este caso tiene la siguiente expresión:

$$F_k = \left. \frac{\partial f}{\partial X} \right|_{X_{k-1}} = \begin{bmatrix} 1 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & \Delta t (-a_x \sin \theta_{k-1} + a_y \cos \theta_{k-1}) \\ 0 & 0 & 0 & 1 & \Delta t (-a_x \cos \theta_{k-1} - a_y \sin \theta_{k-1}) \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.7)$$

En cuanto al modelo de observación es bastante más sencillo y lineal en este caso, ya que las medidas corresponden directamente con los estados:

$$h(\mathbf{X}_k) = \begin{bmatrix} x_k \\ y_k \end{bmatrix} \quad (2.8)$$

Lo último que queda por desarrollar del modelo completo son los ruidos, que se suponen que siguen una distribución normal:

$$\mathbf{w}_k \sim \mathcal{N}(0, Q_k) \quad (2.9)$$

$$\mathbf{v}_k \sim \mathcal{N}(0, R_k) \quad (2.10)$$

Donde  $R_k$  es la matriz de covarianzas del error de observación, que se le ha impuesto el siguiente valor:

$$R_k = \begin{bmatrix} 1cm^2 & 0 \\ 0 & 1cm^2 \end{bmatrix} \quad (2.11)$$

En cuanto a  $Q_k$ , es la matriz de covarianzas del error de predicción, que se calcula en base al ruido de las entradas de la siguiente manera:

$$Q_k = G_k \begin{bmatrix} \sigma_a^2 & 0 & 0 \\ 0 & \sigma_a^2 & 0 \\ 0 & 0 & \sigma_\omega^2 \end{bmatrix} G_k^T \quad (2.12)$$

Donde  $\sigma_a$  y  $\sigma_\omega$  son las desviaciones típicas del acelerómetro y del giróscopo, que se pueden hallar experimentalmente o viendo la hoja de datos de los sensores.  $G_k$  es el jacobiano del modelo de predicción con respecto las entradas:

$$G_k = \frac{\partial f}{\partial \mathbf{u}_k} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \Delta t \cos \theta & \Delta t \sin \theta & 0 \\ -\Delta t \sin \theta & \Delta t \cos \theta & 0 \\ 0 & 0 & \Delta t \end{bmatrix} \quad (2.13)$$

Una vez descrito el modelo completo, ya se tiene todo lo que hace falta para ejecutar los pasos del filtro de Kalman extendido. Dichos pasos se implementarán en la simulación siguiente y el lector que quiera saber sobre ellos puede encontrarlos en [6].

### 2.3 Simulación del quadrotor y del estimador

En este apartado se implementará un estimador de estados con las características explicadas en este capítulo y se pondrá a prueba con el simulador de un quadrotor. Tanto el estimador como el simulador estarán programados en lenguaje Python<sup>1</sup>. El simulador será muy sencillo, describirá el movimiento de un quadrotor en el plano al que únicamente se le aplican la fuerza de la gravedad, un empuje y un par. Estos dos últimos serán generados por un controlador de velocidad vertical y un controlador de ángulo, los cuales toman la velocidad y la inclinación real del vehículo, en lugar de utilizar medidas ruidosas. Sus referencias se han escogido para que desde el reposo, ascienda unos metros, y luego se desplace hacia la dirección negativa del eje  $x$ .

Para obtener la evolución de la inclinación  $\theta$ , se aplica la segunda ley de Newton utilizando el par de actuación  $\tau$  y la inercia  $I$ :

$$\ddot{\theta} = \frac{\tau}{I} \quad (2.14)$$

Se obtiene la velocidad angular a partir de la integración discreta de la aceleración,

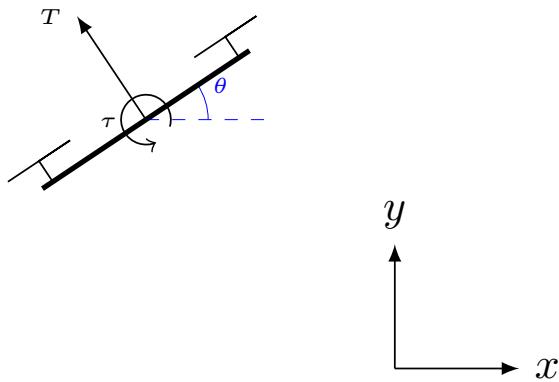
$$\dot{\theta} = \dot{\theta}_{i-1} + \Delta t \ddot{\theta} \quad (2.15)$$

y la inclinación a partir de la integral discreta de la velocidad angular:

$$\theta = \theta_{i-1} + \Delta t \dot{\theta} \quad (2.16)$$

---

<sup>1</sup> En el apéndice A se encuentra el script *main.py* que implementa toda la simulación de manera independiente, sin necesidad de programas externos



**Figura 2.2** Quadrotor en dos dimensiones.

En cuanto a la translación, también se aplica la segunda ley de Newton en ejes inerciales:

$$a = \frac{T_{rot} + F_g}{m} \quad (2.17)$$

Donde  $\mathbf{F}_g$  es la fuerza gravitatoria, y  $\mathbf{T}_{rot}$  es el empuje del quadrotor en ejes iniciales, que se calcula estableciendo que siempre tiene la dirección del eje y del cuerpo y que su magnitud es  $T$ :

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (2.18)$$

$$\boldsymbol{T}_{rot} = R \begin{bmatrix} 0 \\ T \end{bmatrix} \quad (2.19)$$

$$\mathbf{F}_g = \begin{bmatrix} 0 \\ -mg \end{bmatrix} \quad (2.20)$$

La velocidad lineal se obtiene a partir de la aceleración calculada previamente,

$$v = v_{i-1} + a\Delta t \quad (2.21)$$

e integrando esta última se obtiene la posición:

$$p = p_{i-1} + v\Delta t \quad (2.22)$$

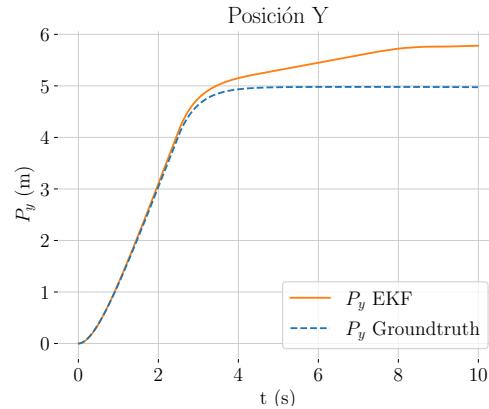
Una vez se ha simulado esta trayectoria, se pasa ejecutar el estimador de estados. Este toma unas medidas a las que se le ha aplicado un ruido gaussiano y genera su estimación de los estados. Finalmente estos se comparan con los estados reales y se verifica el desempeño del estimador.

### **2.3.1 Resultados**

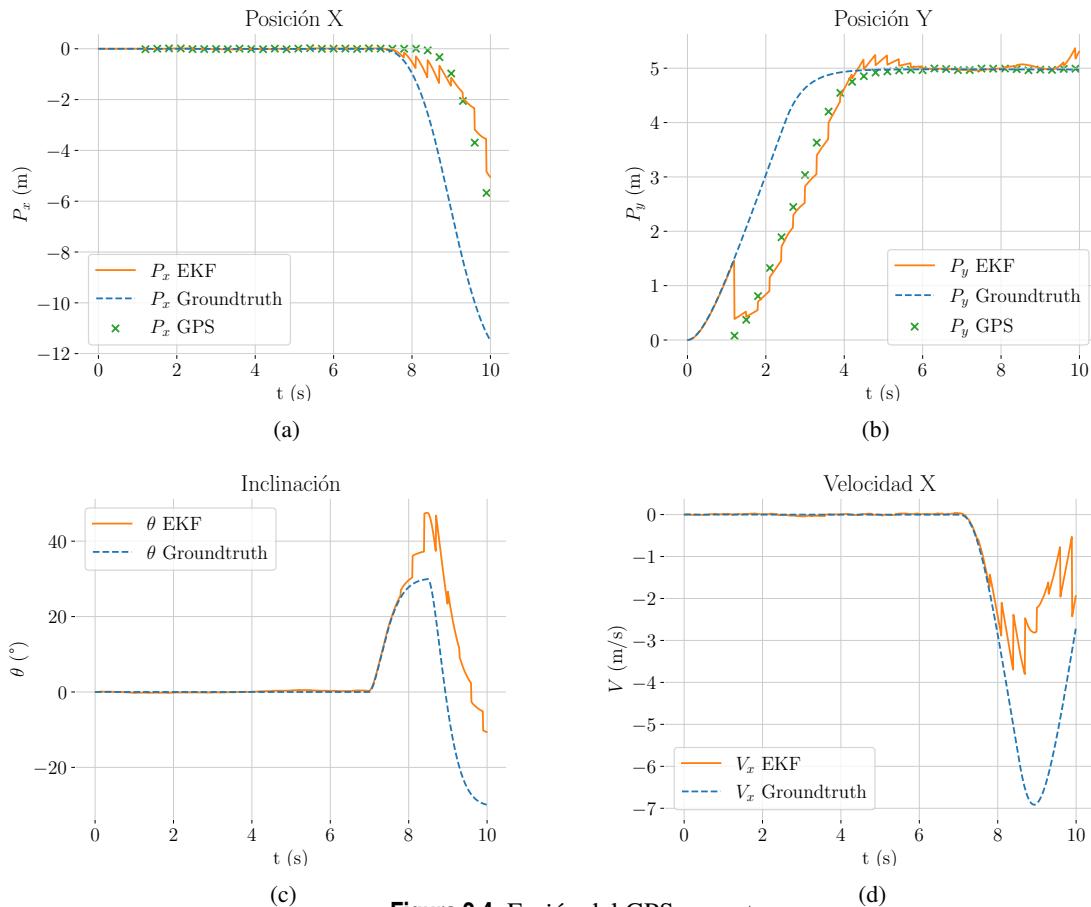
El primer experimento que se va a mostrar, al estimador de estados no le va a entrar ninguna otra medida que no sea la del giróscopo y la del acelerómetro. En la figura 2.3 se puede apreciar que la estimación de la posición tiene una deriva, ya que no hay ningún sensor que aporte posición absoluta.

En el siguiente experimento se va a fusionar un GPS con un retraso de 1 segundo y un periodo de 300ms. Estos valores son poco realistas pero de esta manera se aprecia más la degradación de la estimación. Se puede ver en la figura 2.4 que el GPS empeora la estimación, ya que en el instante  $t = 1s$ , la posición estimada se aleja de su valor real porque llega su primera medida. Aquí se puede aprovechar para ver cómo se comporta el filtro de Kalman cuando la medida tiene mucho más error del que se ha especificado. En este caso se le indicó al filtro que tendría una desviación típica de 1 centímetro, mientras que en realidad está cometiendo errores de más de un metro a causa del retraso. También se puede ver la dependencia mutua entre todos los estados: la medida del GPS no afecta solo a los estados de la posición, sino que también a la velocidad y a la orientación.

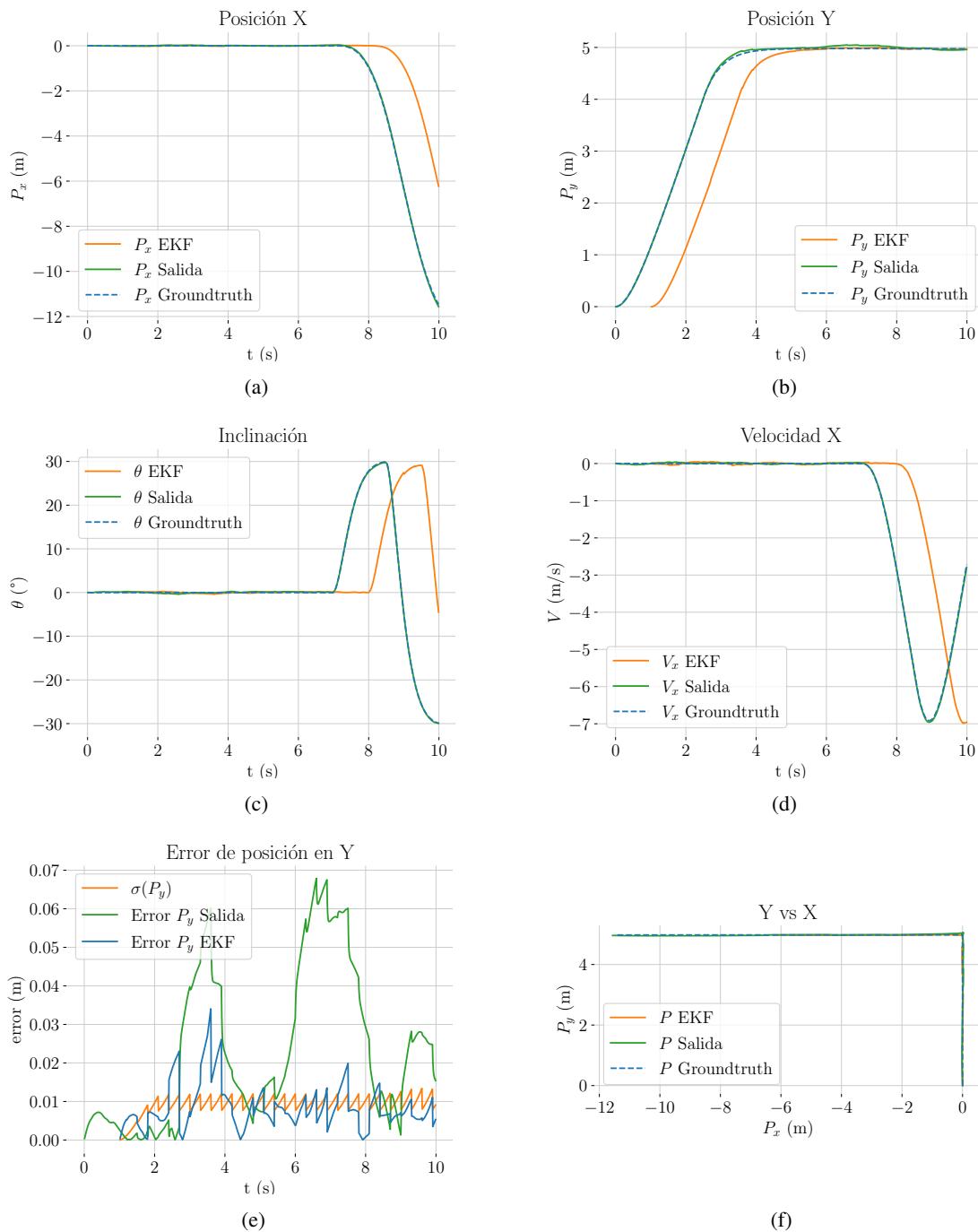
En el tercer experimento se ha activado el manejo de los retrasos explicado en este capítulo. En la figura 2.5 se tiene que, aunque la medida esté retrasada un segundo, esta no se ve degradada por el GPS ya que no se está fusionando con las medidas más recientes de la IMU, sino con aquellas que llegaron hace un segundo. Es más, la estimación es mejor que en los dos casos anteriores. En la imagen 2.5e se puede ver la desviación típica que estima el filtro ( $\sigma$ ), que tiene forma de diente de sierra debido a la llegada periódica de las medidas del GPS. Esta se compara con los valores reales del error, tanto el del filtro de salida como el del EKF. Para este último filtro, el error se calcula como la diferencia entre la estimación con el groundtruth retrasado la misma cantidad de tiempo que dicho filtro. Se puede notar en la imagen, que dicho error no está muy alejado del estimado, sin embargo, el error del filtro de salida es mayor.



**Figura 2.3** EKF no ejecuta la fase de actualización.



**Figura 2.4** Fusión del GPS con retraso.



**Figura 2.5** Experimento con manejo de medidas retrasadas.



# 3 Posicionamiento mediante marcadores visuales

---

Conseguir con precisión la posición de un vehículo aéreo no tripulado es bastante deseable. En la introducción se comentó aplicaciones como la manipulación de objetos o la navegación cerca de obstáculos. En este capítulo se explica que para conseguirlo, se ha construido un quadrotor con los componentes necesarios para detectar un marcador visual. Además, se comenta cómo se ha programado un ordenador embebido para que procese dicho marcador.

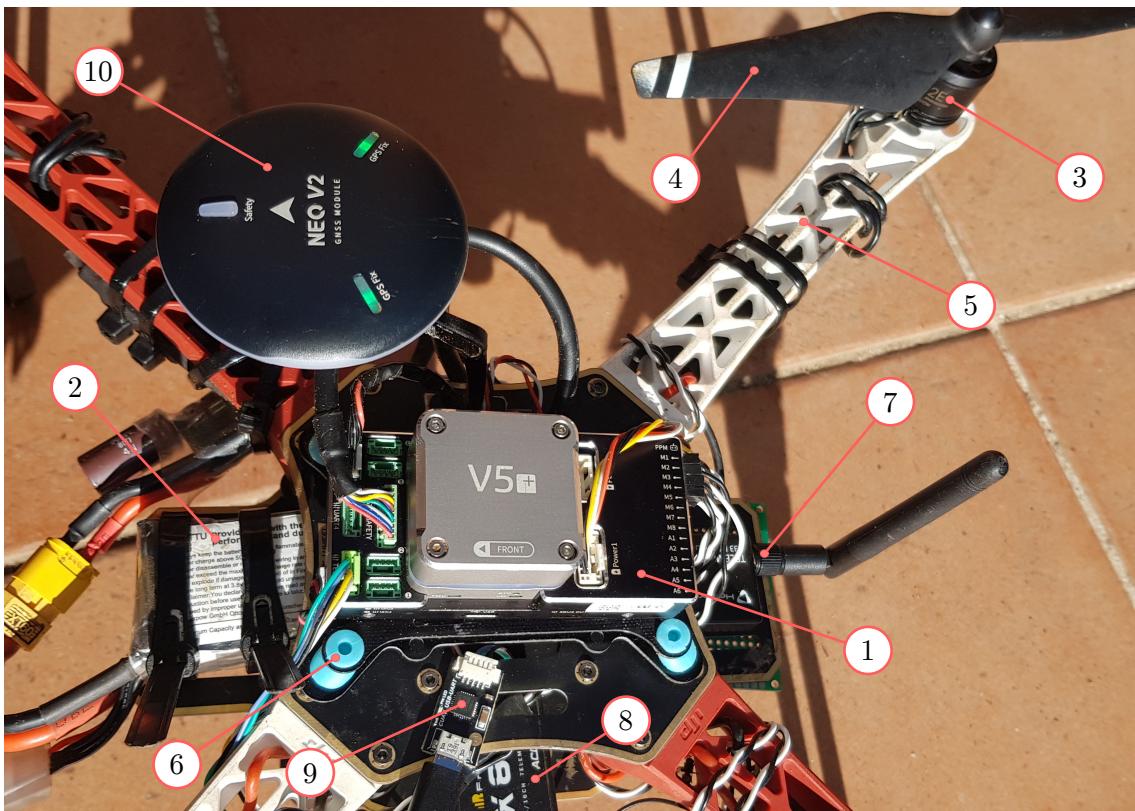
## 3.1 Componentes

Para elegir los componentes se ha tenido en cuenta que no estén discontinuados, para comprar posibles recambios, la rapidez de llegada ya que todos llegan por paquetería, que estén ampliamente probados, y que en la medida de lo posible, estuvieran liberados tanto su software como su hardware.

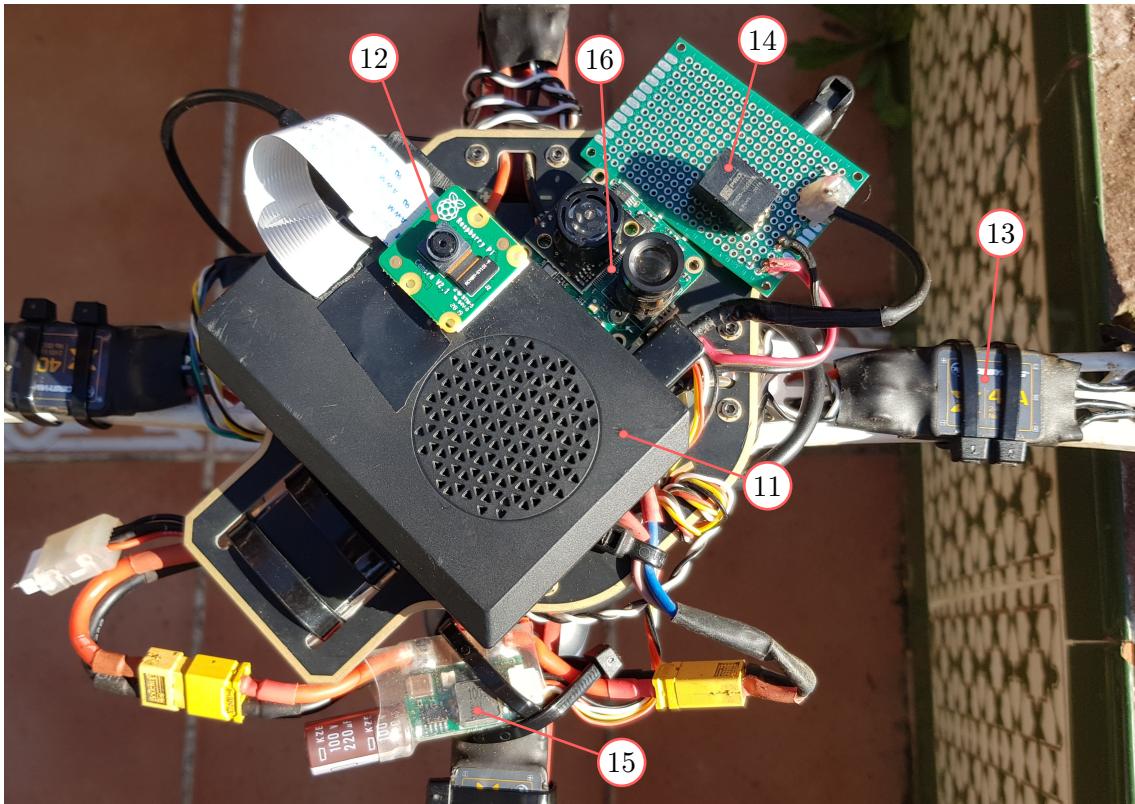
1. Cuav V5+. Autopiloto corriendo PX4. Esquemáticos publicados en [Github](#).
2. *Tattu Funfly 1500mAh*. Batería LiPo de 4 celdas.
3. *DJI 2312E 800KV*. Motor sin escobillas.
4. Hélices de fibra de carbono con un diámetro de 9.4 pulgadas y un paso de 5 pulgadas. Según el [fabricante](#) del motor, con esta hélice se consigue un empuje de 850 gramos alimentado a 14.8 V.
5. *DJI F450*. Chasis de quadrotor de 45 cm de diagonal.
6. Cama amortiguadora para el autopiloto<sup>1</sup>.
7. Módulo de telemetría *Holybro V3*. Permite una comunicación con la estación de control terrestre.
8. Receptor *X8R*. Recibe hasta 16 canales de la emisora, que este caso es una *Taranis Q X7*.
9. *SILABS CP2102*. Puente USB-UART. Se conecta entre el puerto USB del ordenador embebido y el puerto UART del autopiloto.
10. CUAV NEO V2. Este incluye GNSS, magnetómetro, botón de armado, luces indicadoras y alarma sonora. Se monta más arriba que el resto de componentes para alejarlo de los circuitos de potencia y así reciba menos interferencias el magnetómetro.
11. Raspberry Pi 4 Model B. 4 GB de RAM. Se encuentra protegida por una carcasa que incorpora un ventilador.
12. Raspberry Pi Camera Module v2. Campo de visión horizontal de 62 grados, capaz de grabar vídeo con resolución de 1640x1232 a 40fps.
13. *CUAV HVP M (High-Voltage Power Module)*. Regulador de voltaje para alimentar el autopiloto. Además, lee el voltaje y la corriente que suministra la batería.

---

<sup>1</sup> Este componente, al igual que muchos otros, fue comprado en la tienda online [rc-innovations.es](#)



(a) Vista desde arriba.



(b) Vista desde abajo.

**Figura 3.1** Componentes del quadrotor.

14. *Hobbywing XRotor 40A*. Variador de velocidad o ESC. Estos están sobredimensionados ya que fabricante recomienda unos que soporten como mínimo una corriente de 20A.
15. *RS PRO K7805-2000R3L*. Reductor de voltaje de 5V y 2A. Este se utilizará para alimentar el ordenador embebido a partir de la batería. Su voltaje permitido de entrada está entre los 8V y los 32V, lo cual es adecuado para una batería LiPo de 4 celdas.
16. *CUAV PX4FLOW 2.1*. Sensor de flujo óptico. También tiene su software liberado y el esquemático de una versión anterior.
17. Extensor de piernas. Estás fueron impresas mediante la empresa [Impresion 3D LowCost](#) con un modelo tomado de la página [Thingiverse](#). Son necesarias ya que el chasis tiene unas patas demasiado cortas y no dejaban espacio para la Raspberry Pi y su cámara.

## 3.2 Programa ejecutado en el ordenador embebido

De forma resumida, la cámara, que se ha colocado en la parte inferior del quadrotor y conectada al ordenador embebido, captura imágenes de un marcador que se ha impreso y se ha colocado en el suelo. Este ordenador las procesa y genera una posición estimada del UAV con respecto al marcador, que es enviada al autopiloto a través del puerto serie. El autopiloto la fusiona en su estimador de estados y genera una posición estimada que alimenta al controlador de posición. El controlador de posición toma esta medida y sigue la referencia. Esta última puede venir o bien del mando, o bien del ordenador embebido el cual le indique una trayectoria.

Nótese que el controlador de posición también se podría haber ubicado en el ordenador embebido, generando consignas de inclinación al autopiloto. La desventaja de esto es que se no se utilizan los demás sensores para el posicionamiento. De la manera que aquí se ha implementado, si en un instante falta la medida de la visión, el autopiloto podría tomar otras como la del acelerómetro, el GPS o el flujo óptico, para fusionarlas en su estimador de estados mientras se espera a que se recupere la medida de la visión.

En la figura 3.2 se puede ver el diagrama de flujo del programa que se corre en la Raspberry Pi, cuyos pasos se detallarán a continuación.

### 1. Inicialización:

En este paso se espera a detectar el autopiloto y se leen los parámetros de un archivo dedicado a ello.

### 2. Recoger imagen de la cámara:

Este paso podría llegar a ser muy lento si no se escoge una interfaz con la cámara adecuada, por ejemplo USB. En este caso se ha escogido CSI, que lleva la imagen directamente a la GPU y esta la transfiere a la RAM mediante DMA<sup>2</sup>. Esta tiene la desventaja que el cable es plano y más difícil de torsionar. La ventaja es que la imagen llega a la RAM sin consumir tiempo de CPU, permitiendo que esta haga en paralelo otras operaciones como el procesamiento de imagen.

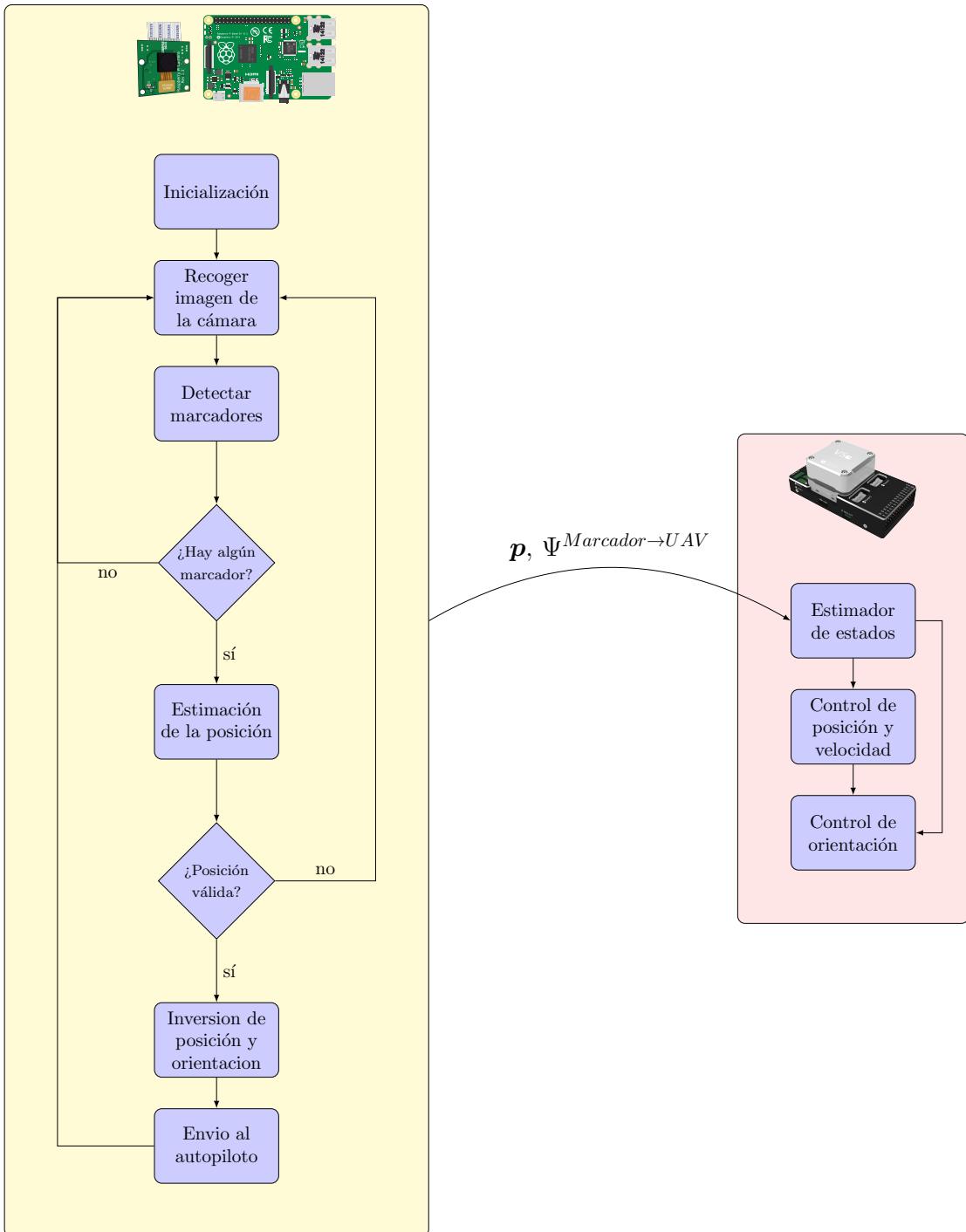
### 3. Detectar marcadores:

El objetivo es ubicar los marcadores en la imagen (en concreto sus 4 esquinas) y extraer su identificador. Este proceso será explicado en la sección 3.3.

### 4. Estimación de la posición:

La estimación de la posición y la orientación se realiza tomando las 4 esquinas de los marcadores obtenidos en el paso anterior. Este problema se denomina PnP (Perspectiva desde n puntos) y su solución es iterativa. Parte de que, dados unos puntos 3D en el espacio, expresados en un sistema de referencia exterior a la cámara y dada la posición de la cámara con respecto a dicho sistema de referencia, se puede predecir qué posición en el plano de la imagen tendrían esos puntos al ser proyectados. Lo que se busca es exactamente lo contrario: la posición de la cámara con respecto a dicho sistema de referencia, a partir de la proyección en la imagen de los puntos tridimensionales. Lamentablemente no se puede invertir las ecuaciones y por tanto no se puede obtener una solución analítica. Para hallar la solución se recurren a algoritmos de optimización que van probando posiciones de la cámara, hacen proyecciones suponiendo esa posición y se compara con las proyecciones reales. A la diferencia de estas dos proyecciones se le denomina *error de reproyección* y es el valor que se trata de minimizar. Para este proyecto este problema no se ha tenido que implementar, solo se ha tenido que llamar a la función *estimatePoseSingleMarkers* de la librería Aruco, que a su vez llama a la función *solvePnP* de OpenCV.

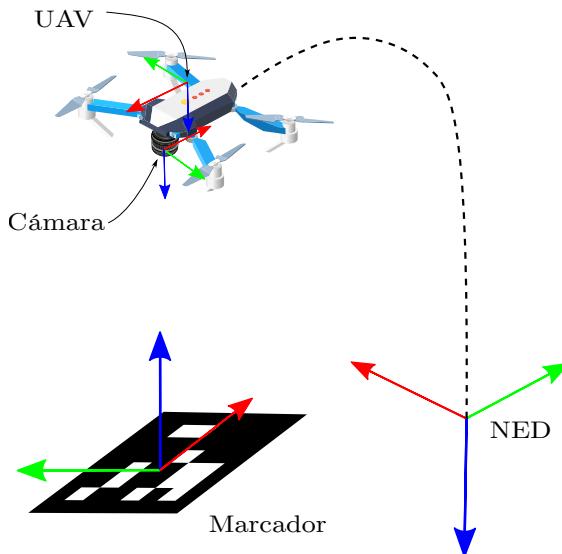
<sup>2</sup> Para más información del proceso de captura visitar la excelente documentación de la interfaz Python de la cámara: <https://picamera.readthedocs.io/en/release-1.13/fov.html#division-of-labor>



**Figura 3.2** A la izquierda: diagrama de flujo del programa que se corre en el ordenador embebido, a la derecha: algunas de las tareas del autopiloto.

##### 5. Inversión de la posición y orientación:

Como se ve en la figura 3.3, hay varios sistemas de referencia que entran en juego, y hay que tenerlos presentes para transformar desde lo que aporta la estimación de la posición del paso anterior, hasta lo que necesita el autopiloto. En primer lugar está el sistema de referencia **NED**, que es aquel cuyos ejes  $x$ ,  $y$  y  $z$  están apuntando hacia el norte, este y abajo respectivamente, y cuyo origen se sitúa en la posición desde la que partió el **UAV**. Este último tiene un sistema de referencia con su eje  $x$  apuntando hacia la dirección de avance y su eje  $z$  apuntando hacia el suelo cuando no está inclinado. Un poco más para abajo, pero para los cálculos, considerado en la misma posición que el anterior, se encuentra el sistema



**Figura 3.3** Sistemas de referencia presentes en el problema.

de referencia de la **cámara**, el cual tiene el sistema convencional ellas: si se viese una fotografía tomada por esta, su eje  $x$  apuntaría hacia la derecha, su eje  $y$  hacia abajo y su eje  $z$  hacia fuera de la cámara. Por último, se puede ver el sistema de referencia del marcador, cuyo eje  $z$  apunta hacia arriba y los otros dos son paralelos a los bordes del marcador. A parte de los que vienen dibujados, se definirán más adelante, unos sistemas auxiliares  $Cámara'$  y  $NED'$  que son derivados de los anteriores.

En el paso anterior, la orientación y posición que se obtiene es la del **marcador con respecto a la cámara**, es decir se obtiene  $R^{Cámara \rightarrow Marcador}$  y  $p^{Cámara \rightarrow Marcador}$ . Lo primero que se realiza es buscar la orientación de la cámara con respecto al marcador. Para ello tenemos que invertir la orientación dada, la cual al estar expresada en forma de una matriz de rotación, se puede obtener mediante su traspuesta:

$$R^{Marcador \rightarrow Cámara} = (R^{Cámara \rightarrow Marcador})^T \quad (3.1)$$

Dicha matriz se utiliza para expresar la posición del marcador en unos ejes paralelos a los ejes del marcador.

$$p^{Cámara' \rightarrow Marcador} = R^{Marcador \rightarrow Cámara} \cdot p^{Cámara \rightarrow Marcador} \quad (3.2)$$

Nótese que esta posición sigue teniendo origen en la cámara, solo que ahora está rotado el sistema de referencia en el que se expresa. Lo que se quiere obtener es la posición con respecto al sistema de referencia del marcador, el cual es paralelo al que se está expresando ahora. Siempre que existen dos sistemas de referencia A y B, con la misma orientación pero ubicados en distintos puntos, se debe de negar la posición de A con respecto a B para conseguir la posición de B con respecto a A. Por esta razón la posición que finalmente se le manda al autopiloto es la negada de la obtenida en la última ecuación.

$$p^{Marcador \rightarrow Cámara'} = -p^{Cámara' \rightarrow Marcador} \quad (3.3)$$

En cuanto a la orientación, como se ve en la figura 3.3, los ejes de la cámara y los del UAV están rotados 180° con respecto al eje  $z$ . Conociendo esto, se obtiene la orientación del marcador vista desde el sistema de referencia del UAV:

$$R^{UAV \rightarrow Cámara} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.4)$$

$$R^{UAV \rightarrow Marcador} = R^{UAV \rightarrow Cámara} \cdot R^{Cámara \rightarrow Marcador} \quad (3.5)$$

Obviamente, la multiplicación se tiene realizar en ese orden. De esta manera, la rotación se realiza con respecto al eje  $z$  de la cámara, mientras que si se hubiese invertido el orden, es decir se postmultiplica a  $R^{Cámara \rightarrow Marcador}$ , la rotación se hubiese hecho alrededor del eje  $z$  del marcador. Finalmente, esta última matriz se transpone para tener la orientación del UAV con respecto al marcador.

$$R^{Marcador \rightarrow UAV} = (R^{UAV \rightarrow Marcador})^T \quad (3.6)$$

Tras expresar esta rotación en ángulos de Euler, ya se podría enviar al autopiloto. Estos cálculos están implementados a partir de la línea 246 del archivo *marker\_vision.h* que se ha incluido en el anexo.

#### 6. Envío al autopiloto:

La orientación y la posición son enviadas al autopiloto a través del protocolo *Mavlink* utilizando la librería *MAVSDK*

Mientras tanto, **en el autopiloto**, una vez que las recibe, este calcula la rotación entre su orientación expresada en ejes NED y su orientación expresada en el sistema de referencia de la visión, que en este caso es el del marcador.

$$R^{NED \rightarrow Marcador} = R^{NED \rightarrow UAV} \cdot (R^{Marcador \rightarrow UAV})^T \quad (3.7)$$

Esta matriz se utiliza para transformar la posición que le llega de la visión, expresándola en ejes NED (norte, este y abajo).

$$p^{NED' \rightarrow UAV} = R^{NED \rightarrow Marcador} \cdot p^{Marcador \rightarrow UAV} \quad (3.8)$$

Siendo  $NED'$  un sistema de referencia paralelo a  $NED$  pero centrado en el marcador. Que tengan esta orientación es importante, ya que el EKF en su fase de predicción, utilizando el acelerómetro y la orientación estimada, expresa su posición en ejes NED. Estos cálculos se pueden ver el código 3.1 donde se han extraído algunos fragmentos de PX4.

#### Código 3.1: Rotación en PX4 de la posición suministrada por la visión

En el archivo *ekf\_helper.cpp* se calcula la rotación que hay que aplicarle a la posición:

```
1460     const Quatf q_error(_state.quat_nominal *
1461                           _ev_sample_delayed.quat.inversed()).normalized();
1462     _R_ev_to_ekf = Dcmf(q_error);
```

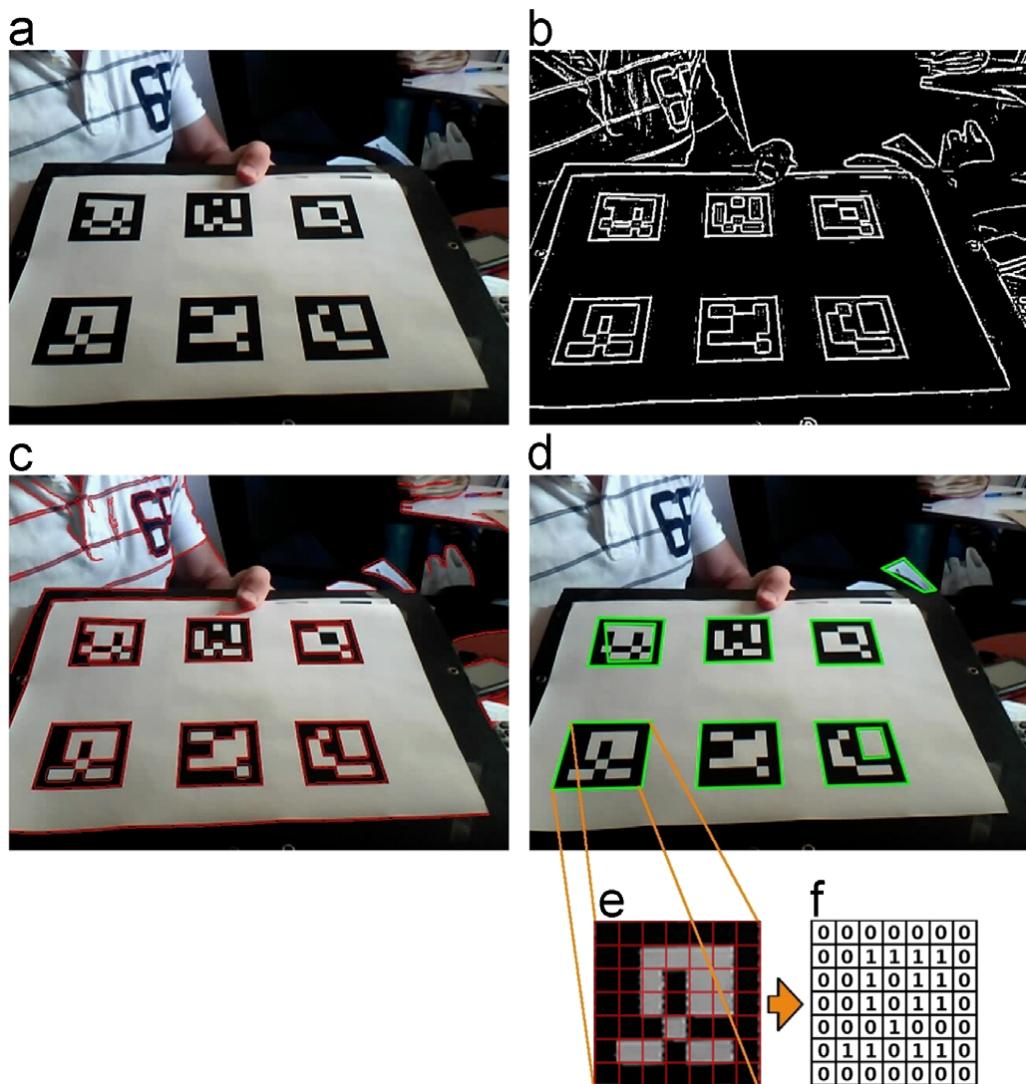
En el archivo *control.cpp* se aplica dicha rotación:

```
273     ev_pos_meas = _R_ev_to_ekf * ev_pos_meas;
274     ev_pos_var = _R_ev_to_ekf * ev_pos_var * _R_ev_to_ekf.transpose();
```

Con estas transformaciones ya se podría fusionar la medida de la visión. Así, se generarán unos estados estimados que serán tomados por los controladores de PX4. Como se explicó en [6], estos forman una estructura en cascada cuyo controlador de mayor nivel es el de posición.

### 3.3 Detección de marcadores Aruco

Los marcadores Aruco fueron creados por el departamento Aplicaciones de la Visión Artificial de la universidad de Córdoba. Estos pertenecen a la categoría de marcadores visuales planos y cuadrados, y su misión es ofrecer su posición relativa a la cámara además de identificarlo. Como se aprecia en la figura 3.4, estos son cuadrados que tienen en su interior unas figuras que codifican un número en binario. Esta zona interna solo se



**Figura 3.4** Pasos intermedios en el proceso de detección de marcadores Aruco. Fuente [7].

utiliza en la identificación, mientras que le borde exterior es el que se usa para la estimación de la posición. El proceso de detección está explicado en [7] pero aquí se hará un resumen del mismo. Los pasos principales son los siguientes:

- Detección de bordes:

Se denomina borde a la línea que separa 2 regiones, en el caso de los marcadores estas regiones son las zonas blancas y las negras. Aunque no es usado en este caso por su coste computacional, el algoritmo de Canny puede servir para explicar un método de detección de bordes. Quienes no estén familiarizados con el campo de la visión artificial, se pueden imaginar la imagen en escala de grises como un campo escalar de 2 dimensiones, y como tal, se le puede calcular el gradiente a cada punto del plano, que es un vector de dos dimensiones que indica la dirección en la que el campo varía más rápidamente y su módulo representa el ritmo de variación. Para distinguir un borde de cualquier otro punto este es útil, ya que en un punto del borde se cumple que su magnitud de gradiente es máximo local en la dirección del gradiente. En la subfigura 3.4b puede verse un ejemplo del resultado de este proceso.

- Detección de contornos:

En el paso anterior es posible que se hayan escogido bordes que no formen un contorno cerrado, por lo que se busca todos los contornos cerrados formados por los bordes y el resto se desechan (figura 3.4c). El contorno que se quiere obtener es el borde externo del marcador, cuya forma es un cuadrado y su proyección en la imagen será aproximadamente un cuadrilátero, por tanto se descartan todos los que no

se aproximen a un polígono de cuatro lados. Todavía puede haber contornos que no se correspondan a lo que se busca, por ejemplo los cuadrados que se encuentren dentro del marcador que se destinan a la identificación, así que se realiza otro cribado más que es el de eliminar los contornos interiores que estén contenidos en otros exteriores y que sean cercanos, quedando los contornos de la figura 3.4d.

- Identificación:

Cada uno de los contornos se rectifica para convertirlos en cuadrados y entonces la imagen se convierte en una cuadrícula. Como se puede ver en la figura 3.4f, en este ejemplo es de tamaño 7X7, donde cada celda se le asigna un 0 o un 1. De esta cuadrícula, en la que todas sus celdas más externas tienen que ser negras (si no se descarta), se extrae un código binario que en este caso es de 25 bits (quitando los bordes negros la cuadrícula se convierte en 5X5). Finalmente, se compara este código con todos los que componen el diccionario<sup>3</sup>, y si se da alguna coincidencia se da por válida la detección.

Existen más librerías de marcadores planos como *AprilTag*, cuyo método de detección es muy parecido a *Aruco* (de hecho la librería de Aruco puede detectar marcadores AprilTag). Una de las partes en las que sí que difieren es la generación de los diccionarios de marcadores. Un diccionario se le denomina a todos los códigos binarios diferentes que pueden tener los marcadores con un determinado número de bits. Este número de marcadores distintos no es igual a  $2^{\text{número de bits}}$  por varias razones. Primero, el marcador tiene que dar información de la rotación con respecto a la cámara y por lo tanto, no puede ser simétrico para que no se dé el caso de que existan dos rotaciones posibles en una sola imagen. Segundo, se debe de contemplar el caso de que exista ruido, y tal vez el código extraído no es el que realmente tiene el marcador. Entonces se reduce el número de posibles códigos de manera que no exista dos marcadores cuya diferencia solo esté en uno o pocos bits (maximizar distancia de Hamming). De esta manera, se evitan falsos positivos y se puede llegar a corregir los errores.

Una posible duda que puede surgir es si es necesario el paso de identificación cuando en una aplicación solo se usa un marcador, por ejemplo para el aterrizaje de precisión. Existen formas mucho más simples que pueden ser detectadas por la cámara como los tableros de ajedrez. Sin embargo, nada nos garantiza que estas formas la podamos encontrar también en el entorno, sobretodo en construcciones humanas, y que se produzca una detección no intencionada.

### 3.4 Metodología de la experimentación

Cuando se tratan problemas que tienen una implementación en el mundo real o se realizan simulaciones complejas, para llegar a la solución normalmente existen 2 etapas bien diferenciadas. La primera consiste en el **estudio teórico** del problema y en la planificación antes de realizar ningún experimento. En esta diseñamos un sistema y elegimos unos parámetros de acuerdo a expresiones analíticas o simulaciones. Después de realizar el primer experimento se llega a la etapa de **pruebas de validación**. En esta se realizan experimentos, se analizan los resultados y si no cumplen las especificaciones se vuelve a realizar el experimento con otro diseño o parámetros. Se podría imaginar un escenario en el que solo se pasara por una de las etapas, por ejemplo por la primera. Esto sería lo ideal, ya que cada parámetro o configuración está definida a priori y le respaldan los modelos matemáticos. Sin embargo, a veces el entorno real no es completamente predecible, los modelos no funcionan o simplemente te has equivocado con los cálculos. Yéndose al otro extremo, en el que no se realiza ningún estudio, pero se generan muchos experimentos, el primer problema que aparece es el valor inicial de los parámetros. También puede darse que los experimentos sean caros o que existe un riesgo de rotura del equipo. Además, surge la duda de cuales serán los siguientes parámetros a probar si no funciona el primer experimento ya que si no se conoce el sistema no se tiene una idea de qué efecto pueden provocar la modificación de los mismos. Por ejemplo, si se quisiera buscar los valores de los controladores PID de un quadrotor, aunque no se realicen cálculos para obtener un valor analítico, si se conoce su teoría de funcionamiento, la iteración de los parámetros se haría de una forma más acertada.

Dicho esto se puede concluir que no se pueden descuidar ninguna de las dos etapas, que hay que dedicarle tiempo tanto a la comprensión de un problema como a la realización de experimentos y a la capacidad de iterar rápidamente. En esta sección se va a explicar cómo se ha afrontado la etapa más experimental, en concreto sus pasos de iteración de parámetros y registro de resultados.

---

<sup>3</sup> Los diccionarios se definen a continuación

### 3.4.1 Iteración de los parámetros

En este problema hay muchos parámetros que frecuentemente se necesitan modificar:

- Propiedades de la cámara. A menudo se tiene que modificar el tiempo de exposición de la cámara dependiendo de la iluminación del entorno. Conviene escoger el mínimo valor con el que se obtenga una imagen iluminada, para que movimientos rápidos de la cámara no produzcan un emborronado de los contornos. También se puede escoger la resolución de la captura haciendo balance entre la rapidez de cálculo y la discretización espacial de la imagen
- Propiedades del marcador. Uno de los aspectos que hay que decidir del marcador es su tamaño, que determinará la altura óptima a la que el UAV volará sobre el marcador. En este trabajo no se detallará, pero existe la posibilidad de utilizar un tablero de marcadores<sup>4</sup> en lugar de uno solo para tener una estimación más precisa y permitir la oclusión. De estos hay que establecer el número de marcadores: a la vez que se capturen más marcadores en una imagen, mayor será la precisión, pero también se verán reducidos para un tamaño dado.
- Activación de funcionalidades. El programa se debe hacer lo más flexible posible, permitiendo por ejemplo que se pueda correr en un ordenador personal con un vídeo previamente grabado, en lugar de una cámara como fuente de visión. En este caso se debe de desactivar la comunicación con el autopiloto. Otra aspecto que cambia es la activación del guardado de resultados, ya que cuando el programa se esté ejecutando en el ordenador embebido esta tarea no conviene realizarse debido a lo lento que es escribir en el almacenamiento persistente (memoria SD).

En programación, los parámetros se pueden establecer de muchas maneras diferentes. La más básica de todas es estableciendo su valor en una constante en el código del programa. Esto tiene la desventaja de que cuando no se utiliza un lenguaje interpretado, hay que compilar el programa cada vez que se cambie un parámetro. Otra manera es mediante argumentos al llamar al programa por la línea de comandos. De esta manera no es necesario compilar cada vez que se toque un parámetro, su inconveniente es que hay que escribir todos los parámetros cada vez que se llame al programa, incluso aquellos que no han cambiado de una ejecución a otra. La forma que se ha utilizado en este trabajo es mediante un archivo de parámetros. Este es leído en tiempo de ejecución y su sintaxis es YAML. Se podría haber escogido otros formatos como el JSON, pero este no es tan leible para los humanos como el primero. Este archivo se puede ver en el anexo bajo el nombre de [vision\\_params.yml](#)

Otra posible solución más sofisticada, es la que se usa en el autopiloto PX4. Este ofrece una interfaz gráfica que se corre en la GCS y se comunica con el autopiloto. Entre sus funcionalidades destacan la indicación de que alguno se haya movido de su valor por defecto, sus valores máximos y mínimos, y su posible modificación en tiempo de vuelo. Esta última característica, que puede acelerar la elección de parámetros, lleva a poner en duda a llamar los parámetros como tales si se toma su definición de valores que no cambian a lo largo de un periodo largo de tiempo.

### 3.4.2 Registro de resultados (*logging*)

Para analizar los resultados primero hay que registrarlos. A continuación, se comenta algunas de las funcionalidades implementadas relacionadas con el *logging*:

- Guardado de la posición y orientación estimadas en un archivo y su representación temporal:

Para verificar el desempeño de la estimación de la posición y orientación, lo ideal sería tener un groundtruth, por ejemplo mediante un sistema de visión como *OptiTrack*. En este caso no se tiene y lo que nos queda es inspeccionar los resultados de manera visual, que es suficiente para hallar muchos de los errores de la estimación. Hay que tener en cuenta que se tiene un sistema dinámico y ni la posición ni la velocidad pueden cambiar bruscamente, por lo tanto, si al inspeccionar las gráficas temporales de la posición y orientación esto sucede, probablemente se trate de una estimación errónea.

- Guardar las imágenes de la cámara con los ejes del marcador superpuestos (realidad aumentada):

Otra forma de verificación es la de ver los ejes del marcador superpuestos en la imagen (figura 3.5). Resulta fácil de inspeccionar si estos se encuentran en el centro del marcador, que es donde se sitúa su sistema de referencia. Además, sus ejes *x* e *y* deben de ser paralelos a los bordes del papel y su eje *z* perpendicular a él. En la imagen también aparece superpuesto un rectángulo que rodea al marcador, que resulta útil para comprobar que la detección de sus esquinas se realiza de manera correcta.

<sup>4</sup> Mas información sobre los tableros de marcadores en [https://docs.opencv.org/4.5.0/d7/d4a/tutorial\\_charuco\\_detection.html](https://docs.opencv.org/4.5.0/d7/d4a/tutorial_charuco_detection.html)



**Figura 3.5** Superposición de los ejes de referencia del marcador y del cuadrilátero que lo rodea (trazado en verde).

- Scripts en lenguaje *Python* de inicio y apagado:

A pesar de ser un lenguaje de ejecución rápida, C++ suele ser más difícil para el desarrollador. En cambio Python es un lenguaje que necesita menos líneas de código para hacer lo mismo, tiene una sintaxis más simple y una cantidad enorme de librerías. Por esta razón, en el programa principal escrito en C++ se han hecho llamadas a scripts de Python en su arranque y finalización, ya que estos son los momentos en los que es menos crítico el tiempo de ejecución. En concreto, una de las tareas de estos, es la de crear una carpeta cuyo nombre es la fecha y hora, donde se guardarán los archivos que se han visto en los puntos anteriores. Además, en la finalización del programa se guardan los parámetros elegidos en dicha carpeta y se genera un archivo de vídeo a partir de todas las imágenes tomadas por cámara que se han estado guardando. El guardado de sucesivos experimentos en carpetas distintas es bastante **útil para hacer comparaciones**.

Todos las pruebas anteriores son para evaluar el programa de detección de marcadores de manera aislada. Si se quiere comprobar la fusión de la medida y la posición estimada que se genera finalmente en el autopiloto, hay que utilizar las herramientas de PX4. Este aprovecha el método de comunicación entre sus módulos, que es mediante mensajes de publicador y suscriptor, para guardar estos en una memoria persistente. Si se tiene una tarjeta SD suficientemente rápida, se puede guardar los mensajes a la misma frecuencia de actualización del autopiloto (400Hz) sin perder ninguno. Toda la información, tanto parámetros como mensajes intercambiados entre los módulos o alertas, se almacena en un archivo binario en un formato llamado *ulog*. Existen varias herramientas para representar su contenido, por ejemplo, la que se ha usado aquí ha sido *FlightPlot*. Estos archivos tienen otra utilidad para los desarrolladores que se denomina *System-wide replay*, que es utilizada para correr el autopiloto en el ordenador personal, pero en lugar de simular un vuelo, se le aplican los estímulos guardados por el archivo *ulog*. La gran ventaja de esto es activar la ejecución paso a paso y acceder a todas aquellas variables intermedias que serían más difícil de obtener cuando el vehículo estuviera volando.

### 3.5 Resultados experimentales

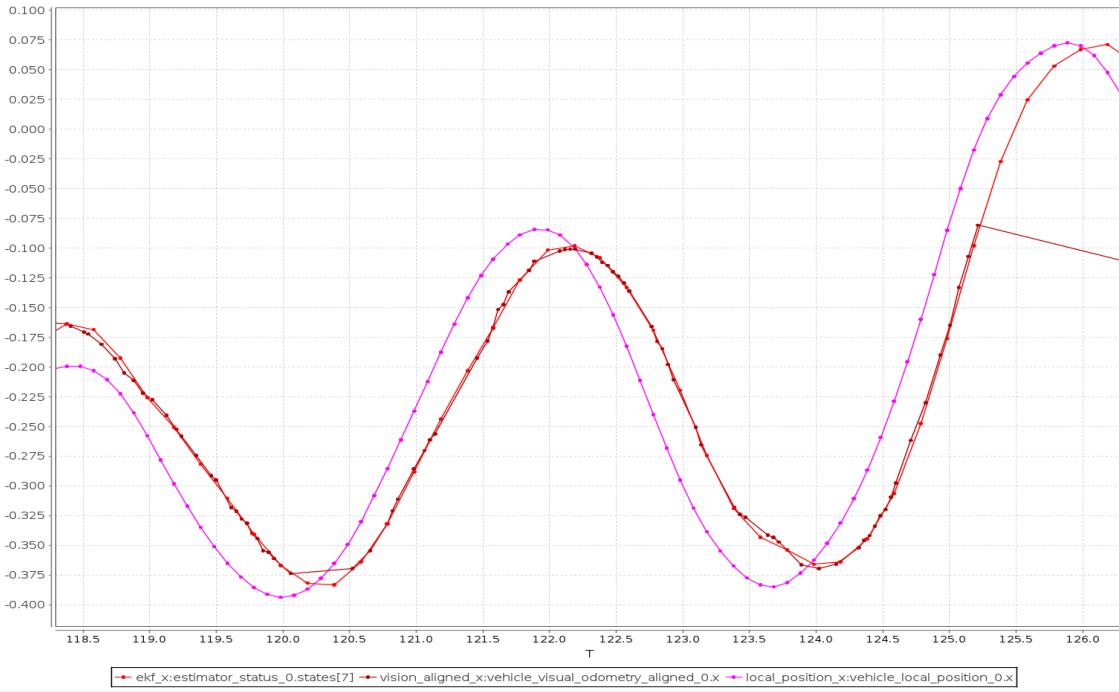
En esta sección se mostrarán algunas gráficas<sup>5</sup> de los datos generados en un vuelo donde el quadrotor volaba sobre un marcador y el autopiloto recibía las medidas de la visión, y en el que estaban desactivados tanto el GNSS como el flujo óptico.

En primer lugar, se tuvo activado el modo *Altitude*, que es aquel en el que el mando genera una orientación y una altura de referencia, por lo tanto la posición de la visión no se estaba usando todavía por los controladores. En la figura 3.6a, se observa el retraso que hay entre el filtro de salida y el EKF. Se puede medir que este retraso es de aproximadamente un cuarto de segundo, que es justo el retraso que se ha indicado para la visión a la hora de realizar el manejo de medidas retrasadas, explicado en el capítulo anterior. En la figura 3.6b ocurre que en varias ocasiones falta la medida de la visión, probablemente porque el marcador haya salido fuera del campo de visión de la cámara. Se puede observar que mientras esta no se recibe, el EKF sigue

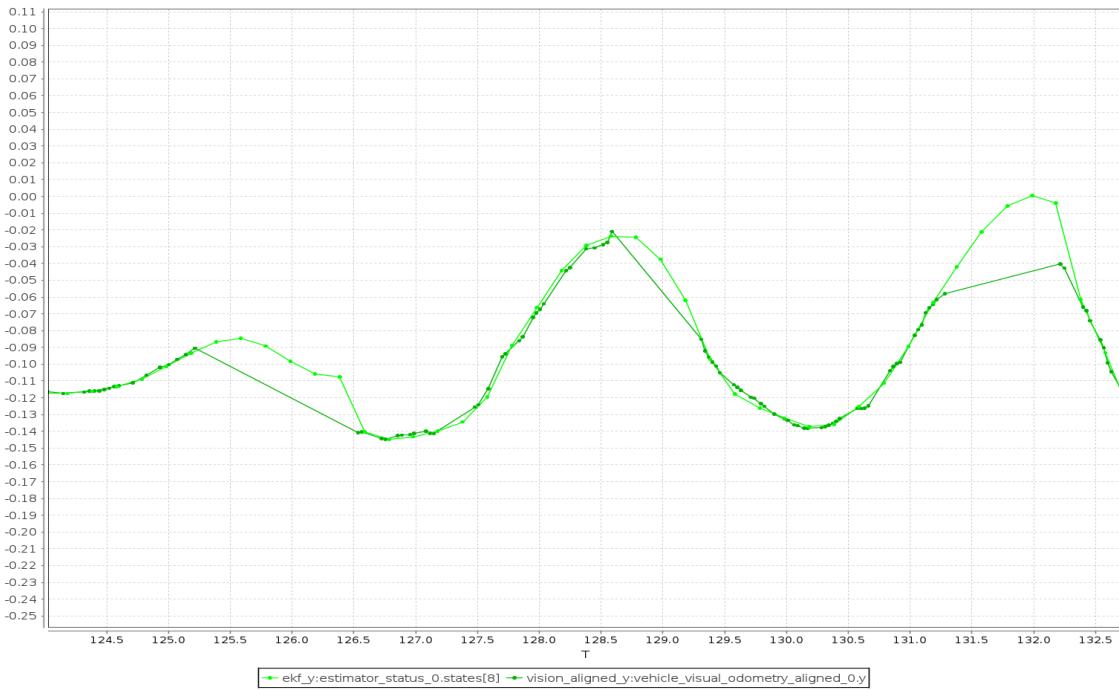
<sup>5</sup> Todas las gráficas de este vuelo se pueden ver en [https://review.px4.io/plot\\_app?log=0dec276d-c5d0-4f6a-8c57-be72523f9605](https://review.px4.io/plot_app?log=0dec276d-c5d0-4f6a-8c57-be72523f9605)

estimando gracias a su fase de predicción y también se puede comprobar, que **la predicción no se aleja** mucho de la medida de la visión cuando esta se recupera.

Unos segundos más adelante, se activó durante 10 segundos el modo *Position*, que es aquel en el que se genera una posición de referencia constante si las palancas de los mandos están en su posición de reposo. En la figura 3.7a se observa que la diferencia entre las posiciones más extremas ronda los 5 centímetros. Hay que destacar que al quadrotor le resulta imposible quedarse en un punto fijo sin estas medidas de la visión, ya que no tiene otra fuente de velocidad ni posición absoluta. Esta idea se puede verificar en la figura 3.7b donde se ve que, para quedarse inmóvil, necesita mantener una inclinación que no es distinta de cero, o bien por el efecto del viento, o bien porque el autopiloto no se encuentre en el plano de los ejes de las hélices.

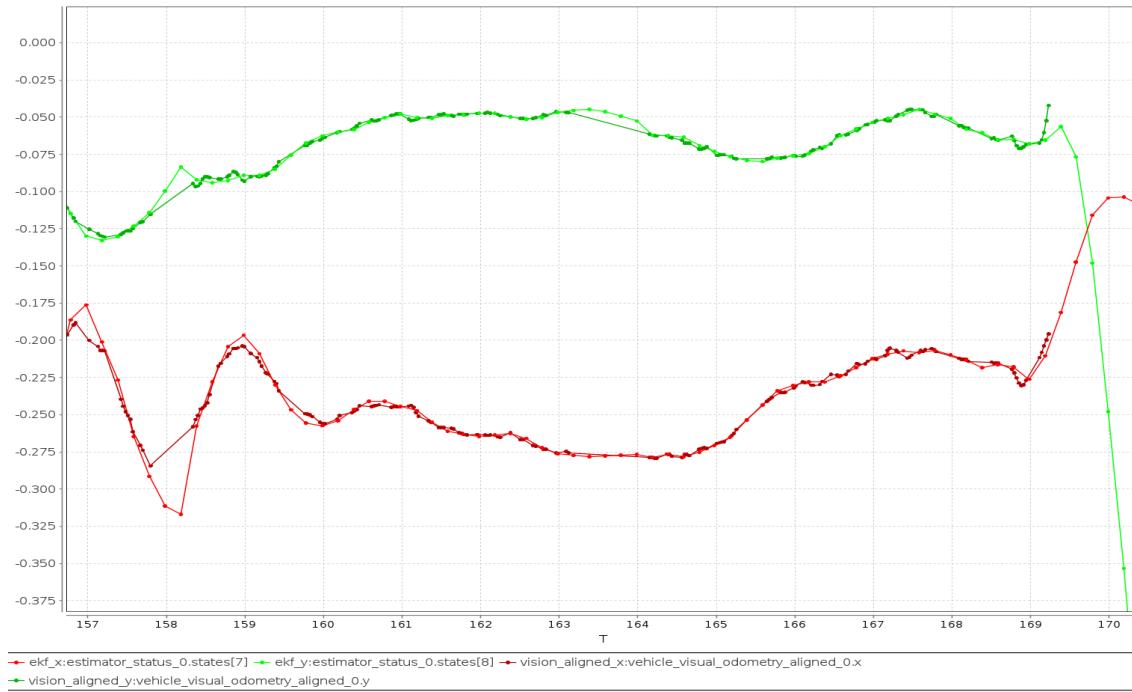


(a) Se muestran: la posición en el eje  $x$  generada por la visión (rojo) , la posición estimada por el EKF (rojo oscuro) y la posición generada por el filtro de salida (morado).

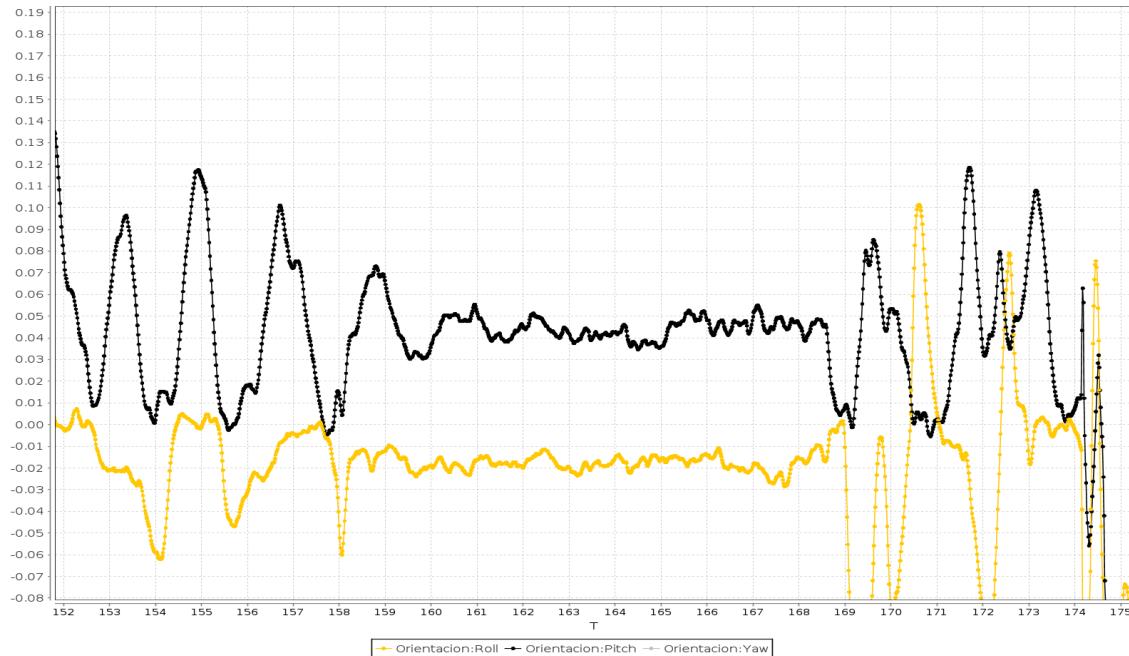


(b) Se muestran: la posición en el eje  $y$  generada por la visión (verde) y la posición estimada por el EKF (verde oscuro).

**Figura 3.6** Modo *altitude*.

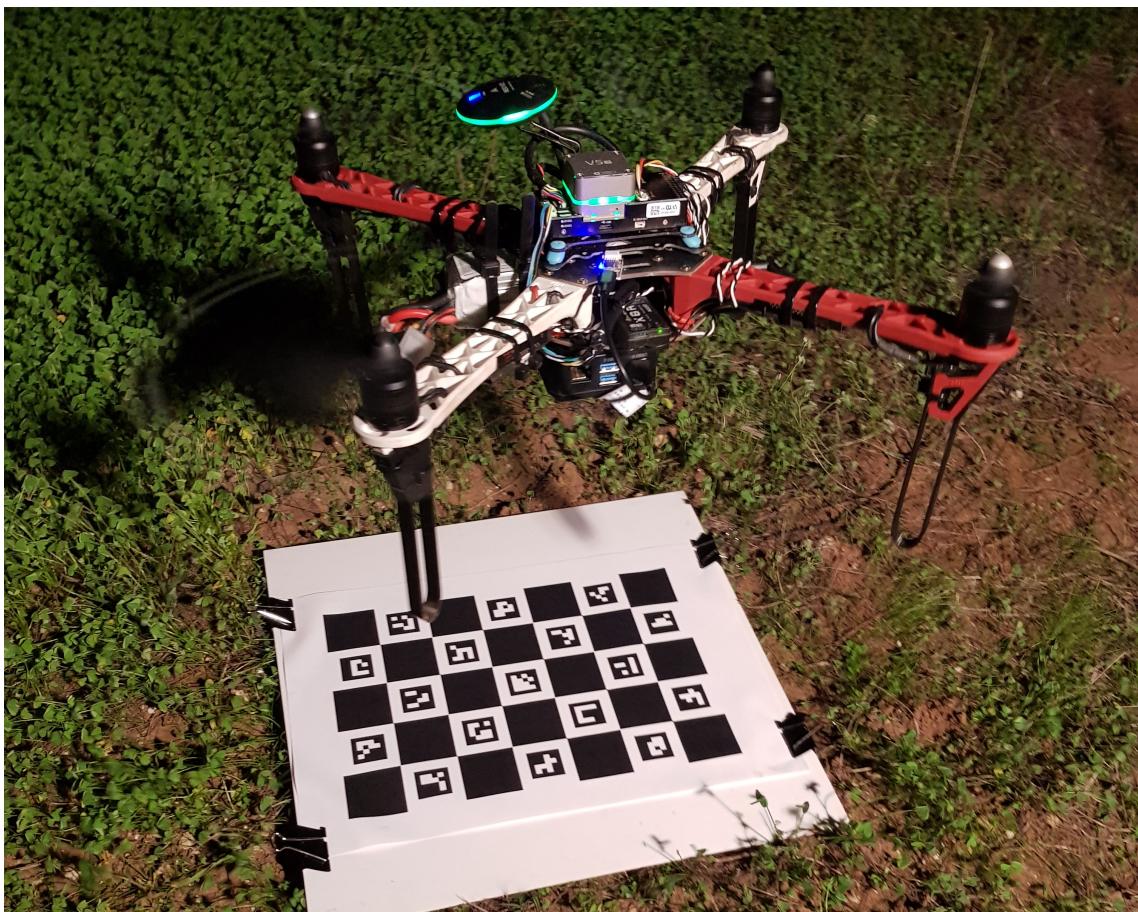


(a) Se muestran: la posición en el eje  $x$  generada por la visión (rojo) y por el EKF (rojo oscuro); la posición en el eje  $y$  generada por la visión (verde) y por el EKF (verde oscuro).



(b) Ángulos de euler roll y pitch medidos en grados sexagesimales.

**Figura 3.7** Modo *position* desde el instante 158s hasta el 168s.



**Figura 3.8** Quadrotor sobrevolando un tablero de marcadores.

## 4 Conclusiones

---

Los resultados mostrados en este trabajo sirven sacar numerosas conclusiones. Para empezar, el **manejo de medidas retrasadas** aquí explicado, es un método que no requiere de complejas demostraciones para predecir que va a funcionar, ya que es intuitivo. Aun así, se ha probado en una simulación la ventaja que supone tenerlo. Una simulación que se ha programado en un lenguaje de programación abierto, que no tiene ningún coste económico, al contrario que el ampliamente usado *Matlab*, y que además es de propósito más general.

En cuanto al uso de **marcadores visuales para estimar la posición** claramente es una de las opciones más baratas para el posicionamiento, además de ser muy precisa. El ordenador embebido escogido, a pesar de ser de los más económicos, se ha comprobado que tiene capacidad de cómputo suficiente para esta tarea, ya que es capaz de aportar medidas cada 20 milisegundos, que suele ser suficiente para alimentar a un controlador de posición. Por otro lado, tiene una limitación importante: allí donde se navegue se necesita que haya marcadores. Por muy baratos que sea imprimirlos, si se quiere un vehículo completamente autónomo en todos los entornos, no es factible llenar el mundo de marcadores o balizas. Sin embargo, estos podrían servir de apoyo a otras tecnologías que no necesitan ningún tipo de instalación fuera del vehículo.

Una de las ideas de este trabajo que no se ha visto en otros, es la de utilizar el estimador de estados del autopiloto para fusionar medidas de la visión, además del controlador de posición también del propio autopiloto. La ventaja que tiene esta forma frente a aquellos que ubican estos elementos en un ordenador separado, es que se reduce el efecto del retraso en las comunicaciones. La parte perjudicial, es que si las medidas de la posición no son correctas, pueden llegar a afectar a la estimación de la orientación del vehículo, ya que en un filtro de Kalman todos los estados pueden estar interrelacionados. Por otra parte, se ha visto que el filtro de Kalman puede ayudar en aquellos momentos en los que se pierde la visión del marcador, llegando a predecirla de manera acertada si no transcurre mucho tiempo sin llegar una nueva medida de la posición.



## 5 Trabajos futuros

---

Existe una gran oportunidad en seguir construyendo sobre la base que presenta este trabajo, ya que la posibilidad de conseguir un posicionamiento preciso, libera muchas de las tareas que hasta ahora necesitaban de un piloto para llevarse a cabo. A continuación, enumero algunos de los próximos pasos que serían interesantes de tomar en próximos proyectos:

- Conseguir la navegación en distancias más grandes únicamente usando marcadores. El número de marcadores que se sería necesario colocar por el camino vendría limitado por el campo de visión de la cámara. Para evitar llenar el espacio de ellos, se podrían intentar algunas soluciones alternativas como utilizar lentes ojo de pez, colocar más de una cámara, o montar la cámara sobre una plataforma articulada (gimbal).
- Cuantificar en tiempo real la incertidumbre de la estimación de la posición mediante marcadores. Este tipo de medida, es un caso de aquellas que varían mucho su precisión en poco tiempo, al igual que la exactitud del GNSS depende del número de satélites que detecte. En este trabajo se ha supuesto que la precisión de la visión tiene un valor constante de un centímetro, cuando en realidad depende fuertemente de factores como la distancia entre el marcador y la cámara. Esta incertidumbre se podría cuantificar, por ejemplo, mediante el error de reproyección mencionado en la sección 3.2.
- Generar marcadores más discretos o incluso, que puedan ser confundidos con elementos decorativos.
- Utilizar otros sensores que aporten posición o velocidad para que se siga conociendo la posición aún cuando no haya marcadores, como el GNSS o el flujo óptico. Uno de los retos que plantea son las transiciones entre las diferentes fuentes de posición o la presencia de varias de ellas al mismo tiempo.
- Aprovechar la precisión de la posición para **manipular objetos**. Si no se generasen muchas interferencias al magnetómetro, se podría utilizar un electroimán para tomar y depositar objetos. Este podría colgarse del UAV mediante varios hilos, ya que con uno solo la carga se balancearía.
- Tomar imágenes en el vuelo, para luego unirlas y crear un mapa donde se puedan especificar los waypoints.
- Crear un nuevo **generador de trayectorias**. El que hay implementado en PX4 consiste en especificar waypoints en un mapa de imágenes satelitales. Para especificar trayectorias de manera más precisa, se podría grabar las posiciones que recorre el vehículo en un modo manual y luego repetirlas de manera automática, de una forma similar a la programación de brazos manipuladores.



# Apéndice A

## Simulador del estimador de estados

El siguiente archivo también puede verse y descargarse en el repositorio [https://github.com/isidroas/quadrrotor\\_simulator](https://github.com/isidroas/quadrrotor_simulator)

```
1 #!/bin/env python3
2 import numpy as np
3 from numpy.random import randn
4 import matplotlib.pyplot as plt
5 import matplotlib
6 from scipy.ndimage.interpolation import shift
7
8 import time
9 from pdb import set_trace
10 import os
11 import datetime
12
13 # Parameters
14 DATA_L = 1000
15 MASS = 1 # Kg
16 G_CONSTANT = 9.8 # m/s^2
17 INERTIA = MASS * 0.45 ** 2 / 12 # Kg.m^2
18 DT = 0.01 # s # Reducir el paso mejora la precisión de la predicción, aunque haya ruido
19
20 ACCEL_NOISE = 0.25 # m/s^2
21 GYRO_NOISE = 0.03 # rad/s
22 GPS_NOISE= 0.01 # m
23 GPS_DELAY = 1 # s
24 GPS_PERIOD = 0.3 # s
25
26 # Plot flags
27 DRAW_ESTIMATED = True
28 SHOW_ANIMATED = False
29 SHOW_PLOTS = False
30 SHOW_OUTPUT_CORRECTION = False
31 SHOW_OUTPUT_DELAYED= False
32 SHOW_OUTPUT_ERROR= True
33 IMAGE_FOLDER = "images/"
34 IMAGE_FOLDER = "n_update/"
35 #IMAGE_FOLDER = "no_handle_delay/"
36 #IMAGE_FOLDER = "handle_delay/"
37 IMAGE_EXTENSION = "pdf"
38 SHOW_GPS = False
39
40 # Fusion flags
41 TAU_P = 0.5
42 TAU_V = 0.01
43 TAU_THETA = 0.0
44 FUSE_GPS = False
45 HANDLE_DELAYS = False
46 OUTPUT_CORRECTION = False
47
48
49
```

```

50  # Control se realiza sobre los estados reales para acotar más el efecto del estimador
51  def control_actuators(
52      theta: float, thetad: float, theta_ref: float, yd_e: float
53  ) -> [float, float]:
54      # Control gains
55      K_height = 2
56      K_tilt = 0.2
57      Kd_tilt = 0.1
58      thrust = MASS * G_CONSTANT / np.cos(theta) + yd_e * K_height
59      torque = (theta_ref - theta) * K_tilt - thetad * Kd_tilt
60      return thrust, torque
61
62
63
64  # Jacobianos de los modelos de observación
65  H_gps = np.zeros((2, 5))
66  H_gps[0, 0] = 1
67  H_gps[1, 1] = 1
68  R_gps = np.diag([GPS_NOISE ** 2, GPS_NOISE ** 2])
69
70  def output_filter(
71      p_prev, v_prev, theta_prev, accel, gyro
72  ) -> list:
73
74      if np.isnan(p_prev[0]):
75          # we need to initialize
76          p_prev = np.array([0,0])
77          v_prev = np.array([0,0])
78          theta_prev = 0
79
80          theta_pred = theta_prev + DT * gyro
81          #c = np.cos(theta_pred)
82          #s = np.sin(theta_pred)
83          # TODO: utilizar aquí el predicho ahora o el estimado anterior?
84          c = np.cos(theta_prev)
85          s = np.sin(theta_prev)
86          rot_mat = np.array([[c, -s], [s, c]])
87          v_pred = v_prev + DT * rot_mat @ accel
88          p_pred = p_prev + DT * v_prev
89          return p_pred, v_pred, theta_pred
90
91
92  def ekf_estimator(
93      p_prev, v_prev, theta_prev, cov_mat_prev, accel, gyro, gps=None
94  ) -> list:
95
96      if gyro is None:
97          # we can't predict states without IMU measurements
98          return [None]*4
99
100     if np.isnan(p_prev[0]):
101         # we need to initialize
102         p_prev = np.array([0,0])
103         v_prev = np.array([0,0])
104         theta_prev = 0
105         cov_mat_prev = np.zeros([5,5])# TODO: esto está bien?
106
107         # Predicción de los estados
108         theta_pred = theta_prev + DT * gyro
109         #c = np.cos(theta_pred)
110         #s = np.sin(theta_pred)
111         # TODO: utilizar aquí el predicho ahora o el estimado anterior?
112         c = np.cos(theta_prev)
113         s = np.sin(theta_prev)
114         rot_mat = np.array([[c, -s], [s, c]])
115         v_pred = v_prev + DT * rot_mat @ accel
116         p_pred = p_prev + DT * v_prev
117
118         # Predicción de la matriz de covarianzas
119         x_pred = np.array([p_pred[0], p_pred[1], v_pred[0], v_pred[1], theta_pred])

```

```

120     F = np.array(
121         [
122             [1, 0, DT, 0, 0],
123             [0, 1, 0, DT, 0],
124             [
125                 0,
126                 0,
127                 1,
128                 0,
129                 DT * (-accel[0] * np.sin(theta_prev) + accel[1] * np.cos(theta_prev)),
130             ],
131             [
132                 0,
133                 0,
134                 0,
135                 1,
136                 DT * (-accel[0] * np.cos(theta_prev) - accel[1] * np.sin(theta_prev)),
137             ],
138             [0, 0, 0, 0, 1],
139         ]
140     )
141     G = np.array(
142         [
143             [0, 0, 0],
144             [0, 0, 0], # que pasaría si desarollo v(a) aquí?
145             [DT * c, DT * s, 0],
146             [-DT * s, DT * c, 0],
147             [0, 0, DT],
148         ]
149     )
150     Q = (
151         G
152         @ np.diag([ACCEL_NOISE ** 2, ACCEL_NOISE ** 2, GYRO_NOISE ** 2])
153         @ np.transpose(G)
154     )
155     cov_mat_est = F @ cov_mat_prev @ np.transpose(F) + Q
156
157     x_est = x_pred
158     p_est = x_est[0:2] # Remind slices x:y doesn't include y
159     v_est = x_est[2:4]
160     theta_est = x_est[4]
161     cov_mat = cov_mat_est
162
163     ### Update
164
165     ## gps
166     if FUSE_GPS and not np.isnan(gps).any():
167         innov = gps - p_est
168         S_gps = H_gps @ cov_mat @ np.transpose(H_gps) + R_gps
169         K_f = cov_mat @ np.transpose(H_gps) @ np.linalg.inv(S_gps)
170         x_est = x_est + K_f @ innov
171         p_est = x_est[0:2] # Remind slices x:y doesn't include y
172         v_est = x_est[2:4]
173         theta_est = x_est[4]
174         cov_mat = cov_mat - K_f @ H_gps @ cov_mat
175
176     return p_est, v_est, theta_est, cov_mat
177
178
179 def draw_animation(x, y, theta):
180     import numpy as np
181     import matplotlib.pyplot as plt
182     from matplotlib.animation import FuncAnimation
183
184     fig, ax = plt.subplots()
185     xdata, ydata = [], []
186     ln1, = plt.plot([], [], "r")
187     ln2, = plt.plot([], [], "b")
188
189     def init():

```

```

190     margin = 2
191     ax.set_xlim(min(x) - margin, max(x) + margin)
192     ax.set_ylim(min(y) - margin, max(y) + margin)
193     ax.set_aspect("equal")
194     return (ln,)
195
196 def update(frame):
197     xdata.append(x[frame])
198     ydata.append(y[frame])
199     ln.set_data(xdata, ydata)
200     c = np.cos(theta[frame])
201     s = np.sin(theta[frame])
202     rot_mat = np.array([[c, -s], [s, c]])
203     p1 = [-0.5, 0]
204     p2 = [0.5, 0]
205     p1_rot = rot_mat @ p1
206     p2_rot = rot_mat @ p2
207     ln2.set_data(
208         [p1_rot[0], p2_rot[0]] + x[frame], [p1_rot[1], p2_rot[1]] + y[frame]
209     )
210     return ln, ln2
211
212 ani = FuncAnimation(
213     fig,
214     update,
215     frames=len(x),
216     init_func=init,
217     blit=True,
218     interval=DT * 1e3,
219     repeat=False,
220 )
221 plt.show()
222
223
224 def main():
225     print("-----")
226     print("Simulador quadrotor")
227     print("-----")
228
229     # Actuation signals
230     thrust = np.ones(DATA_L) * MASS * G_CONSTANT
231     torque = np.zeros(DATA_L)
232
233     # translational variables
234     a = np.zeros((2, DATA_L))
235     v = np.zeros((2, DATA_L))
236     p = np.zeros((2, DATA_L))
237
238     # angular variables. Initialized in zero
239     theta = np.zeros(DATA_L)
240     thetad = np.zeros(DATA_L)
241     thetadd = np.zeros(DATA_L)
242
243     # sensores
244     accel = np.zeros((2, DATA_L))
245     accel_gt = np.zeros((2, DATA_L))
246     gyro = np.zeros(DATA_L)
247     gps = np.empty((2, DATA_L)) * np.nan
248
249     # Setpoints
250     yd_ref = np.zeros(DATA_L)
251     theta_ref = np.zeros(DATA_L)
252     yd_ref[: int(DATA_L * 0.25)] = 2
253     theta_ref[int(DATA_L * 0.70) : int(DATA_L * 0.85)] = np.pi / 6
254     theta_ref[int(DATA_L * 0.85) :] = -np.pi / 6
255
256     t = np.array(list(range(DATA_L))) * DT
257
258     # Simulate 2 newton law
259     # Simular despegue y avance dibujando el suelo

```

```

260     for i in range(1, DATA_L): # Pass states are needed, so we start at second
261         # Control actuators
262         thrust[i], torque[i] = control_actuators(
263             theta[i - 1], thetad[i - 1], theta_ref[i], yd_ref[i] - v[1, i - 1]
264         )
265
266         # Rotational dynamics
267         thetadd[i] = torque[i] / INERTIA
268         thetad[i] = (
269             thetad[i - 1] + DT * thetadd[i]
270         ) # TODO: test trapezoidal integration
271         theta[i] = theta[i - 1] + DT * thetad[i]
272
273         # Rotation matrix. Transform body coordinates to inertial coordinates
274         c = np.cos(theta[i])
275         s = np.sin(theta[i])
276         rot_mat = np.array([[c, -s], [s, c]])
277
278         # Translational dynamics
279         thrust_rot = rot_mat @ np.array([0, thrust[i]])
280         gravity_force = np.array([0, -G_CONSTANT]) * MASS
281         a[:, i] = (thrust_rot + gravity_force) / MASS
282         v[:, i] = v[:, i - 1] + DT * a[:, i]
283         p[:, i] = p[:, i - 1] + DT * v[:, i]
284
285         # simulate sensors
286         accel_gt[:, i] = np.linalg.inv(rot_mat) @ a[:, i]
287         accel[:, i] = (
288             accel_gt[:, i] + randn(2) * ACCEL_NOISE
289         ) # TODO: Habría que multiplicarlo por la inversa de rot_mat?
290         gyro[i] = thetad[i] + randn(1) * GYRO_NOISE
291
292         if i > GPS_DELAY / DT:
293             gps[:, i] = p[:, int(i - GPS_DELAY / DT)] + randn(2) * GPS_NOISE
294         else:
295             gps[:, i] = None
296
297         if GPS_PERIOD!=0:
298             if i%(GPS_PERIOD/DT)!=0:
299                 gps[:, i] = None
300
301
302         ### Estimación de los estados ###
303         # States at delayed time horizon
304         v_est = np.empty((2, DATA_L))*np.nan
305         p_est = np.empty((2, DATA_L))*np.nan
306         theta_est = np.empty(DATA_L)*np.nan
307
308         # States at current time horizon
309         v_output = np.empty((2, DATA_L))*np.nan
310         p_output = np.empty((2, DATA_L))*np.nan
311         theta_output = np.empty(DATA_L)*np.nan
312
313         # Output states delayed. Only for logging
314         v_output_del = np.empty((2, DATA_L))*np.nan
315         p_output_del = np.empty((2, DATA_L))*np.nan
316         theta_output_del = np.empty(DATA_L)*np.nan
317
318         # Corrección aplicada al filtro de salida
319         v_correction = np.empty((2, DATA_L))*np.nan
320         p_correction = np.empty((2, DATA_L))*np.nan
321         theta_correction = np.empty(DATA_L)*np.nan
322
323         # Matriz de covarianzas
324         P_est = np.empty((5, 5, DATA_L))*np.nan
325
326         # Buffer de medidas
327         max_delay = max(GPS_DELAY, 0)
328         #buffer_size = int(max_delay / DT) # TODO: redondear hacia arriba?
329         buffer_size = int(max_delay / DT) + 1

```

```

330     buffer_ekf = [{}]*buffer_size
331     # gps_insert_pos = int(GPS_DELAY / DT) - 1
332     gps_insert_pos = int(GPS_DELAY / DT)
333
334     # Buffer de estados
335     buffer_output = [{}]*buffer_size
336
337     for i in range(1, DATA_L):
338
339         ## Fill buffer
340         # Sensors with no delay (pos 0)
341         buffer_ekf.insert(0, {"accel": accel[:, i], "gyro": gyro[i]})  

342         # gps
343         buffer_ekf[gps_insert_pos]["gps"] = gps[:, i]
344
345         ## Pop buffer
346         delayed_meas = buffer_ekf.pop()
347         accel_delayed = (
348             delayed_meas["accel"] if "accel" in delayed_meas.keys() else None
349         )
350         gyro_delayed = delayed_meas["gyro"] if "gyro" in delayed_meas.keys() else None
351         gps_delayed = delayed_meas["gps"] if "gps" in delayed_meas.keys() else None
352
353         [p_est[:, i], v_est[:, i], theta_est[i], P_est[:, :, i]] = (
354             ekf_estimator(
355                 p_est[:, i - 1],
356                 v_est[:, i - 1],
357                 theta_est[i - 1],
358                 P_est[:, :, i - 1],
359                 accel[:, i],
360                 gyro[i],
361                 gps=gps[:, i],
362             )
363             if not HANDLE_DELAYS
364             else ekf_estimator(
365                 p_est[:, i - 1],
366                 v_est[:, i - 1],
367                 theta_est[i - 1],
368                 P_est[:, :, i - 1],
369                 accel_delayed,
370                 gyro_delayed,
371                 gps=gps_delayed,
372             )
373         )
374
375         ## Output filter
376         [p_output[:, i], v_output[:, i], theta_output[i]] = output_filter(
377             p_output[:, i - 1],
378             v_output[:, i - 1],
379             theta_output[i - 1],
380             accel[:, i],
381             gyro[i],
382         )
383         # Fill output buffer
384         buffer_output.insert(0, {"p": p_output[:, i], "v": v_output[:, i], "theta": theta_output[i]}
385         ↵ )
386
387         ## Corrección
388
389         # Pop buffer
390         delayed_state = buffer_output.pop()
391
392         if "p" in delayed_state.keys() and OUTPUT_CORRECTION:
393             p_delayed = delayed_state["p"]
394             v_delayed = delayed_state["v"]
395             theta_delayed = delayed_state["theta"]
396
397             p_error = p_est[:, i] - p_delayed
398             v_error = v_est[:, i] - v_delayed
399             theta_error = theta_est[i] - theta_delayed

```

```

400     for index, elem in enumerate(buffer_output):
401         # TODO: improve control
402         buffer_output[index]["p"] = buffer_output[index]["p"] + p_error* TAU_P
403         buffer_output[index]["v"] = buffer_output[index]["v"] + v_error* TAU_V
404         buffer_output[index]["theta"] = buffer_output[index]["theta"] + theta_error*
405             ↪ TAU_THETA
406
407         p_output[:, i] = buffer_output[0]["p"]
408         v_output[:, i] = buffer_output[0]["v"]
409         theta_output[i] = buffer_output[0]["theta"]
410
411         p_output_del[:,i] = p_delayed
412         v_output_del[:,i] = v_delayed
413         theta_output_del[i] = theta_delayed
414
415         p_correction[:,i] = p_error * TAU_P
416         v_correction[:,i] = v_error * TAU_V
417         theta_correction[i] = theta_error * TAU_THETA
418
419     # create result folders
420     if IMAGE_EXTENSION=="pdf":
421         results_path = IMAGE_FOLDER + os.sep
422     else:
423         subdir_name = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
424         results_path = IMAGE_FOLDER + subdir_name + os.sep
425
426     os.makedirs(results_path)
427
428     # save parameters
429     with open(results_path + "parametros.txt", "w") as f:
430         f.write(str(globals()))
431
432     # Activate global grid
433     plt.rcParams['axes.facecolor'] = 'white'
434     plt.rcParams['axes.edgecolor'] = 'white'
435     plt.rcParams['axes.grid'] = True
436     plt.rcParams['grid.alpha'] = 1
437     plt.rcParams['grid.color'] = "#cccccc"
438
439     plt.rcParams.update({'font.size': 16})
440     plt.rcParams.update({'font.weight': 'bold'})
441
442     matplotlib.rcParams['font.family'] = 'serif'
443     matplotlib.rcParams['font.serif'] = 'Computer Modern Roman'
444     matplotlib.rcParams["text.usetex"] = True
445
446     # Plot results
447     fig, ax = plt.subplots()
448     ax.set_title("Posición X")
449     if DRAW_ESTIMATED:
450         ax.plot(t, p_est[0, :], label="$P_x$ EKF", color="tab:orange")
451         if OUTPUT_CORRECTION:
452             ax.plot(t, p_output[0, :], label="$P_x$ Salida", color="tab:green")
453             if SHOW_OUTPUT_CORRECTION:
454                 ax.plot(t, p_correction[0, :], label="$P_x$ correction")
455             if SHOW_OUTPUT_DELAYED:
456                 ax.plot(t, p_output_del[0, :], label="$P_x$ output delayed", color="tab:purple",
457                     ↪ linestyle="--")
458         if SHOW_GPS:
459             ax.scatter(t, gps[0, :], label="$P_x$ GPS", marker="x", color="tab:green")
460             ax.plot(t, p[0, :], label="$P_x$ Groundtruth", color="tab:blue", linestyle="--")
461             plt.xlabel("t (s)")
462             plt.ylabel("$P_x$ (m)")
463             ax.legend()
464             plt.savefig(results_path + "x_t." + IMAGE_EXTENSION)
465
466     fig, ax = plt.subplots()
467     ax.set_title("Posición Y")
468     if DRAW_ESTIMATED:
469         ax.plot(t, p_est[1, :], label="$P_y$ EKF", color="tab:orange")

```

```

469     if OUTPUT_CORRECTION:
470         ax.plot(t, p_output[1, :], label="$P_y$ Salida", color="tab:green")
471         if SHOW_OUTPUT_CORRECTION:
472             ax.plot(t, p_correction[1, :], label="$P_y$ correction")
473         if SHOW_OUTPUT_DELAYED:
474             ax.plot(t, p_output_del[1, :], label="$P_y$ output delayed", color="tab:purple",
475                     linestyle="--")
476     if SHOW_GPS:
477         ax.scatter(t, gps[1, :], label="$P_y$ GPS", marker="x", color="tab:green")
478         ax.plot(t, p[1, :], label="$P_y$ Groundtruth", color="tab:blue", linestyle="--")
479     plt.xlabel("t (s)")
480     plt.ylabel("$P_y$ (m)")
481     ax.legend()
482     plt.savefig(results_path + "y_t." + IMAGE_EXTENSION)

483     fig, ax = plt.subplots()
484     ax.set_title("Y vs X")
485     if DRAW_ESTIMATED:
486         ax.plot(p_est[0, :], p_est[1, :], label="$P$ EKF", color="tab:orange")
487         if OUTPUT_CORRECTION:
488             ax.plot(p_output[0, :], p_output[1, :], label="$P$ Salida", color="tab:green")
489     if SHOW_GPS:
490         ax.scatter(gps[0, :], gps[1, :], label="$P$ GPS", marker="x", color="tab:green")
491         ax.plot(p[0, :], p[1, :], label="$P$ Groundtruth", color="tab:blue", linestyle="--")
492     plt.xlabel("$P_x$ (m)")
493     plt.ylabel("$P_y$ (m)")
494     ax.legend()
495     ax.set_aspect("equal")
496     plt.savefig(results_path + "tray." + IMAGE_EXTENSION)

497     fig, ax = plt.subplots()
498     ax.set_title("Velocidad X")
499     if DRAW_ESTIMATED:
500         ax.plot(t, v_est[0, :], label="$V_x$ EKF", color="tab:orange")
501         if OUTPUT_CORRECTION:
502             ax.plot(t, v_output[0, :], label="$V_x$ Salida", color="tab:green")
503             if SHOW_OUTPUT_CORRECTION:
504                 ax.plot(t, v_correction[0, :], label="$V_x$ correction")
505             if SHOW_OUTPUT_DELAYED:
506                 ax.plot(t, v_output_del[0, :], label="$V_x$ output delayed", color="tab:purple",
507                         linestyle="--")
508         ax.plot(t, v[0, :], label="$V_x$ Groundtruth", color="tab:blue", linestyle="--")
509     plt.xlabel("t (s)")
510     plt.ylabel("$V$ (m/s)")
511     ax.legend()
512     plt.savefig(results_path + "Vx." + IMAGE_EXTENSION)

513     fig, ax = plt.subplots()
514     ax.set_title("Velocidad Y")
515     if DRAW_ESTIMATED:
516         ax.plot(t, v_est[1, :], label="$V_y$ EKF", color="tab:orange")
517         if OUTPUT_CORRECTION:
518             ax.plot(t, v_output[1, :], label="$V_y$ Salida", color="tab:green")
519             if SHOW_OUTPUT_CORRECTION:
520                 ax.plot(t, v_correction[1, :], label="$V_y$ correction")
521             if SHOW_OUTPUT_DELAYED:
522                 ax.plot(t, v_output_del[1, :], label="$V_y$ output delayed", color="tab:purple",
523                         linestyle="--")
524         ax.plot(t, v[1, :], label="$V_y$ Groundtruth", color="tab:blue", linestyle="--")
525     plt.xlabel("t (s)")
526     plt.ylabel("$V$ (m/s)")
527     ax.legend()
528     plt.savefig(results_path + "Vy." + IMAGE_EXTENSION)

529     fig, ax = plt.subplots()
530     ax.set_title("Inclinación")
531     if DRAW_ESTIMATED:
532         ax.plot(t, theta_est, label=r"$\theta$ EKF", color="tab:orange")
533         if OUTPUT_CORRECTION:
534             ax.plot(t, theta_output, label=r"$\theta$ Salida", color="tab:green")
535             if SHOW_OUTPUT_CORRECTION:

```

```

537         ax.plot(t, theta_correction, label=r"$\theta$ correction", color="tab:purple",
538                 ↪ linestyle="--")
539     if SHOW_OUTPUT_DELAYED:
540         ax.plot(t, theta_output_del, label=r"$\theta$ output delayed", color="tab:purple",
541                 ↪ linestyle="--")
542     ax.plot(t, theta, label=r"$\theta$ Groundtruth", color="tab:blue", linestyle="--")
543     plt.xlabel("t (s)")
544     plt.ylabel(r"$\theta$ (rad)")
545     ax.legend()
546     plt.savefig(results_path + "theta." + IMAGE_EXTENSION)

547
548 # Sensors
549 fig, ax = plt.subplots()
550 ax.set_title("Acelerómetro")
551 ax.plot(
552     t, accel_gt[0, :], color="tab:orange", label="$a_x$ Groundtruth", linestyle="--"
553 )
554 ax.plot(
555     t, accel_gt[1, :], color="tab:blue", label="$a_y$ Groundtruth", linestyle="--"
556 )
557 ax.plot(t, accel[0, :], color="tab:orange", label="$a_x$ measure")
558 ax.plot(t, accel[1, :], color="tab:blue", label="$a_y$ measure")
559 plt.xlabel("t (s)")
560 plt.ylabel("a (m/s)")
561 ax.legend()
562 plt.savefig(results_path + "accel." + IMAGE_EXTENSION)

563 fig, ax = plt.subplots()
564 ax.set_title(r"Giróscopo ($\omega$)")
565 ax.plot(t, thetad, label="Groundtruth")
566 ax.plot(t, gyro, label="measure")
567 plt.xlabel("t (s)")
568 plt.ylabel(r"$\omega$ (rad/s)")
569 ax.legend()
570 plt.savefig(results_path + "gyro." + IMAGE_EXTENSION)

571
572 # Errors
573 p_groundtruth_del = np.empty((2, DATA_L))*np.nan
574 v_groundtruth_del = np.empty((2, DATA_L))*np.nan
575 theta_groundtruth_del = np.empty(DATA_L)*np.nan
576 p_groundtruth_del[0, :] = shift(p[0, :], buffer_size, cval=np.NaN)
577 p_groundtruth_del[1, :] = shift(p[1, :], buffer_size, cval=np.NaN)
578 v_groundtruth_del = shift(v, buffer_size, cval=np.NaN)
579 theta_groundtruth_del = shift(v, buffer_size, cval=np.NaN)

580
581 fig, ax = plt.subplots()
582 ax.set_title("Error de velocidad")
583 ax.plot(t, np.sqrt(P_est[2, 2, :]), color="tab:orange", label=r"$\sigma_{V_x}$", linestyle="--")
584 ax.plot(t, np.sqrt(P_est[3, 3, :]), color="tab:blue", label=r"$\sigma_{V_y}$", linestyle="--")
585 if SHOW_OUTPUT_ERROR and OUTPUT_CORRECTION:
586     ax.plot(t, abs(v[0, :]-v_output[0, :]), color="tab:orange", label="Error $V_x$ Salida")
587     ax.plot(t, abs(v[1, :]-v_output[1, :]), color="tab:blue", label="Error $V_y$ Salida")
588 else:
589     ax.plot(t, abs(v[0, :]-v_est[0, :]), color="tab:orange", label="Error $V_x$ EKF")
590     ax.plot(t, abs(v[1, :]-v_est[1, :]), color="tab:blue", label="Error $V_y$ EKF")
591 plt.xlabel("t (s)")
592 plt.ylabel("error (m/s)")
593 ax.legend()
594 plt.savefig(results_path + "V_error." + IMAGE_EXTENSION)

595
596 fig, ax = plt.subplots()
597 ax.set_title("Error de posición en X")
598 ax.plot(t, np.sqrt(P_est[0, 0, :]), color="tab:orange", label=r"$\sigma(P_x)$")
599 if OUTPUT_CORRECTION:
600     if SHOW_OUTPUT_ERROR:
601         ax.plot(t, abs(p[0, :]-p_output[0, :]), color="tab:green", label="Error $P_x$ Salida")
602         ax.plot(t, abs(p_groundtruth_del[0, :]-p_est[0, :]), color="tab:blue", label="Error $P_x$ EKF")
603     else:
604

```

```

605         ax.plot(t, abs(p[0, :]-p_est[0, :]), color="tab:blue", label="Error $P_x$ EKF")
606         plt.xlabel("t (s)")
607         plt.ylabel("error (m)")
608         ax.legend()
609         plt.savefig(results_path + "Px_error." + IMAGE_EXTENSION)
610
611
612     fig, ax = plt.subplots()
613     ax.set_title("Error de posición en Y")
614     ax.plot(t, np.sqrt(P_est[1, 1, :]), color="tab:orange", label=r"$\sigma(P_y)$")
615     if OUTPUT_CORRECTION:
616         if SHOW_OUTPUT_ERROR:
617             ax.plot(t, abs(p[1, :]-p_output[1, :]), color="tab:green", label="Error $P_y$ Salida")
618             ax.plot(t, abs(p_groundtruth_del[1, :]-p_est[1, :]), color="tab:blue", label="Error $P_y$"
619                     → EKF")
620         else:
621             ax.plot(t, abs(p[1, :]-p_est[1, :]), color="tab:blue", label="Error $P_y$ EKF")
622         plt.xlabel("t (s)")
623         plt.ylabel("error (m)")
624         ax.legend()
625         plt.savefig(results_path + "Py_error." + IMAGE_EXTENSION)
626
627     fig, ax = plt.subplots()
628     ax.set_title("Error de inclinación")
629     ax.plot(t, np.sqrt(P_est[4, 4, :]), label=r"$\sigma(\theta)$")
630     if SHOW_OUTPUT_ERROR and OUTPUT_CORRECTION:
631         ax.plot(t, abs(theta-theta_output), label="Error $\theta$ Salida")
632     else:
633         ax.plot(t, abs(theta-theta_est), label="Error $\theta$ EKF")
634     plt.xlabel("t (s)")
635     plt.ylabel("error (rad)")
636     ax.legend()
637     plt.savefig(results_path + "theta_error." + IMAGE_EXTENSION)
638
639     fig, ax = plt.subplots()
640     ax.set_title("Matriz de covarianzas") # Es diagonal
641     ax.plot(t, P_est[0, 1, :], label="$P_{\{est\}}[0,1]$")
642     ax.plot(t, P_est[0, 2, :], label="$P_{\{est\}}[0,2]$")
643     ax.plot(t, P_est[0, 3, :], label="$P_{\{est\}}[0,3]$")
644     ax.plot(t, P_est[0, 4, :], label="$P_{\{est\}}[0,4]$")
645     ax.plot(t, P_est[1, 2, :], label="$P_{\{est\}}[1,2]$")
646     ax.plot(t, P_est[1, 3, :], label="$P_{\{est\}}[1,3]$")
647     ax.plot(t, P_est[1, 4, :], label="$P_{\{est\}}[1,4]$")
648     ax.plot(t, P_est[2, 3, :], label="$P_{\{est\}}[2,3]$")
649     ax.plot(t, P_est[2, 4, :], label="$P_{\{est\}}[2,4]$")
650     ax.plot(t, P_est[3, 4, :], label="$P_{\{est\}}[3,4]$")
651     ax.plot(t, P_est[0, 0, :], label="$P_x$", linestyle="--")
652     ax.plot(t, P_est[1, 1, :], label="$P_y$", linestyle="--")
653     ax.plot(t, P_est[2, 2, :], label="$V_x$", linestyle="--")
654     ax.plot(t, P_est[3, 3, :], label="$V_y$", linestyle="--")
655     plt.xlabel("$t$ (s)")
656     ax.legend()
657     plt.savefig(results_path + "P_est." + IMAGE_EXTENSION)
658
659     if SHOW_PLOTS:
660         plt.show()
661
662     if SHOW_ANIMATED:
663         draw_animation(p[0, :], p[1, :], theta)
664
665 if __name__ == "__main__":
666     main()

```

# Apéndice B

## Detector de marcadores visuales

Los siguientes archivos se encuentran también en [https://github.com/isidroas/rpi\\_vision\\_uav](https://github.com/isidroas/rpi_vision_uav)

### B.1 main.cpp

```
1  /* mavsdk header */
2  #include <chrono>
3  #include <cmath>
4  #include <future>
5  #include <iostream>
6  #include <fstream>
7  #include <thread>
8  #include <unistd.h>
9  #include <Eigen/Dense>
10 #include "marker_vision.h"
11 #include "mavlink_helper.h"
12 #include "shared_memory_helper.h"
13
14 using std::chrono::milliseconds;
15 using std::chrono::seconds;
16 using std::this_thread::sleep_for;
17
18 using namespace std;
19
20 #define TELEMETRY_CONSOLE_TEXT "\033[34m" // Turn text on console blue
21
22
23 #define DEBUG
24
25
26 #define UPDATE_DEBUG_RATE 30 // Cada cuantas iteraciones se calculan e imprimen las estadísticas
27
28 static bool readParameters(string filename, bool &mav_connect, bool &log_file, int &loop_period_ms,
29     → bool &wait_key, int &wait_key_mill) {
30     FileStorage fs(filename, FileStorage::READ);
31     if(!fs.isOpened())
32         return false;
33     mav_connect = (string)fs["mav_connect"]=="true";
34     log_file = (string)fs["log_file"]=="true";
35     loop_period_ms = (int)fs["loop_period_ms"];
36     wait_key_mill = (int)fs["wait_key_mill"];
37     wait_key = (string)fs["wait_key"]=="true";
38     cout << "Parámetros generales:" << endl;
39     cout << "\tConexión mavlink:\t\t\t" << mav_connect << endl;
40     cout << "\tLog de medidas:\t\t\t" << log_file << endl;
41     cout << "\tPeriodo mínimo de actualización:\t" << loop_period_ms << endl;
42     if (wait_key){
43         cout << "\tEspera a la presión de una tecla:\t" << wait_key_mill << endl;
44     }
45     cout << endl;
46     return true;
```

```

46 }
47
48
49
50 int main()
51 {
52     cout << "-----" << endl;
53     cout << "-----Vision position estimator-----" << endl;
54     cout << "-----" << endl;
55     cout << endl;
56
57     /* Startup python script for logging */
58     int res=system("python3 ..../python_scripts/startup.py");
59     if (res!=0){
60         cout << "El script de inicio ha fallado con código " << res << endl;
61         exit(1);
62     }
63
64     bool mav_connect, log_file, wait_key;
65     int loop_period_ms, wait_key_mill;
66     readParameters("../vision_params.yml", mav_connect, log_file, loop_period_ms, wait_key,
67     ↵ wait_key_mill);
68
69     CommunicationClass commObj;
70     if (mav_connect)
71         commObj.init();
72
73     double total_time = 0;
74     int totalIterations = 0;
75     auto wake_up_time = std::chrono::steady_clock::now() +
76     ↵ std::chrono::milliseconds(loop_period_ms);
77     double tick_global_ant = (double)getTickCount();
78     Eigen::Vector3d pos, euler_angles ;
79
80     double seconds_init = (double)getTickCount()/getTickFrequency();
81
82     //TODO: transladar estos parámetros al archivo.
83     bool vision_activated=true;
84
85     VisionClass visionMarker;
86     if (vision_activated)
87         visionMarker.init();
88
89     std::ofstream myfile;
90     if (log_file){
91         myfile.open("../results/latest/log.csv");
92         myfile << "px" << "," << "py" << "," << "pz" << "," << "roll" << ","
93         << "pitch" << "," << "yaw" << "," << "t" <<"\n";
94     }
95
96     // Inicializa la memoria compartida
97     shmem_init();
98
99     /*** Main Loop ***/
100    while(true){
101
102        if (vision_activated){
103            // Grab image and exists if there is no one
104            if (!visionMarker.grab_and_retrieve_image()) break;
105            // detect markers
106            bool found_marker = visionMarker.detect_marker(pos, euler_angles);
107
108            if (found_marker and mav_connect)
109                commObj.send_msg(pos, euler_angles);
110
111        }
112
113        //TODO: hacer esto con menor frecuencia
114        // Get position setpoint from Trajectory Generator

```

```

115     Eigen::Vector3d pos_setpoint;
116     get_pos_from_tray_gen(pos_setpoint);
117
118     if (mav_connect){
119         // Send setpoint to autopilot
120         commObj.send_pos_setpoint(pos_setpoint);
121
122         // Get position from autopilot. Not blocking function
123         Eigen::Vector3d pos_ned;
124         bool valid_ned = commObj.get_pos_ned(pos_ned);
125
126         // Send ned position to Trajectory Generator
127         if (valid_ned)
128             send_pos_ned_to_tray_gen(pos_ned);
129             //cout << "Posición NED: " << pos_ned;
130     }
131
132
133 #ifdef DEBUG
134 // Update counters
135 double tick_global_act = (double)getTickCount();
136 double execution_time = (tick_global_act - tick_global_ant) / getTickFrequency();
137 tick_global_ant = tick_global_act;
138 total_time += execution_time;
139 totalIterations++;
140
141 /* Print data every 30 frames = 1 seg approx*/
142 if(totalIterations % UPDATE_DEBUG_RATE == 0) {
143     cout << "Execution time = " << execution_time * 1000 << " ms "
144     << "(Mean = " << 1000 * total_time / float(UPDATE_DEBUG_RATE) << " ms)" << endl;
145     if (vision_activated)
146         visionMarker.print_statistics(pos, euler_angles);
147     total_time=0;
148     cout << endl;
149 }
150 #endif
151
152
153 if (log_file){
154     double seconds = getTickCount() / getTickFrequency() - seconds_init;
155     myfile << pos[0] << "," << pos[1] << "," << pos[2] << "," << euler_angles[0] << ","
156     << euler_angles[1] << "," << euler_angles[2] << "," << seconds << "\n";
157 }
158
159 if (loop_period_ms!=0){
160     std::this_thread::sleep_until(wake_up_time);
161     wake_up_time = std::chrono::steady_clock::now() +
162     → std::chrono::milliseconds(loop_period_ms);
163 }
164
165 if (wait_key){
166     int key = waitKey( wait_key_mill );
167     if(key == 27) break; // exits if esc is pressed in window
168 }
169
170 if (log_file)
171     myfile.close();
172
173 res=system("python3 ../python_scripts/shutdown.py");
174 if (res!=0){
175     cout << "El script de apagado ha fallado con código " << res << endl;
176     exit(1);
177 }
178 shmem_cleanup();
179 std::cout << "Finished..." << std::endl;
180 return EXIT_SUCCESS;
181 }
```

## B.2 vision\_params.yml

```

1  %YAML:1.0
2  #dict_type: 10    # 6x6 256
3  dict_type: 0     # 4x4 50
4
5  ## Single marker
6  #marker_length: 0.179
7  marker_length: 0.2335
8
9  ## Diamond
10 diamond: false
11 # autoscale not implemented yet
12 autoScale: false
13
14 ## Charuco
15 charuco: true
16 refinStrategy: true
17 # Charuco Boards
18 squaresX: 5
19 squaresY: 7
20 squareLength: 0.0475
21 markerLength: 0.0285
22 #squareLength: 0.0471
23 #markerLength: 0.0282
24
25
26 ## Camera
27 exposure_time: 20
28 # if fps>40, fov decreases
29 fps: 40
30 video_file: "../videos/vuelo_foco.h264"
31 frame_width: 640
32 frame_height: 480
33 #frame_width: 1280
34 #frame_height: 720
35 camera_parameters: "../calibration/rpi_v2_camera/cal.yml"
36 #camera_parameters: "../calibration/hp_camera/cal.yml"
37
38
39 ## General parameters
40 mav_connect: true
41 loop_period_ms: 0
42 open_window: true
43 wait_key_mill: 1
44 #TODO: this is mandatory when open_window=true
45 wait_key: true
46 show_rejected: false
47
48 ## Logging
49 write_images: false
50 log_file: false
51 generate_video_from_images: false

```

## B.3 marker\_vision.h

```

1  #include <opencv2/highgui.hpp>
2  #include <opencv2/aruco.hpp>
3  #include <opencv2/aruco/charuco.hpp>
4  #include <opencv2/calib3d.hpp>
5  #include <opencv2/core/eigen.hpp>
6  #include <unistd.h> // for sleep
7  using namespace cv;
8  using namespace std;

```

```

9
10 //##define WAIT_KEY_MILL      1 // tiempo de espera entre fotogramas cuando se abre la ventana, si
11 // vale 0, solo avanza cuando se presiona alguna tecla
12 #define AUTO_SCALE_FACTOR 1
13
14 //##define ROT_POS_ORI
15
16 class VisionClass {
17     public:
18     void init();
19     int grab_and_retrieve_image(){
20         double tick0 = (double)getTickCount();
21         int res = inputVideo.grab();
22         inputVideo.retrieve(image);
23         double tick1 = (double)getTickCount();
24         execution_time_video_grab_and_ret = (tick1-tick0) / getTickFrequency();
25         return res;
26     }
27     bool detect_marker(Eigen::Vector3d &pos, Eigen::Vector3d &eul);
28     void print_statistics(Eigen::Vector3d &pos, Eigen::Vector3d &eul);
29
30     private:
31         bool readDetectorParameters(string filename, Ptr<aruco::DetectorParameters> &params);
32         bool readVisionParameters(string filename);
33         void InvertPose(Eigen::Vector3d &pos, Eigen::Vector3d &eul, Vec3d &rvec, Vec3d &tvec);
34         bool readCameraParameters(string filename, Mat &camMatrix, Mat &distCoeffs);
35         Eigen::Vector3d rotationMatrixToEulerAngles(Eigen::Matrix3d &R);
36
37         Mat image;
38         Mat imageCopy;
39         VideoCapture inputVideo;
40         Ptr<aruco::Dictionary> dictionary;
41         Ptr<aruco::DetectorParameters> detectorParams;
42         Mat camMatrix;
43         Mat distCoeffs;
44         string calibration_file;
45         string video_file;
46         bool show_rejected;
47         float marker_length;
48         int dict_type;
49         float axisLength;
50         bool open_window;
51         bool write_images;
52         // camera config
53         int exposure_time;
54         int fps;
55         int frame_width;
56         int frame_height;
57         // diamond specific
58         bool diamond;
59         bool autoScale;
60         // charuco specific
61         bool charuco;
62         bool refindStrategy;
63         int squaresX;
64         int squaresY;
65         float square_length;
66         float marker_length_ch;
67         Ptr<aruco::CharucoBoard> charucoboard;
68         Ptr<aruco::Board> board;
69         // logging
70         int totalIterations=0;
71         double execution_time_video_grab_and_ret;
72         double execution_time_detect;
73         double n_position_get=0;
74         double n_since_call_statistics=0;
75         double valid_pose;
76     };
77
78

```

```

79 // Calculates rotation matrix to euler angles
80 Eigen::Vector3d VisionClass::rotationMatrixToEulerAngles(Eigen::Matrix3d &R)
81 {
82
83     float sy = sqrt(R(0,0) * R(0,0) + R(1,0) * R(1,0) );
84
85     bool singular = sy < 1e-6; // If
86
87     float x, y, z;
88     if (!singular)
89     {
90         x = atan2(R(2,1) , R(2,2));
91         y = atan2(-R(2,0), sy);
92         z = atan2(R(1,0), R(0,0));
93     }
94     else
95     {
96         x = atan2(-R(1,2), R(1,1));
97         y = atan2(-R(2,0), sy);
98         z = 0;
99     }
100    return Eigen::Vector3d(x, y, z);
101 }
102
103
104 bool VisionClass::readCameraParameters(string filename, Mat &camMatrix, Mat &distCoeffs) {
105     FileStorage fs(filename, FileStorage::READ);
106     if(!fs.isOpened())
107         return false;
108     fs["camera_matrix"] >> camMatrix;
109     fs["distortion_coefficients"] >> distCoeffs;
110     return true;
111 }
112
113
114 bool VisionClass::detect_marker(Eigen::Vector3d &pos, Eigen::Vector3d &eul){
115     double tick0 = (double)getTickCount();
116
117     vector< int > ids, charucoIds;
118     vector< vector< Point2f > > corners, rejected;
119     vector< Vec3d > rvecs, tvecs;
120     Vec3d rvec, tvec;
121     vector< Point2f > charucoCorners;
122     vector< vector< Point2f > > diamondCorners;
123     vector< Vec4i > diamondIds;
124
125
126     aruco::detectMarkers(image, dictionary, corners, ids, detectorParams, rejected);
127
128     bool found_marker=ids.size() > 0;
129     valid_pose = false;
130     int interpolatedCorners = 0;
131
132     if (charuco){
133         if(refindStrategy)
134             aruco::refineDetectedMarkers(image, board, corners, ids, rejected,
135                                         camMatrix, distCoeffs);
136
137         // interpolate charuco corners
138         if(found_marker)
139             interpolatedCorners =
140                 aruco::interpolateCornersCharuco(corners, ids, image, charucoboard,
141                                                 charucoCorners, charucoIds, camMatrix,
142                                                 → distCoeffs);
143         if ((int)ids.size()==17){
144             // estimate charuco board pose
145             valid_pose = aruco::estimatePoseCharucoBoard(charucoCorners, charucoIds, charucoboard,
146                                               camMatrix, distCoeffs, rvec, tvec);
147         }
148         else{
149             valid_pose= false;

```

```

149         }
150     }
151     else if (diamond){
152         if (found_marker){
153             aruco::detectCharucoDiamond(image, corners, ids,
154                                         square_length / marker_length_ch, diamondCorners,
155                                         ↪ diamondIds,
156                                         camMatrix, distCoeffs);
157
158             if(!autoScale) {
159                 aruco::estimatePoseSingleMarkers(diamondCorners, square_length, camMatrix,
160                                                 distCoeffs, rvecs, tvecs);
161             } else {
162                 // if autoscale, extract square size from last diamond id
163                 //for(unsigned int i = 0; i < diamondCorners.size(); i++) {
164                 //    float autoSquareLength = AUTO_SCALE_FACTOR * float(diamondIds[i].val[3]);
165                 //    vector< vector< Point2f > > currentCorners;
166                 //    vector< Vec3d > currentRvec, currentTvec;
167                 //    currentCorners.push_back(diamondCorners[i]);
168                 //    aruco::estimatePoseSingleMarkers(currentCorners, autoSquareLength,
169                 ↪ camMatrix,
170                 //    distCoeffs, currentRvec, currentTvec);
171                 //    rvecs.push_back(currentRvec[0]);
172                 //    tvecs.push_back(currentTvec[0]);
173                 //}
174                 cout << "Autoscale todavía no implementado" << endl;
175                 exit(0);
176             }
177             if (tvecs.size()>0){
178                 rvec=rvecs[0];
179                 tvec=tvecs[0];
180                 valid_pose= true;
181             }
182         }
183         else{
184             if(found_marker){
185                 aruco::estimatePoseSingleMarkers(corners, marker_length, camMatrix, distCoeffs, rvecs,
186                                             ↪ tvecs);
187                 valid_pose = true;
188                 rvec=rvecs[0];
189                 tvec=tvecs[0];
190             }
191
192             if (open_window){
193
194                 image.copyTo(imageCopy);
195
196                 if(found_marker){
197                     aruco::drawDetectedMarkers(imageCopy, corners, ids);
198                 }
199                 if(interpolatedCorners > 0) {
200                     Scalar color;
201                     color = Scalar(255, 0, 0);
202                     aruco::drawDetectedCornersCharuco(imageCopy, charucoCorners, charucoIds, color);
203                 }
204                 if (valid_pose){
205                     aruco::drawAxis(imageCopy, camMatrix, distCoeffs, rvec, tvec, axisLength);
206                 }
207
208                 if(show_rejected && rejected.size() > 0)
209                     aruco::drawDetectedMarkers(imageCopy, rejected, noArray(), Scalar(100, 0, 255));
210
211                 imshow("out", imageCopy);
212                 if (write_images){
213                     char path [100];
214                     sprintf(path,"../results/latest/images/image%d.png", totalIterations);
215                     imwrite(path,imageCopy);
216                 }

```

```

217     //      waitKey( WAIT_KEY_MILL );
218 }
219
220 if (valid_pose){
221     InvertPose(pos, eul, rvec, tvec);
222 }
223 else{
224     pos[0]=pos[1]=pos[2]=NAN;
225     eul[0]=eul[1]=eul[2]=NAN;
226 }
227
228 totalIterations++;
229
230 double tick1 = (double)getTickCount();
231
232 execution_time_detect = (tick1-tick0) / getTickFrequency();
233 n_since_call_statistics++;
234 if (valid_pose)
235     n_position_get++;
236
237 return valid_pose;
238 }
239
240 /* @brief Invierte la posición y la rotación. También corrige la posición de la cámara con respecto
241    → al UAV
242 * @param rvec Vector de entrada. Vector de rotación del marcador con respecto a los ejes de la
243    → cámara
244 * @param tvec Vector de entrada. Translación del marcador con respecto a los ejes de la cámara
245 * @param pos Vector de salida. Posición del uav/cámara con respecto al marcador
246 * @param eul Vector de salida. Orientación del uav con respecto al marcador. El orden de los
247    → elementos son 0: roll, 1: pitch, 2: yaw
248 */
249 void VisionClass::InvertPose(Eigen::Vector3d &pos, Eigen::Vector3d &eul, Vec3d &rvec, Vec3d
250    → &tvec){//
251
252     Eigen::Vector3d pos_marker_in_camera(tvec[0],tvec[1],tvec[2]);
253
254     // Transformación de vector de rotación a matriz de rotación
255     cv::Mat rot_mat;
256     Eigen::Matrix3d rot_mat_marker_from_camera;
257     Rodrigues(rvec,rot_mat);
258     cv::cv2eigen(rot_mat, rot_mat_marker_from_camera);
259
260     // La inversa de una matriz de rotación es igual a su traspuesta
261     Eigen::Matrix3d rot_mat_camera_from_marker = rot_mat_marker_from_camera.transpose() ;
262
263     // Se obtiene la posición del marcador en unos ejes paralelos al marcador centrados en la
264     → cámara
265     Eigen::Vector3d pos_marker_in_marker_axis =
266     → rot_mat_camera_from_marker*pos_marker_in_camera;
267
268     // Si queremos que la posición esté centrada en el marcador y no en la cámara, es necesario
269     → negarla
270     pos = -pos_marker_in_marker_axis;
271
272     // Aquí debemos de tener en cuenta la rotación de la cámara con respecto al uav. Esta es de
273     → 180º alrededor del eje z.
274     // Queremos rotar en ejes absolutos y no en los ejes de rot_mat_marker_from_camera, por lo
275     → tanto premultiplicamos.
276     Eigen::Matrix3d rot_mat_camera_from_uav;
277     rot_mat_camera_from_uav << -1, 0, 0,
278                           0, -1, 0,
279                           0, 0, 1;
280     Eigen::Matrix3d rot_mat_marker_from_uav = rot_mat_camera_from_uav *
281     → rot_mat_marker_from_camera;
282
283     // Se obtiene la orientación del uav visto desde el marcador
284     Eigen::Matrix3d rot_mat_uav_from_marker = rot_mat_marker_from_uav.transpose() ;
285
286     // Se obtiene los ángulos de Tait-Bryan en el orden Z-Y-X (ángulos de euler)
287     eul = rotationMatrixToEulerAngles(rot_mat_uav_from_marker);
288 }
```

```

278
279 #ifdef ROT_POS_ORI
280 // Transformaciones después de invertir la posición y la orientación. Queremos que la posición
281 // esté expresado en un sistema de referencia con su eje z apuntando hacia abajo. El del
282 // marcador apunta hacia arriba, así
283 // que se rotará 180° en el eje x
284 Eigen::Matrix3d          rot_mat_marker_from_NED;
rot_mat_marker_from_NED << 1, 0, 0,

```

La siguiente función invierte la posición y la rotación. También corrige la posición de la cámara con respecto al UAV. Sus argumentos son:

- rvec. Vector de entrada. Vector de rotación del marcador con respecto a los ejes de la cámara
- tvec. Vector de entrada. Translación del marcador con respecto a los ejes de la cámara
- pos. Vector de salida. Posición del uav/cámara con respecto al marcador
- eul. Vector de salida. Orientación del uav con respecto al marcador. El orden de los elementos son 0: roll, 1: pitch, 2: yaw

```

291 #endif
292 }
293
294
295 bool VisionClass::readVisionParameters(string filename) {
296     FileStorage fs(filename, FileStorage::READ);
297     if(!fs.isOpened())
298         return false;
299
300     fs["camera_parameters"] >> calibration_file;
301     fs["video_file"] >> video_file;
302     open_window = (string)fs["open_window"]=="true";
303     show_rejected = (string)fs["show_rejected"]=="true";
304     charuco = (string)fs["charuco"]=="true";
305     refindStrategy = (string)fs["refindStrategy"]=="true";
306     marker_length = (float)fs["marker_length"];
307     dict_type = (int)fs["dict_type"];
308     exposure_time = (int)fs["exposure_time"];
309     fps = (int)fs["fps"];
310     squaresX = (int)fs["squaresX"];
311     squaresY = (int)fs["squaresY"];
312     marker_length_ch = (float)fs["markerLength"];
313     square_length = (float)fs["squareLength"];
314     squaresY = (int)fs["squaresY"];
315     frame_height = (int)fs["frame_height"];
316     frame_width = (int)fs["frame_width"];
317     diamond = (string)fs["diamond"]=="true";
318     autoScale = (string)fs["autoScale"]=="true";
319     write_images = (string)fs["write_images"]=="true";
320
321     // Print them
322     cout << "Parámetros de la visión:" << endl;
323     cout << "\tEl fichero de calibracion es:\t" << calibration_file << endl;
324     cout << "\tArchivo de video:\t\t" << video_file << endl;
325     cout << "\tActivación de la ventana:\t" << open_window << endl;
326     cout << "\tMostrar rechazados:\t\t" << show_rejected << endl;
327     cout << "\tTamaño del marcador:\t\t" << marker_length << endl;
328     cout << "\tTipo de diccionario:\t\t" << dict_type << endl;
329     cout << "\tTiempo de exposición:\t\t" << exposure_time << endl;
330     cout << "\tFPS:\t\t\t\t" << fps << endl;
331     if (charuco){
332         cout << endl;
333         cout << "Charuco:" << endl;
334         cout << "\trefindStrategy:\t\t" << refindStrategy << endl;
335         cout << "\tmarkerLength:\t\t" << marker_length_ch << endl;
336         cout << "\tsquareLength:\t\t" << square_length << endl;

```

```

337     cout << "\tsquaresX:\t\t" << squaresX << endl;
338     cout << "\tsquaresY:\t\t" << squaresY << endl;
339 }
340 else if(diamond){
341     cout << endl;
342     cout << "Diamond:" << endl;
343     cout << "\tautoScale:\t\t" << autoScale << endl;
344 }
345
346 cout << endl;
347 return true;
348 }

349
350 bool VisionClass::readDetectorParameters(string filename, Ptr<aruco::DetectorParameters> &params) {
351     FileStorage fs(filename, FileStorage::READ);
352     if(!fs.isOpened())
353         return false;
354     fs["adaptiveThreshWinSizeMin"] >> params->adaptiveThreshWinSizeMin;
355     fs["adaptiveThreshWinSizeMax"] >> params->adaptiveThreshWinSizeMax;
356     fs["adaptiveThreshWinSizeStep"] >> params->adaptiveThreshWinSizeStep;
357     fs["adaptiveThreshConstant"] >> params->adaptiveThreshConstant;
358     fs["minMarkerPerimeterRate"] >> params->minMarkerPerimeterRate;
359     fs["maxMarkerPerimeterRate"] >> params->maxMarkerPerimeterRate;
360     fs["polygonalApproxAccuracyRate"] >> params->polygonalApproxAccuracyRate;
361     fs["minCornerDistanceRate"] >> params->minCornerDistanceRate;
362     fs["minDistanceToBorder"] >> params->minDistanceToBorder;
363     fs["minMarkerDistanceRate"] >> params->minMarkerDistanceRate;
364     fs["cornerRefinementMethod"] >> params->cornerRefinementMethod;
365     fs["cornerRefinementWinSize"] >> params->cornerRefinementWinSize;
366     fs["cornerRefinementMaxIterations"] >> params->cornerRefinementMaxIterations;
367     fs["cornerRefinementMinAccuracy"] >> params->cornerRefinementMinAccuracy;
368     fs["markerBorderBits"] >> params->markerBorderBits;
369     fs["perspectiveRemovePixelPerCell"] >> params->perspectiveRemovePixelPerCell;
370     fs["perspectiveRemoveIgnoredMarginPerCell"] >> params->perspectiveRemoveIgnoredMarginPerCell;
371     fs["maxErroneousBitsInBorderRate"] >> params->maxErroneousBitsInBorderRate;
372     fs["minOtsuStdDev"] >> params->minOtsuStdDev;
373     fs["errorCorrectionRate"] >> params->errorCorrectionRate;
374     return true;
375 }
376
377 void VisionClass::init(){
378     bool readOk = readVisionParameters("../vision_params.yml");
379     if(!readOk) {
380         cerr << "Invalid general vision parameters file" << endl;
381         exit(0);
382     }
383
384     /*** Vision setup ***/
385     detectorParams = aruco::DetectorParameters::create();
386     readOk = readDetectorParameters( "../detector_params.yml", detectorParams );
387     if(!readOk) {
388         cerr << "Invalid detector parameters file" << endl;
389         exit(0);
390     }
391
392     dictionary =
393         aruco::getPredefinedDictionary(aruco::PREDEFINED_DICTIONARY_NAME(dict_type));
394
395     readOk = readCameraParameters(calibration_file, camMatrix, distCoeffs);
396     if(!readOk) {
397         cerr << "Invalid camera file" << endl;
398         exit(0);
399     }
400
401     if (charuco){
402         charucoboard = aruco::CharucoBoard::create(squaresX, squaresY, square_length,
403             marker_length_ch, dictionary);
404         board = charucoboard.staticCast<aruco::Board>();
405         axisLength = 0.5f * ((float)min(squaresX, squaresY) * (square_length));
406     }
407     else{

```

```

406     axisLength = 0.5f * marker_length;
407 }
408
409
410 if (video_file!=""){
411     inputVideo.open(video_file);
412 }
413 else{
414     inputVideo.open(0);
415     inputVideo.set(CAP_PROP_FRAME_WIDTH, frame_width);
416     inputVideo.set(CAP_PROP_FRAME_HEIGHT, frame_height);
417     string cmd;
418     if (fps!=0){
419         cmd=(string)"v4l2-ctl -d /dev/video0 -p "+ to_string(fps);
420         const char* aux1=cmd.data();
421         system(aux1);
422     }
423     if (exposure_time!=0){
424         cmd=(string)"v4l2-ctl -d /dev/video0 -c auto_exposure=1 -c
425             exposure_time_absolute="+ to_string(exposure_time);
426         const char* aux2=cmd.data();
427         system(aux2);
428     }
429     else{
430         system("v4l2-ctl -d /dev/video0 -c auto_exposure=0");
431     }
432     system(" v4l2-ctl -V");
433 }
434 /* Test an image */
435 grab_and_retrieve_image();
436 cout << "Ancho de la imagen:\t" << image.cols << endl;
437 cout << "Alto de la imagen:\t" << image.rows << endl << endl;
438
439 void VisionClass::print_statistics(Eigen::Vector3d &pos, Eigen::Vector3d &eul){
440     cout << "Image grabbing and retrieving= " << execution_time_video_grab_and_ret * 1000
441         << " ms " << endl;
442     cout << "Marker detection= " << execution_time_detect * 1000 << " ms " << endl;
443     cout << "Frames with position = " << n_position_get/n_since_call_statistics * 100 << "
444         << "% " << endl;
445
446     if(valid_pose){
447         cout << "Estimated position:\t" << pos[0] << "\t" << pos[1] << "\t"
448             << pos[2] << endl;
449         cout << "Estimated orientation:\t" << eul[0] << "\t" << eul[1] << "\t" <<
450             eul[2] << endl;
451     }
452     n_position_get=0;
453     n_since_call_statistics=0;
454     cout << endl;
455 }

```

## B.4 mavlink\_helper.h

```

1 #include <mavSDK/mavSDK.h>
2 #include <mavSDK/plugins/offboard/offboard.h>
3 #include <mavSDK/plugins/telemetry/telemetry.h>
4 #include <mavSDK/plugins/offboard/offboard.h>
5 #include <mavSDK/plugins/mocap/mocap.h>
6 #include <Eigen/Dense>
7
8 using namespace mavSDK;
9 using std::chrono::milliseconds;
10 using std::this_thread::sleep_for;
11
12 //##define CONNECTION_URL "serial:///dev/ttyUSB0:921600"

```

```

13 #define CONNECTION_URL "udp://:14540"
14 //##define UUID 3690507541151037490 // autopilot cube
15 //##define UUID 3762846584429098293 // autopilot cuav
16 #define UUID 5283920058631409231 // simulation
17 #define ERROR_CONSOLE_TEXT "\033[31m" // Turn text on console red
18 #define NORMAL_CONSOLE_TEXT "\033[0m" // Restore normal console colour
19
20 static float pos_north=0;
21 static float pos_east=0;
22 static float pos_down=0;
23 static bool valid_ned=false;
24
25 class CommunicationClass{
26     public:
27         void send_msg(Eigen::Vector3d pos, Eigen::Vector3d eul);
28         void send_pos_setpoint(Eigen::Vector3d );
29         void init();
30         bool get_pos_ned(Eigen::Vector3d &pos);
31     private:
32         void wait_until_discover(Mavsdk& dc);
33         shared_ptr<Mocap> mocap;
34         shared_ptr<Telemetry> telemetry;
35         shared_ptr<Offboard> offboard;
36         Mocap::VisionPositionEstimate est_pos;
37         Mavsdk dc;
38         ConnectionResult connection_result;
39     };
40
41
42 void CommunicationClass::wait_until_discover(Mavsdk& dc)
43 {
44     std::cout << "Waiting to discover system..." << std::endl;
45     std::promise<void> discover_promise;
46     auto discover_future = discover_promise.get_future();
47
48     dc.register_on_discover([&discover_promise](uint64_t uuid) {
49         std::cout << "Discovered system with UUID: " << uuid << std::endl;
50         discover_promise.set_value();
51     });
52
53     discover_future.wait();
54 }
55
56 void CommunicationClass::init()
57 {
58     connection_result = dc.add_any_connection(CONNECTION_URL);
59
60     if (connection_result != ConnectionResult::Success) {
61         std::cout << ERROR_CONSOLE_TEXT << "Connection failed: " << connection_result
62             << NORMAL_CONSOLE_TEXT << std::endl;
63         exit(0);
64     }
65
66     bool connected = dc.is_connected(UUID);
67     while(connected==false){
68         connected = dc.is_connected(UUID);
69         cout << "Waiting system for connection ..." << endl;
70         sleep_for(milliseconds(500));
71     }
72     System& system = dc.system(UUID);
73
74     mocap = std::make_shared<Mocap>(system);
75     telemetry = std::make_shared<Telemetry>(system);
76     offboard = std::make_shared<Offboard>(system);
77
78     // Set update rate
79     // const Telemetry::Result set_rate_result = telemetry->set_rate_position(1.0);
80     // if (set_rate_result != Telemetry::Result::Success) {
81     //     // handle rate-setting failure (in this case print error)
82     //     std::cout << "Setting rate failed:" << set_rate_result << std::endl;
83     // }

```

```

84     telemetry->subscribe_position_velocity_ned([](Telemetry::PositionVelocityNed posvel) {
85         pos_north=posvel.position.north_m;
86         pos_east=posvel.position.east_m;
87         pos_down=posvel.position.down_m;
88         valid_ned=true;
89     });
90 }
91
92 void CommunicationClass::send_msg(Eigen::Vector3d pos, Eigen::Vector3d eul)
93 {
94     est_pos.position_body.x_m = pos[0];
95     est_pos.position_body.y_m = pos[1];
96     est_pos.position_body.z_m = pos[2];
97     est_pos.angle_body.roll_rad = eul[0];
98     est_pos.angle_body.pitch_rad = eul[1];
99     est_pos.angle_body.yaw_rad = eul[2];
100    std::vector<float> covariance{NAN};
101    est_pos.pose_covariance.covariance_matrix=covariance;
102    Mocap::Result result= mocap->set_vision_position_estimate(est_pos);
103    if(result!=Mocap::Result::Success){
104        std::cerr << ERROR_CONSOLE_TEXT << "Set vision position failed: " << result <<
105        NORMAL_CONSOLE_TEXT << std::endl;
106    }
107
108 void CommunicationClass::send_pos_setpoint(Eigen::Vector3d pos)
109 {
110     Offboard::PositionNedYaw pos_setpoint{};
111     pos_setpoint.north_m=pos[0];
112     pos_setpoint.east_m=pos[1];
113     pos_setpoint.down_m=pos[2];
114     pos_setpoint.yaw_deg=0; // TODO: decide yaw. Se podrá Nan?
115     Offboard::Result result = offboard->set_position_ned(pos_setpoint);
116     if(result!=Offboard::Result::Success){
117         std::cerr << ERROR_CONSOLE_TEXT << "Set offboard position setpoint failed: " << result <<
118         NORMAL_CONSOLE_TEXT << std::endl;
119     }
120     bool CommunicationClass::get_pos_ned(Eigen::Vector3d &pos){
121         pos[0]=pos_north;
122         pos[1]=pos_east;
123         pos[2]=pos_down;
124         return valid_ned;
125     }

```



# Índice de Figuras

---

1.1	Estación de carga de <i>Flytbase</i>	1
2.1	Manejo de medidas retrasadas	4
2.2	Quadrotor en dos dimensiones	9
2.3	EKF no ejecuta la fase de actualización	10
2.4	Fusión del GPS con retraso	10
2.5	Experimento con manejo de medidas retrasadas	11
3.1	Componentes del quadrotor	14
3.2	A la izquierda: diagrama de flujo del programa que se corre en el ordenador embebido, a la derecha: algunas de las tareas del autopiloto	16
3.3	Sistemas de referencia presentes en el problema	17
3.4	Pasos intermedios en el proceso de detección de marcadores Aruco. Fuente [7]	19
3.5	Superposición de los ejes de referencia del marcador y del cuadrilátero que lo rodea (trazado en verde)	22
3.6	Modo <i>altitude</i>	24
3.7	Modo <i>position</i> desde el instante 158s hasta el 168s	25
3.8	Quadrotor sobrevolando un tablero de marcadores	26



# **Lista de códigos**

---

2.1	Código 2.1: Corrección del buffer de salida. Ubicado en la línea 488 del archivo <i>Firmware/src/lib/ecl/EKF/ekf.cpp</i>	5
2.2	Código 2.2: Corrección de la orientación. Ubicado en el archivo <i>Firmware/src/lib/ecl/EKF/ekf.cpp</i>	6
2.3	Código 2.3: Cálculo del tamaño del buffer. Ubicado en el archivo <i>Firmware/src/lib/ecl/EKF/estimator_interface.cpp</i>	6
3.1	Código 3.1: Rotación en PX4 de la posición suministrada por la visión	18



# Bibliografía

---

- [1] Aaron Walawalkar. *NHS using drones to deliver coronavirus kit between hospitals*. 2020. URL: <https://www.theguardian.com/technology/2020/oct/17/nhs-drones-deliver-coronavirus-kit-between-hospitals-essex> (visitado 25-11-2020).
- [2] Douglas Gimesy. *Drones and thermal imaging: saving koalas injured in the bushfires*. 2020. URL: <https://www.theguardian.com/australia-news/gallery/2020/feb/11/drones-thermal-imaging-australia-koalas-bushfire-crisis> (visitado 25-11-2020).
- [3] *The art of landing really precisely without GPS*. 2017. URL: <https://www.everdrone.com/news/2017/11/21/the-art-of-landing-really-precisely-without-gps> (visitado 07-11-2020).
- [4] Mohammad Fattahí Sani y Ghader Karimian. «Automatic navigation and landing of an indoor AR. drone quadrotor using ArUco marker and inertial sensors». En: *2017 International Conference on Computer and Drone Applications (IConDA)*. IEEE. 2017, págs. 102-107.
- [5] Shuo Yang y col. «Precise quadrotor autonomous landing with SRUKF vision perception». En: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, págs. 2196-2201.
- [6] Isidro Jesús Arias Sánchez. «Control de un tiltrotor implementado en el autopiloto de código abierto PX4». En: (2019). URL: <https://idus.us.es/handle/11441/94450>.
- [7] Sergio Garrido-Jurado y col. «Automatic generation and detection of highly reliable fiducial markers under occlusion». En: *Pattern Recognition* 47.6 (2014), págs. 2280-2292. doi: <https://doi.org/10.1016/j.patcog.2014.01.005>. URL: <http://www.sciencedirect.com/science/article/pii/S0031320314000235>.

