

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:

Resumen

[Pendiente]

Abstract

[Pendiente]

Índice Abreviado

Índice

Notación

P	Posición del vehículo en ejes inerciales
P_{sp}	Posición deseada del vehículo en ejes inerciales
V	Velocidad del vehículo en ejes inerciales
V_{sp}	Velocidad deseada del vehículo en ejes inerciales
F_{NED}	Fuerzas deseadas en el vehículo en ejes inerciales
$\phi_{sp}, \theta_{sp}, \psi_{sp}$	Orientación deseada expresada en angulos de euler
q_{sp}	Orientación deseada expresada en cuaterniones
τ_{sp}	Pares deseados en ejes cuerpo
Ω	Velocidad angular en ejes cuerpo
Ω_{sp}	Velocidad angular deseada en ejes cuerpo
p, q, r	Componentes de Ω
R	Matriz de rotación del sistema de coordenadas del vehículo a los ejes inerciales
R_{sp}	Matriz de rotación de los ejes deseados del vehículo a los ejes inerciales

Acrónimos

<i>UAV</i>	Vehículo aéreo no tripulado
<i>GCS</i>	Estación de control terrestre
<i>NED</i>	Ejes norte, este y abajo

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 ocio o para 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 sus fases de vuelo, como 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 y 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. Por ejemplo, se podría programar un vuelo de reconocimiento cuando se detecte un intruso en una propiedad. También permitiría manipular objetos como la recogida y depósito de paquetes.

El interés de realizar estas tareas de forma autónoma no solo está en que cualquiera pueda hacer uso del vehículo, sin necesitar licencia ni habilidades especiales, si no que también hace el sistema más escalable. Si se quisiera por ejemplo, instalar un sistema de reparto de paquetes mediante vehículos aéreos, y la flota es 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 el sistema sea más óptimo. 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 las personas a estar trabajando en horas de descanso.

Si se disponen de los suficientes recursos existe la tecnología para conseguirlo, por ejemplo mediante balizas sonoras, que son usadas en interiores de fábricas o si se opera en el exterior, existe la posibilidad de *navegación cinética satelital en tiempo real* (RTK). Otra solución que no necesita de instalación en el entorno, son las cámaras o Lidars, 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 o bien sus sensores son caros o bien tienen una alta carga computacional que en necesitar mayor capacidad de cómputo.

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. Estos contienen figuras que son fácilmente detectables procesando una imagen de dicho marcador, por ejemplo los contornos de un cuadrilátero. Además, para su procesamiento se usará uno de los ordenadores embebidos más baratos del mercado

La empresa *Everdrone* es una de las que más avanzado en este campo.



En este trabajo se propone conseguir ese posicionamiento mediante marcadores visuales. No como única fuente de posición, sino combinándola con otras como el GPS.

En este trabajo en primer lugar, en el capítulo ?? se hace un estudio de una parte clave para el posicionamiento con marcadores, que es el estimador de estados. En el capítulo ?? se explica cómo se ha implementado este posicionamiento, mostrando los componentes utilizados y explicando el código escrito.

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 [2] 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, si no 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 que 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. La medida 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,

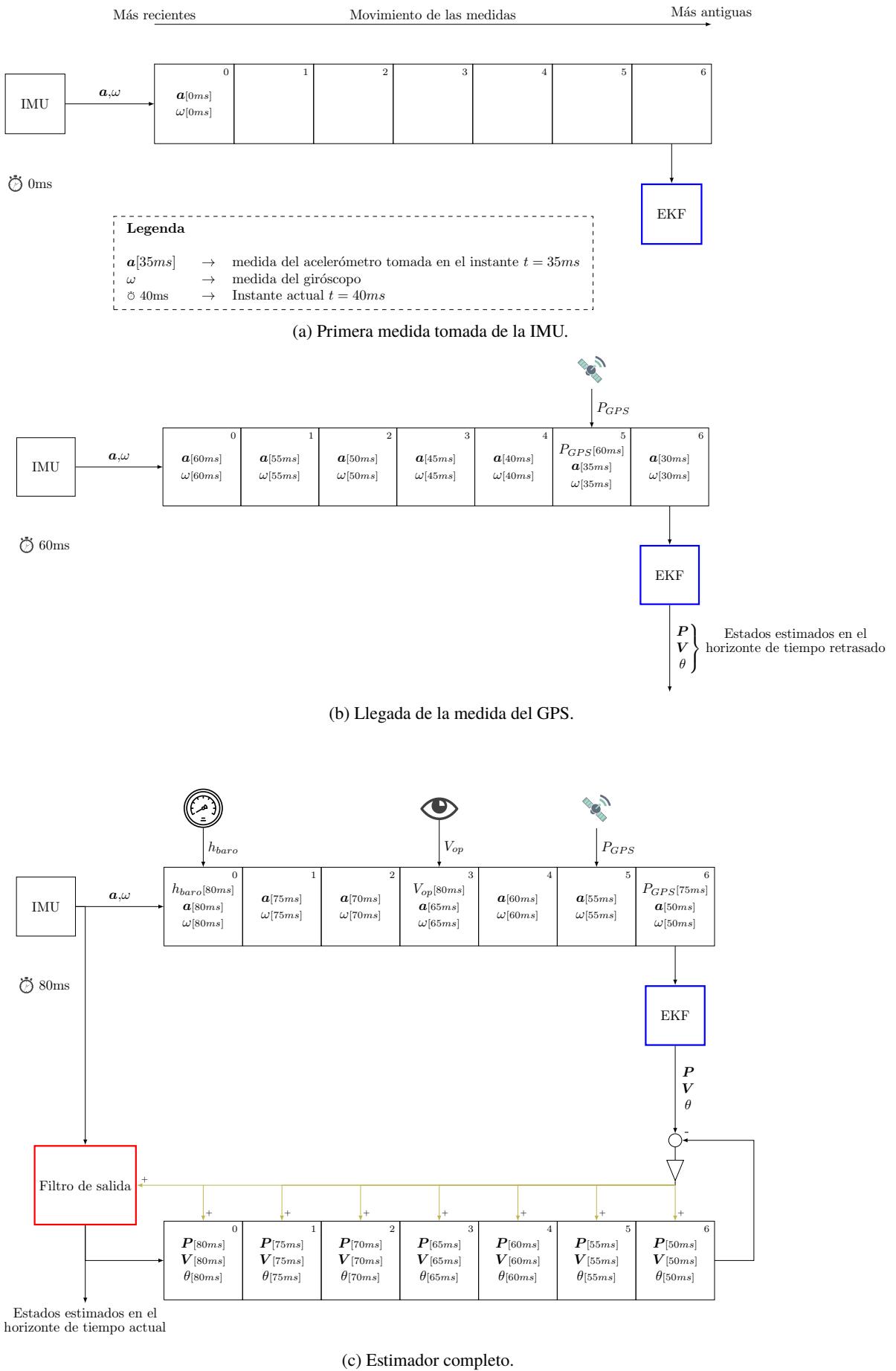


Figura 2.1 Manejo de medidas retrasadas.

como por ejemplo para alimentar al controlador de orientación, por esta razón se le denomina filtro de salida. 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 almacenado 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 de código de PX4 que implementan lo anteriormente descrito más algunos detalles que he se han omitido en el apartado anterior para que fuese más fácil su comprensión.

En el apartado anterior se explicó que la error entre los estados estimados en el horizonte de tiempo retrasado y los estados del buffer de salida, se atenuaban (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 ver 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

```

```

514     _output_new = _output_buffer.get_newest();
515 }
```

En el código anterior no ha aparecido la corrección de la orientación, y esto es porque requieren 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, 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 EKF para modelo bidimensional

Se buscará un modelo discreto de espacio de estados descrito de la siguiente manera:

$$X_{k+1} = f(X_k) \quad (2.1)$$

Se va aplicar a un quadrotor en 2 dimensiones, pero el modelo al ser cinemático, se podría aplicar a cualquier otro móvil.

Estados:

$$X = \begin{bmatrix} x \\ y \\ V_x \\ V_y \\ \theta \end{bmatrix} \quad (2.2)$$

Modelo de predicción cinemático:

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

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix}_{k+1} = \begin{bmatrix} V_x \\ V_y \end{bmatrix}_k + \Delta t \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \mathbf{a} + \begin{bmatrix} 0 \\ -m g \end{bmatrix} \Delta t \quad (2.4)$$

$$\theta_{k+1} = \theta_k + \Delta t \omega \quad (2.5)$$

Jacobiano del modelo de predicción con respecto a los estados:

$$F = \frac{\partial f}{\partial X} \Big|_{X_{k-1}} = \begin{bmatrix} 1 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & \Delta t (-a_x \sin \theta_{k-1} + a_y \cos \theta_{k-1}) \\ 0 & 0 & 0 & 1 & \Delta t (-a_x \cos \theta_{k-1} - a_y \sin \theta_{k-1}) \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.6)$$

Jacobiano del modelo de predicción con respecto a la aceleración y a la velocidad angular.

$$G = \frac{\partial f}{\partial \mathbf{a}, \omega} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \Delta t \cos \theta & \Delta t \sin \theta & 0 \\ -\Delta t \sin \theta & \Delta t \cos \theta & 0 \\ 0 & 0 & \Delta t \end{bmatrix} \quad (2.7)$$

Matriz de covarianzas de la predicción:

$$Q = G \begin{bmatrix} \sigma_a^2 & 0 & 0 \\ 0 & \sigma_a^2 & 0 \\ 0 & 0 & \sigma_\omega^2 \end{bmatrix} G^T \quad (2.8)$$

Donde σ_a^2 y σ_ω^2 son las varianzas del acelerómetro y del giróscopo, que se pueden hallar experimentalmente o viendo la hoja de datos de los sensores.

2.3 Simulación del quadrotor y del estimador

En este apartado se implementará el filtro explicado en este capítulo y pondrá a prueba con un simulador de un quadrotor. Tanto el estimador como el simulador estarán programados en lenguaje Python. 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 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.

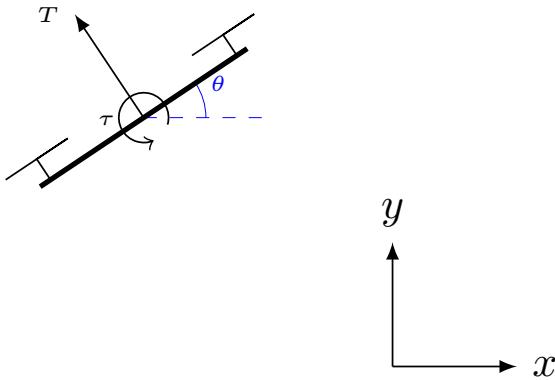


Figura 2.2 Quadrotor en dos dimensiones.

Para simular el quadrotor se realiza una integración discreta de la segunda ley de Newton:

$$\ddot{\theta} = \frac{\tau}{I} \quad (2.9)$$

$$\dot{\theta} = \dot{\theta}_{i-1} + \Delta t \ddot{\theta} \quad (2.10)$$

$$\theta = \theta_{i-1} + \Delta t \dot{\theta} \quad (2.11)$$

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (2.12)$$

$$\mathbf{T}_{rot} = R \begin{bmatrix} 0 \\ T \end{bmatrix} \quad (2.13)$$

$$\mathbf{F}_g = \begin{bmatrix} 0 \\ -mg \end{bmatrix} \quad (2.14)$$

$$\mathbf{a} = \frac{\mathbf{T}_{rot} + \mathbf{F}_g}{m} \quad (2.15)$$

$$\mathbf{v} = v_{i-1} + \mathbf{a} \Delta t \quad (2.16)$$

$$\mathbf{p} = p_{i-1} + \mathbf{v} \Delta t \quad (2.17)$$

$$(2.18)$$

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.

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 giroscopio 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 estimación 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 impuso que tuviese 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 realizado 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 se le está compensando su retraso antes de fusionarlo con las medidas de la IMU. 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, que tiene forma de diente de sierra debido a la llegada periódica de las medidas del GPS, comparado con los errores reales: el del filtro de salida y el del EKF, que se calcula como la diferencia con el groundtruth retrasado la misma cantidad de tiempo que el filtro de Kalman extendido. En este último caso, el error

estimado no se aleja mucho del real, pero en el caso del filtro de salida este error es mayor.

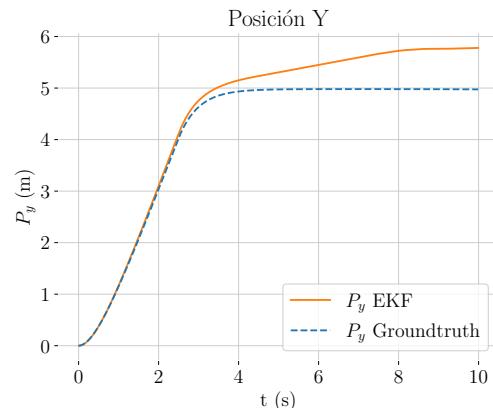


Figura 2.3 EKF no ejecuta la fase de actualización.

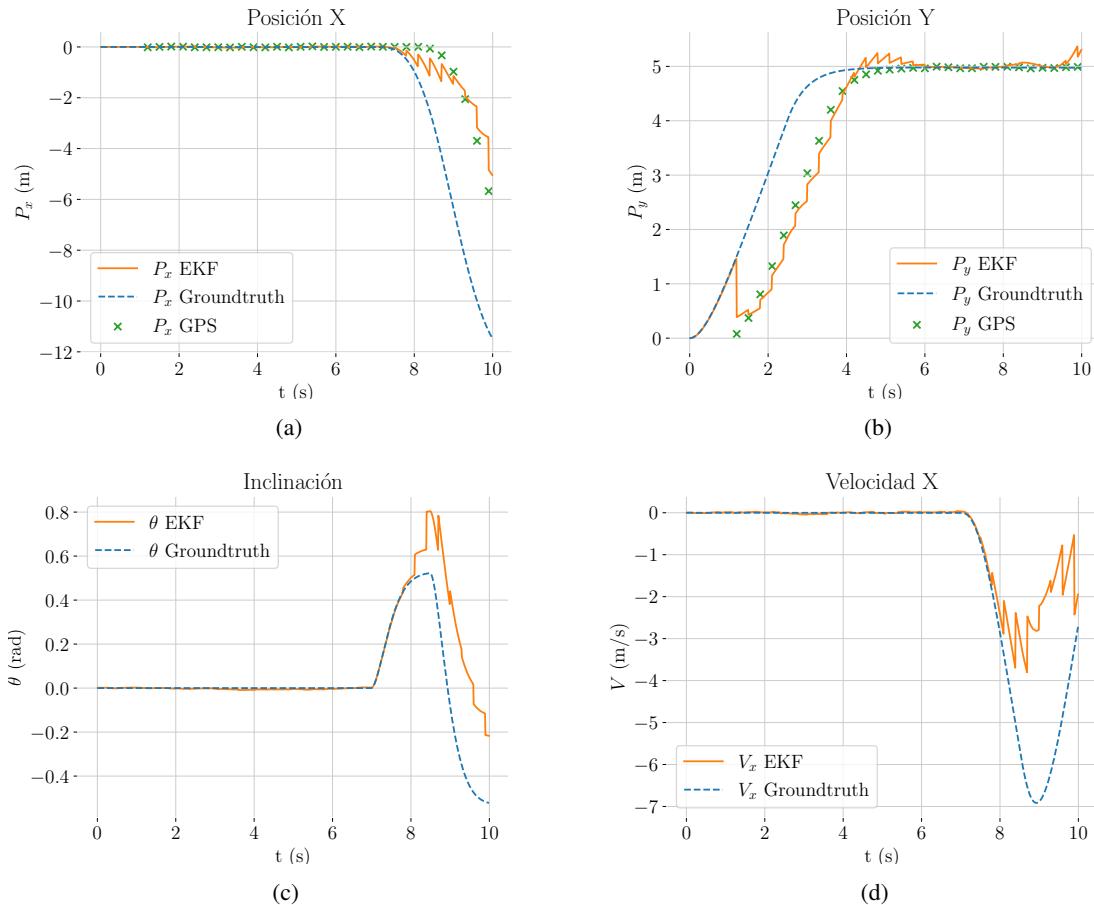
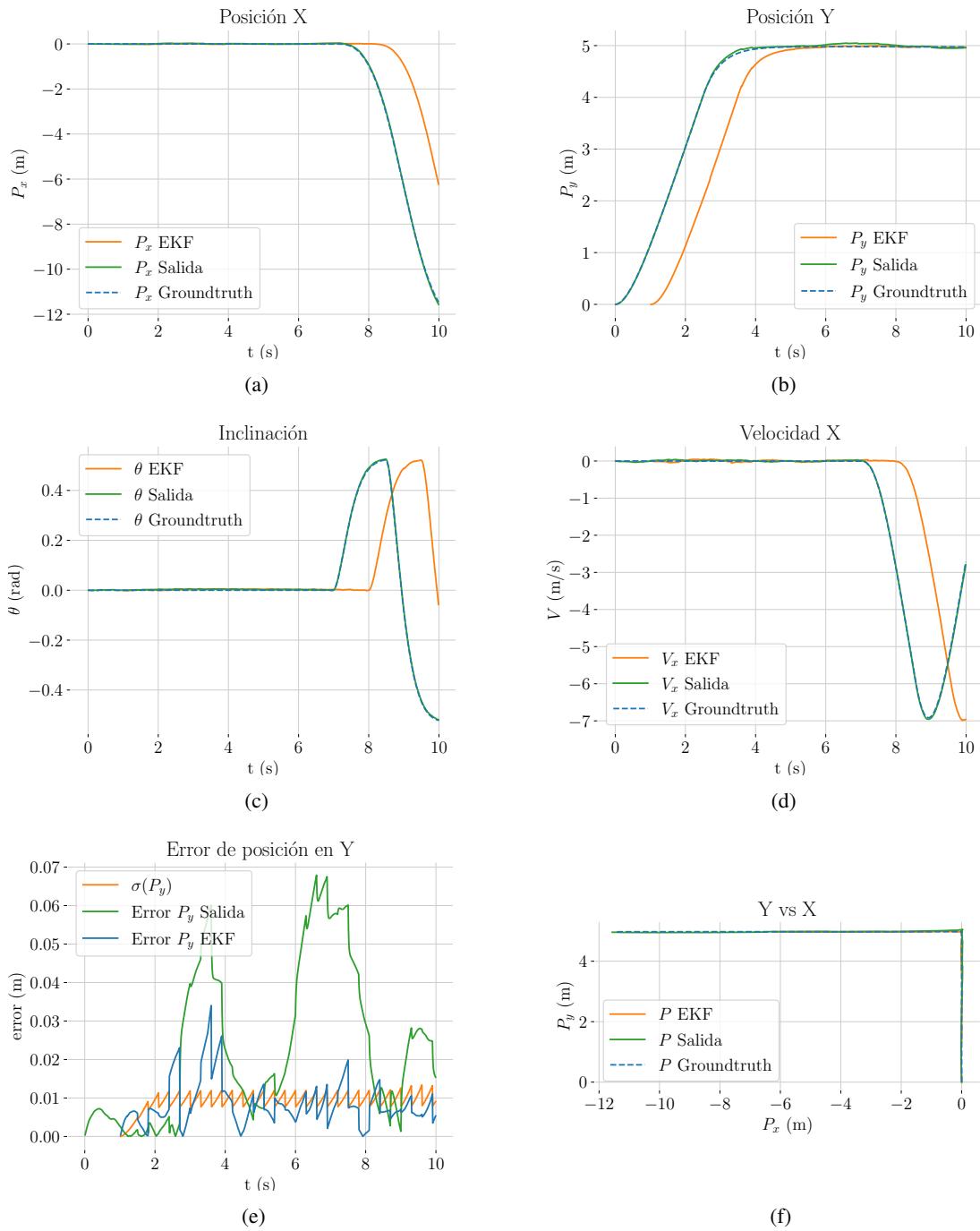


Figura 2.4 Fusión del GPS con retraso.

**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 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 autopiloto¹.
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.
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 HV PM (High-Voltage Power Module)*. Regulador de voltaje para alimentar el autopiloto. Además, lee el voltaje y la corriente que suministra la batería.
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.

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

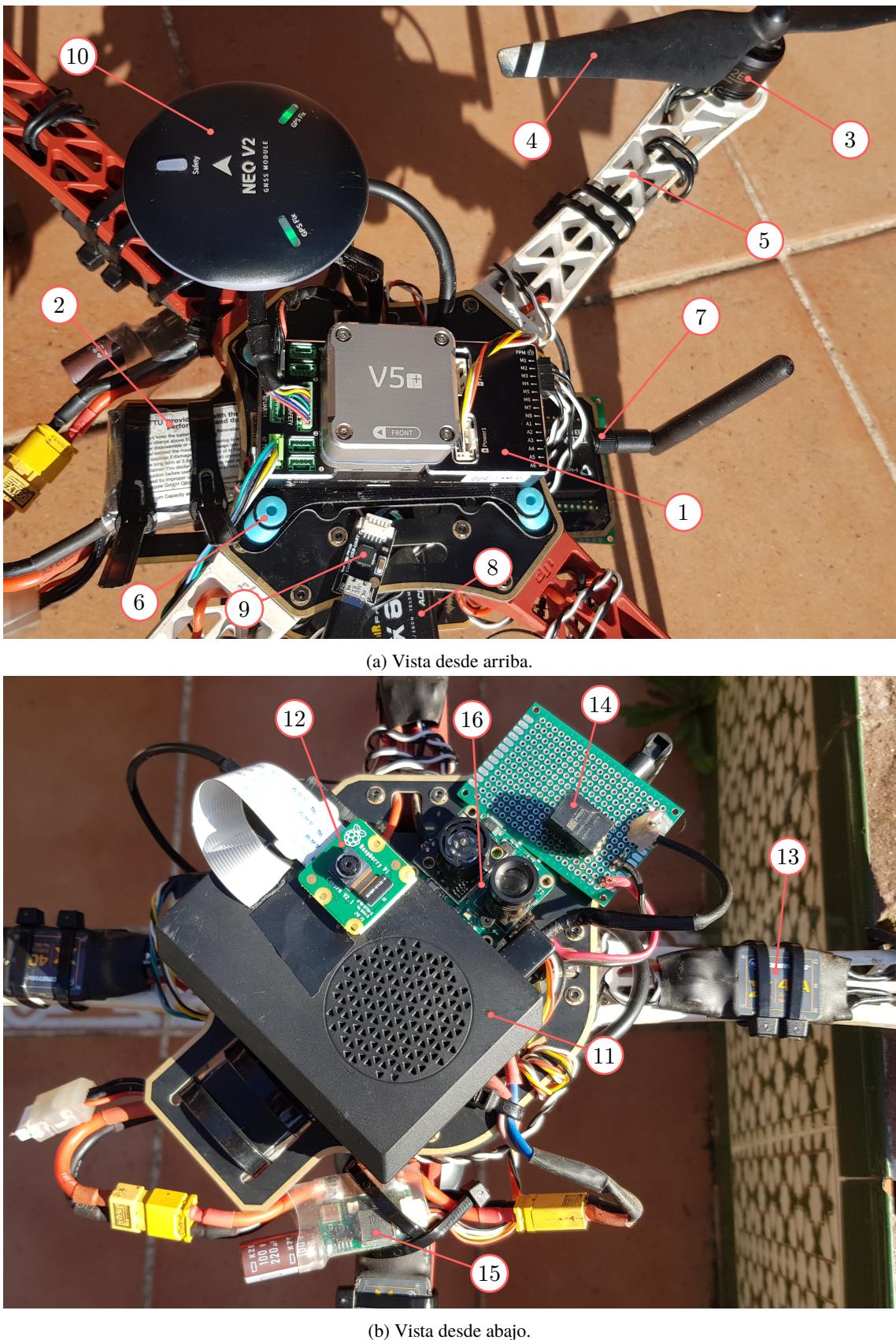


Figura 3.1 Componentes del quadrotor.

15. *RS PRO K7805-2000R3L*. Reductor de voltaje de 5V y 2A. Este se utilizará para alimentar al 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 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 DMA². 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 está explicado en [3].

4. Estimación de la posición:

La estimación de la posición y la orientación se realiza tomando las esquinas de un marcador obtenido 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 que 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 de unos puntos tridimensionales en la imagen. 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.

5. Inversión de posición y orientación:

² 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>

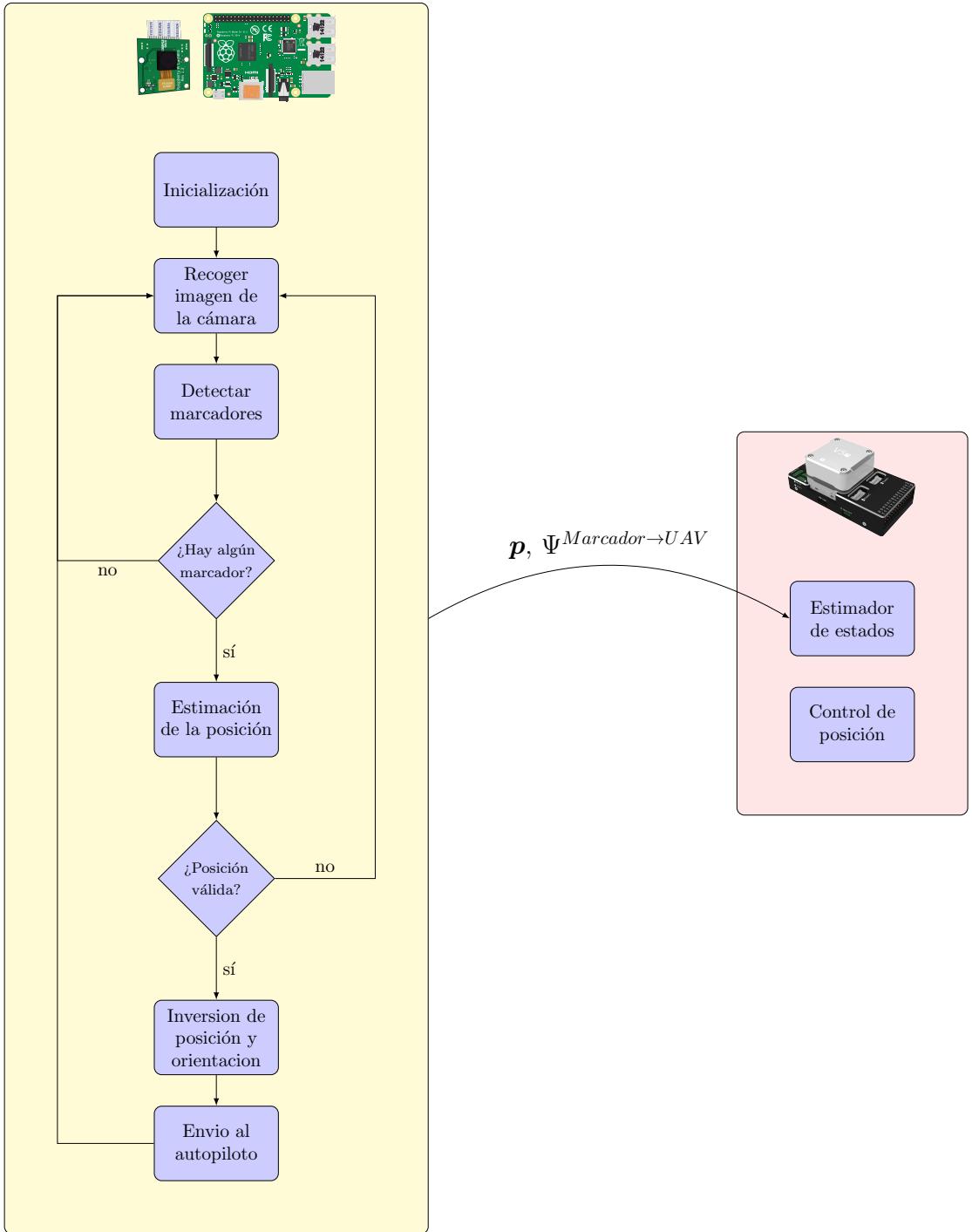


Figura 3.2 A la izquierda: diagrama de flujo del programa que se corre en el ordenador embebido, a la derecha: algunas de las tareas del autopiloto.

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 hasta lo que necesita el autopiloto. 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 ser una matriz de rotación, se puede obtener mediante su traspuesta:

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

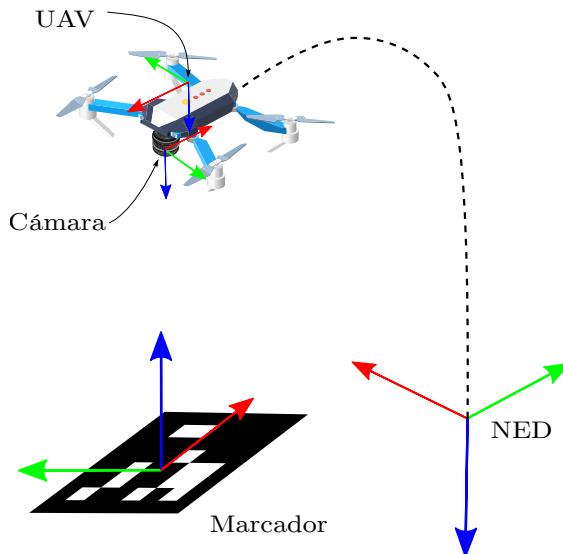


Figura 3.3 Sistemas de referencia presentes en el problema.

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 visto desde el sistema de referencia del UAV:

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

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

Es muy importante que la multiplicación se realice en ese orden, ya que la multiplicación de matrices no es commutativa. 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 en a partir de la línea ?? 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.

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

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

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

Siendo *NED* el sistema de referencia centrado en la posición que partió el UAV y *NED'* uno que es paralelo a este último 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 *
→      _ev_sample_delayed.quat.inversed()).normalized());
1461     _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 [2] estos forman una estructura en cascada cuyo controlador de mayor nivel es el de posición.

3.3 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 o elegimos unos parámetros de acuerdo a expresiones analíticas o simulaciones. Despues 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. 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. Llénose 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 análisis de resultados.

3.3.1 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 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. Antes de estimar la posición de la cámara con respecto al marcador se debe de conocer el tamaño de este.
- 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 video 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 estos 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, 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 un programa 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 para el programa detector de marcadores 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 como valores que no cambian a lo largo de un periodo largo de tiempo.

3.3.2 Registro de resultados

Para analizar los resultados primero hay que registrarlos. Funcionalidades implementadas del registro de resultados:

- Guardar la posición y orientación estimadas en un archivo.
Representación de estas en una gráfica temporal Para verificar el desempeño de la estimación de la posición y orientación, lo ideal sería tener un groundtruth, por ejemplo con 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 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 ??). 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 y resulta útil para comprobar que la detección de sus esquinas se realiza de manera correcta.
- Scripts en 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



Figura 3.4 Superposición de los ejes de referencia del marcador y del cuadrilátero que lo rodea (trazado en verde).

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, y 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 video 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

3.4 Resultados experimentales

En esta sección se mostrarán gráficas de los datos generados en un vuelo en el que se estaba fusionando la posición de la visión. Están desactivados tanto el GNSS como el flujo óptico

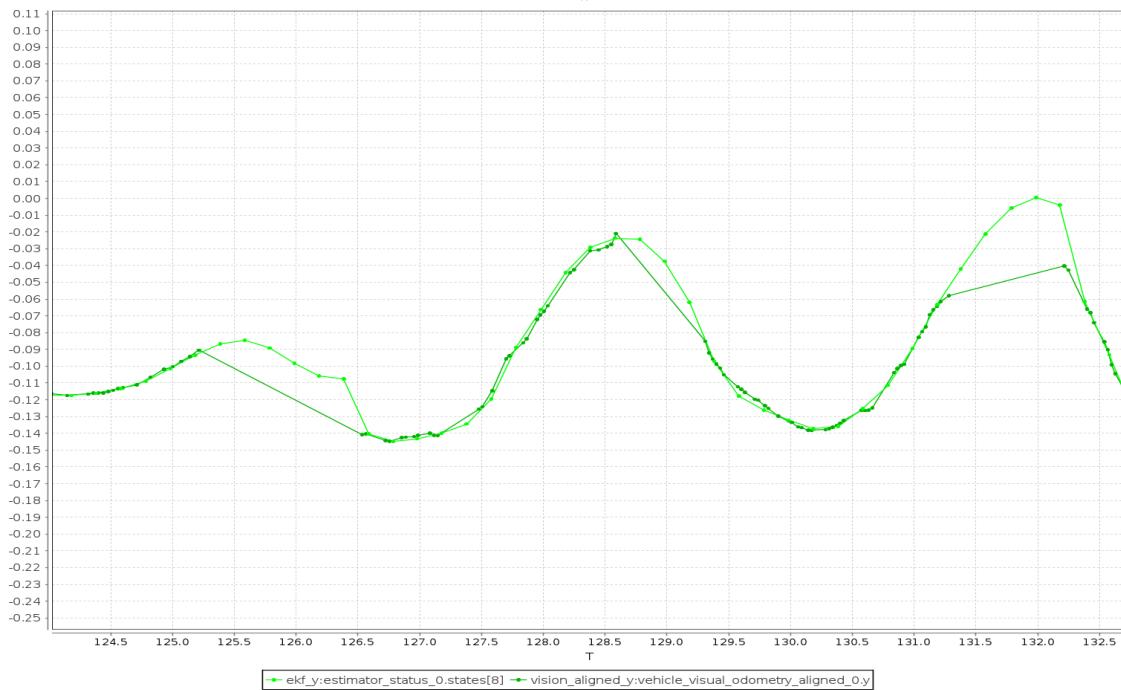
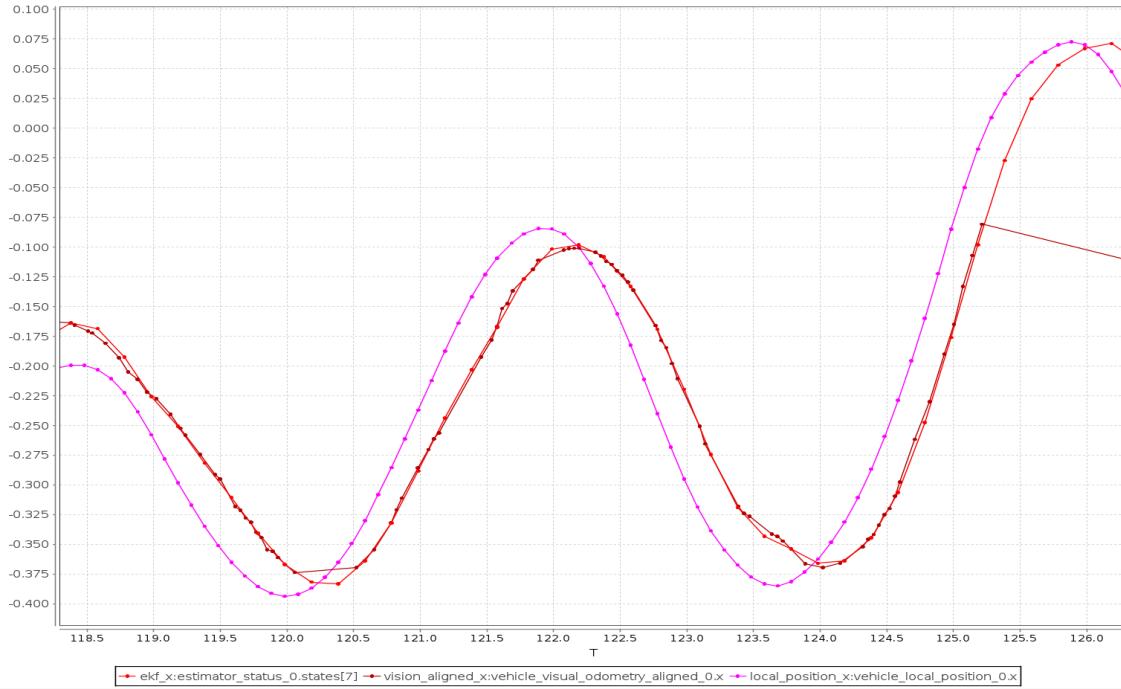
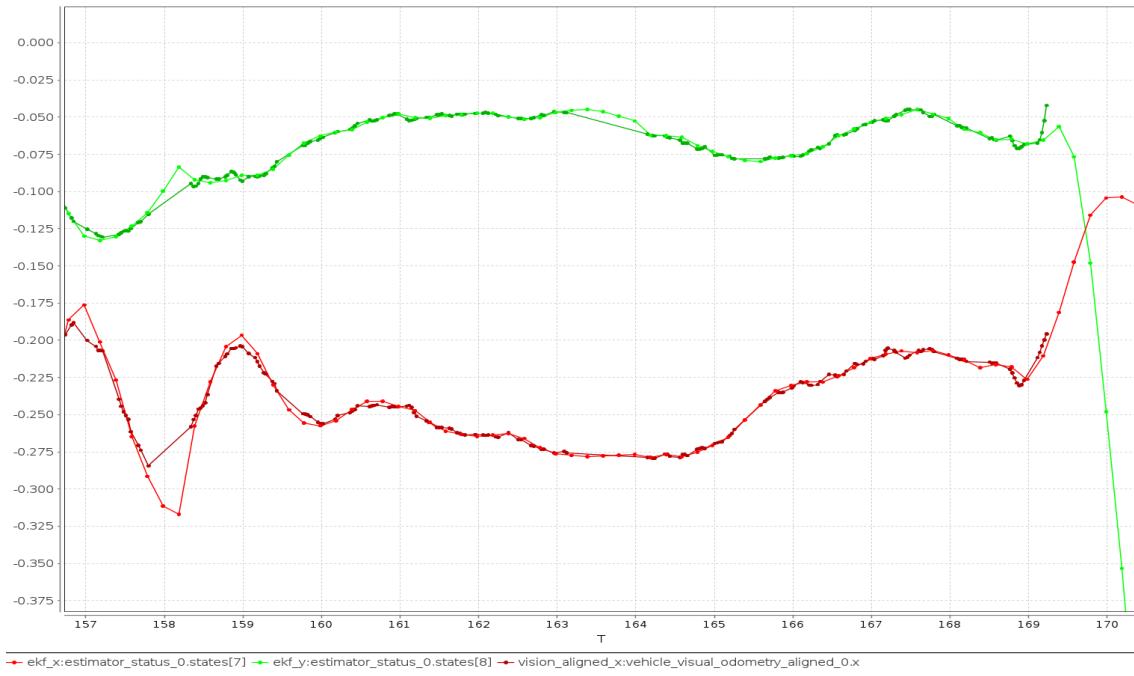


Figura 3.5 Modo altitud.

(a) P_x y P_y .**Figura 3.6** Modo position.

4 Conclusiones

5 Trabajos futuros

- Disminuir la necesidad de colocar marcadores cada poca distancia. Se podría usar lentes ojo de pez, varias cámaras (abajo y delante) o un gimbal.

Apéndice A

Simulador del estimador de estados

El siguiente archivo también puede verse y descargarse en el repositorio 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

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
13 using std::chrono::milliseconds;
14 using std::chrono::seconds;
15 using std::this_thread::sleep_for;
16
17 using namespace std;
18
19 #define TELEMETRY_CONSOLE_TEXT "\033[34m" // Turn text on console blue
20
21
22 #define DEBUG
23
24
25 #define UPDATE_DEBUG_RATE 30 // Cada cuantas iteraciones se calculan e imprimen las estadísticas
26
27 static bool readParameters(string filename, bool &mav_connect, bool &log_file, int &loop_period_ms,
28   → bool &wait_key, int &wait_key_mill) {
29     FileStorage fs(filename, FileStorage::READ);
30     if(!fs.isOpened())
31       return false;
32     mav_connect = (string)fs["mav_connect"]=="true";
33     log_file = (string)fs["log_file"]=="true";
34     loop_period_ms = (int)fs["loop_period_ms"];
35     wait_key_mill = (int)fs["wait_key_mill"];
36     wait_key = (string)fs["wait_key"]=="true";
37     cout << "Parámetros generales:" << endl;
38     cout << "\tConexión mavlink:\t\t\t" << mav_connect << endl;
39     cout << "\tLog de medidas:\t\t\t\t" << log_file << endl;
40     cout << "\tPeriodo mínimo de actualización:\t" << loop_period_ms << endl;
41     if (wait_key){
42       cout << "\tEspera a la presión de una tecla:\t" << wait_key_mill << endl;
43     }
44     cout << endl;
45     return true;
}
```

```

46
47
48
49 int main()
50 {
51     cout << "-----" << endl;
52     cout << "-----Vision position estimator-----" << endl;
53     cout << "-----" << endl;
54     cout << endl;
55
56     /* Startup python script for logging */
57     int res=system("python3 ..//python_scripts/startup.py");
58     if (res!=0){
59         cout << "El script de inicio ha fallado con código " << res << endl;
60         exit(1);
61     }
62
63     bool mav_connect, log_file, wait_key;
64     int loop_period_ms, wait_key_mill;
65     readParameters("../vision_params.yml", mav_connect, log_file, loop_period_ms, wait_key,
66     ↵ wait_key_mill);
67
68     CommunicationClass commObj;
69     if (mav_connect)
70         commObj.init();
71
72     VisionClass visionMarker;
73
74     double total_time = 0;
75     int totalIterations = 0;
76     int n_position_get = 0;
77     auto x = std::chrono::steady_clock::now() + std::chrono::milliseconds(loop_period_ms);
78     double tick_global_ant = (double)getTickCount();
79     Eigen::Vector3d pos, euler_angles ;
80
81     double seconds_init = (double)getTickCount()/getTickFrequency();
82
83     std::ofstream myfile;
84     if (log_file){
85         myfile.open("../results/latest/log.csv");
86         myfile << "px" << "," << "py" << "," << "pz" << "," << "roll" << "," << "pitch" << "," <<
87         ↵ "yaw" << "," << "t" << "\n";
88     }
89
90     /*** Main Loop ***/
91     while(true){
92
93         // TODO: move perf counter to vision class
94         double tick0 = (double)getTickCount();
95         // Grab image and exists if there is no one
96         if (!visionMarker.grab_and_retrieve_image()) break;
97         double tick1 = (double)getTickCount();
98         // detect markers
99         bool found_marker = visionMarker.detect_marker(pos, euler_angles);
100        double tick2 = (double)getTickCount();
101
102        if (found_marker){
103            if (mav_connect)
104                commObj.send_msg(pos, euler_angles);
105            n_position_get++;
106        }
107
108        #ifdef DEBUG
109        // Update counters
110        double tick_global_act = (double)getTickCount();
111        double execution_time = (tick_global_act - tick_global_ant) / getTickFrequency();
112        tick_global_ant = tick_global_act;
113        double execution_time_detect = (tick2-tick1) / getTickFrequency();
114        double execution_time_video_grab_and_ret = (tick1-tick0) / getTickFrequency();
115        total_time += execution_time;

```

```

115     totalIterations++;
116
117     /* Print data every 30 frames = 1 seg approx*/
118     if(totalIterations % UPDATE_DEBUG_RATE == 0) {
119         cout << "Image grabbing and retrieving= " << execution_time_video_grab_and_ret * 1000
120         → << " ms " << endl;
121         cout << "Marker detection= " << execution_time_detect * 1000 << " ms " << endl;
122         cout << "Execution time = " << execution_time * 1000 << " ms "
123         → << "(Mean = " << 1000 * total_time / float(UPDATE_DEBUG_RATE) << " ms)" << endl;
124         cout << "Frames with position = " << n_position_get/float(UPDATE_DEBUG_RATE) * 100 << "
125         → \% " << endl;
126
127         if(found_marker){
128             cout << "Estimated position:\t" << pos[0] << "\t" << pos[1] << "\t"
129             → << pos[2] << endl;
130             cout << "Estimated orientation:\t" << euler_angles[0] << "\t" << euler_angles[1]
131             → << "\t" << euler_angles[2] << endl;
132         }
133     #endif
134
135
136     if (log_file){
137         double seconds = getTickCount() / getTickFrequency() - seconds_init;
138         myfile << pos[0] << "," << pos[1] << "," << pos[2] << "," << euler_angles[0] << ","
139         → << euler_angles[1] << "," << euler_angles[2] << "," << seconds << "\n";
140
141     if (loop_period_ms!=0){
142         std::this_thread::sleep_until(x);
143         x = std::chrono::steady_clock::now() + std::chrono::milliseconds(loop_period_ms);
144     }
145
146     if (wait_key){
147         int key = waitKey( wait_key_mill );
148         if(key == 27) break; // exits if esc is pressed in window
149     }
150 }
151
152 if (log_file)
153     myfile.close();
154
155 res=system("python3 ../python_scripts/shutdown.py");
156 if (res!=0){
157     cout << "El script de apagado ha fallado con código " << res << endl;
158     exit(1);
159 }
160 std::cout << "Finished..." << std::endl;
161 return EXIT_SUCCESS;
162 }
```

B.2 vision_params.yml

```

1  %YAML:1.0
2  #dict_type: 10    # 6x6 256
3  dict_type: 0      # 6x6 256
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 refindStrategy: 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: false
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: true
50 log_file: true
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         VisionClass(){
19             bool readOk = readVisionParameters("../vision_params.yml");
20             if(!readOk) {
21                 cerr << "Invalid general vision parameters file" << endl;
22                 exit(0);
```

```

22     }
23
24     /*** Vision setup ***/
25     detectorParams = aruco::DetectorParameters::create();
26     readOk = readDetectorParameters( "../detector_params.yml", detectorParams );
27     if(!readOk) {
28         cerr << "Invalid detector parameters file" << endl;
29         exit(0);
30     }
31
32     dictionary =
33         → aruco::getPredefinedDictionary(aruco::PREDEFINED_DICTIONARY_NAME(dict_type));
34
35     readOk = readCameraParameters(calibration_file, camMatrix, distCoeffs);
36     if(!readOk) {
37         cerr << "Invalid camera file" << endl;
38         exit(0);
39     }
40
41     if (charuco){
42         charucoboard = aruco::CharucoBoard::create(squaresX, squaresY, square_length,
43             → marker_length_ch, dictionary);
44         board = charucoboard.staticCast<aruco::Board>();
45         axisLength = 0.5f * ((float)min(squaresX, squaresY) * (square_length));
46     }
47     else{
48         axisLength = 0.5f * marker_length;
49     }
50
51     if (video_file!=""){
52         inputVideo.open(video_file);
53     }
54     else{
55         inputVideo.open(0);
56         inputVideo.set(CAP_PROP_FRAME_WIDTH, frame_width);
57         inputVideo.set(CAP_PROP_FRAME_HEIGHT, frame_height);
58         string cmd;
59         if (fps!=0){
60             cmd=(string)"v4l2-ctl -d /dev/video0 -p "+ to_string(fps);
61             const char* aux1=cmd.data();
62             system(aux1);
63         }
64         if (exposure_time!=0){
65             cmd=(string)"v4l2-ctl -d /dev/video0 -c auto_exposure=1 -c
66             → exposure_time_absolute="+ to_string(exposure_time);
67             const char* aux2=cmd.data();
68             system(aux2);
69         }
70         else{
71             system("v4l2-ctl -d /dev/video0 -c auto_exposure=0");
72         }
73         system(" v4l2-ctl -V");
74     }
75     /* Test an image */
76     grab_and_retrieve_image();
77     cout << "Ancho de la imagen:\t" << image.cols << endl;
78     cout << "Alto de la imagen:\t" << image.rows << endl << endl;
79
80     int grab_and_retrieve_image(){
81         int res = inputVideo.grab();
82         inputVideo.retrieve(image);
83         return res;
84     }
85
86     bool detect_marker(Eigen::Vector3d &pos, Eigen::Vector3d &eul);
87
88     private:
89         bool readDetectorParameters(string filename, Ptr<aruco::DetectorParameters> &params);
90         bool readVisionParameters(string filename);

```

```

90     void InvertPose(Eigen::Vector3d &pos, Eigen::Vector3d &eul, Vec3d &rvec, Vec3d &tvec);
91     bool readCameraParameters(string filename, Mat &camMatrix, Mat &distCoeffs);
92     Eigen::Vector3d rotationMatrixToEulerAngles(Eigen::Matrix3d &R);
93
94     Mat image;
95     Mat imageCopy;
96     VideoCapture inputVideo;
97     Ptr<aruco::Dictionary> dictionary;
98     Ptr<aruco::DetectorParameters> detectorParams;
99     Mat camMatrix;
100    Mat distCoeffs;
101    string calibration_file;
102    string video_file;
103    bool show_rejected;
104    float marker_length;
105    int dict_type;
106    float axisLength;
107    bool open_window;
108    bool write_images;
109    // camera config
110    int exposure_time;
111    int fps;
112    int frame_width;
113    int frame_height;
114    // diamond specific
115    bool diamond;
116    bool autoScale;
117    // charuco specific
118    bool charuco;
119    bool refindStrategy;
120    int squaresX;
121    int squaresY;
122    float square_length;
123    float marker_length_ch;
124    Ptr<aruco::CharucoBoard> charucoboard;
125    Ptr<aruco::Board> board;
126    // logging
127    int totalIterations=0;
128
129    };
130
131
132    // Calculates rotation matrix to euler angles
133    Eigen::Vector3d VisionClass::rotationMatrixToEulerAngles(Eigen::Matrix3d &R)
134    {
135
136        float sy = sqrt(R(0,0) * R(0,0) + R(1,0) * R(1,0) );
137
138        bool singular = sy < 1e-6; // If
139
140        float x, y, z;
141        if (!singular)
142        {
143            x = atan2(R(2,1) , R(2,2));
144            y = atan2(-R(2,0), sy);
145            z = atan2(R(1,0), R(0,0));
146        }
147        else
148        {
149            x = atan2(-R(1,2), R(1,1));
150            y = atan2(-R(2,0), sy);
151            z = 0;
152        }
153        return Eigen::Vector3d(x, y, z);
154    }
155
156
157    bool VisionClass::readCameraParameters(string filename, Mat &camMatrix, Mat &distCoeffs) {
158        FileStorage fs(filename, FileStorage::READ);
159        if(!fs.isOpened())
160            return false;

```

```

161     fs["camera_matrix"] >> camMatrix;
162     fs["distortion_coefficients"] >> distCoeffs;
163     return true;
164 }
165
166
167 bool VisionClass::detect_marker(Eigen::Vector3d &pos, Eigen::Vector3d &eul){
168
169     vector< int > ids, charucoIds;
170     vector< vector< Point2f > > corners, rejected;
171     vector< Vec3d > rvecs, tvecs;
172     Vec3d rvec, tvec;
173     vector< Point2f > charucoCorners;
174     vector< vector< Point2f > > diamondCorners;
175     vector< Vec4i > diamondIds;
176
177
178     aruco::detectMarkers(image, dictionary, corners, ids, detectorParams, rejected);
179
180     bool found_marker=ids.size() > 0;
181     bool valid_pose = false;
182     int interpolatedCorners = 0;
183
184     if (charuco){
185         if(refindStrategy)
186             aruco::refineDetectedMarkers(image, board, corners, ids, rejected,
187                                         camMatrix, distCoeffs);
188
189         // interpolate charuco corners
190         if(found_marker)
191             interpolatedCorners =
192                 aruco::interpolateCornersCharuco(corners, ids, image, charucoboard,
193                                                 charucoCorners, charucoIds, camMatrix,
194                                                 → distCoeffs);
195         if ((int)ids.size()==17){
196             // estimate charuco board pose
197             valid_pose = aruco::estimatePoseCharucoBoard(charucoCorners, charucoIds, charucoboard,
198                                              camMatrix, distCoeffs, rvec, tvec);
199         }
200         else{
201             valid_pose= false;
202         }
203     }
204     else if (diamond){
205         if (found_marker){
206             aruco::detectCharucoDiamond(image, corners, ids,
207                                         square_length / marker_length_ch, diamondCorners,
208                                         → diamondIds,
209                                         camMatrix, distCoeffs);
210
211             if(!autoScale) {
212                 aruco::estimatePoseSingleMarkers(diamondCorners, square_length, camMatrix,
213                                                 distCoeffs, rvecs, tvecs);
214             } else {
215                 // if autoscale, extract square size from last diamond id
216                 //for(unsigned int i = 0; i < diamondCorners.size(); i++) {
217                 //    float autoSquareLength = AUTO_SCALE_FACTOR * float(diamondIds[i].val[3]);
218                 //    vector< vector< Point2f > > currentCorners;
219                 //    vector< Vec3d > currentRvec, currentTvec;
220                 //    currentCorners.push_back(diamondCorners[i]);
221                 //    aruco::estimatePoseSingleMarkers(currentCorners, autoSquareLength,
222                 → camMatrix,
223                 //    distCoeffs, currentRvec, currentTvec);
224                 //    rvecs.push_back(currentRvec[0]);
225                 //    tvecs.push_back(currentTvec[0]);
226                 //}
227                 cout << "Autoscale todavía no implementado" << endl;
228                 exit(0);
229             }
230             if (tvecs.size()>0){
231                 rvec=rvecs[0];
232             }
233         }
234     }
235 }
```

```

229         tvec=tvecs[0];
230         valid_pose= true;
231     }
232 }
233 else{
234     if(found_marker){
235         aruco::estimatePoseSingleMarkers(corners, marker_length, camMatrix, distCoeffs, rvecs,
236             → tvecs);
237         valid_pose = true;
238         rvec=rvecs[0];
239         tvec=tvecs[0];
240     }
241 }
242
243 if (open_window){
244
245     image.copyTo(imageCopy);
246
247     if(found_marker){
248         aruco::drawDetectedMarkers(imageCopy, corners, ids);
249     }
250     if(interpolatedCorners > 0) {
251         Scalar color;
252         color = Scalar(255, 0, 0);
253         aruco::drawDetectedCornersCharuco(imageCopy, charucoCorners, charucoIds, color);
254     }
255     if (valid_pose){
256         aruco::drawAxis(imageCopy, camMatrix, distCoeffs, rvec, tvec, axisLength);
257     }
258
259     if(show_rejected && rejected.size() > 0)
260         aruco::drawDetectedMarkers(imageCopy, rejected, noArray(), Scalar(100, 0, 255));
261
262     imshow("out", imageCopy);
263     if (write_images){
264         char path [100];
265         sprintf(path,"..../results/latest/images/image%d.png", totalIterations);
266         imwrite(path,imageCopy);
267     }
268 //    waitKey( WAIT_KEY_MILL );
269 }
270
271 if (valid_pose){
272     InvertPose(pos, eul, rvec, tvec);
273 }
274 else{
275     pos[0]=pos[1]=pos[2]=NAN;
276     eul[0]=eul[1]=eul[2]=NAN;
277 }
278
279 totalIterations++;
280
281 return valid_pose;
282 }
283
284

```

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 void VisionClass::InvertPose(Eigen::Vector3d &pos, Eigen::Vector3d &eul, Vec3d &rvec, Vec3d
292   ↵ &tvec){//}
293
294   Eigen::Vector3d pos_marker_in_camera(tvec[0],tvec[1],tvec[2]);
295
296   // Transformación de vector de rotación a matriz de rotación
297   cv::Mat rot_mat;
298   Eigen::Matrix3d rot_mat_marker_from_camera;
299   Rodrigues(rvec,rot_mat);
300   cv::cv2eigen(rot_mat, rot_mat_marker_from_camera);
301
302   // La inversa de una matriz de rotación es igual a su traspuesta
303   Eigen::Matrix3d rot_mat_camera_from_marker = rot_mat_marker_from_camera.transpose();
304
305   // Se obtiene la posición del marcador en unos ejes paralelos al marcador centrados en la
306   // cámara
307   Eigen::Vector3d pos_marker_in_marker_axis =
308     rot_mat_camera_from_marker*pos_marker_in_camera;
309
310   // Si queremos que la posición esté centrada en el marcador y no en la cámara, es necesario
311   // negarla
312   pos = -pos_marker_in_marker_axis;
313
314   // Aquí debemos de tener en cuenta la rotación de la cámara con respecto al uav. Esta es de
315   // 180º alrededor del eje z.
316   // Queremos rotar en ejes absolutos y no en los ejes de rot_mat_marker_from_camera, por lo
317   // tanto premultiplicamos.
318   Eigen::Matrix3d rot_mat_camera_from_uav;
319   rot_mat_camera_from_uav << -1, 0, 0,
320           0, -1, 0,
321           0, 0, 1;
322   Eigen::Matrix3d rot_mat_marker_from_uav = rot_mat_camera_from_uav *
323     rot_mat_marker_from_camera;
324
325   // Se obtiene la orientación del uav visto desde el marcador
326   Eigen::Matrix3d rot_mat_uav_from_marker = rot_mat_marker_from_uav.transpose();
327
328   // Se obtiene los ángulos de Tait-Bryan en el orden Z-Y-X (ángulos de euler)
329   eul = rotationMatrixToEulerAngles(rot_mat_uav_from_marker);
330
331   #ifdef ROT_POS_ORI
332   // Transformaciones después de invertir la posición y la orientación. Queremos que la posición
333   // y la orientación del UAV
334   // esté expresado en un sistema de referencia con su eje z apuntando hacia abajo. El del
335   // marcador apunta hacia arriba, así
336   // que se rotará 180º en el eje x
337   Eigen::Matrix3d rot_mat_marker_from_NED;
338   rot_mat_marker_from_NED << 1, 0, 0,
339           0, -1, 0,
340           0, 0, -1;
341   Eigen::Matrix3d rot_mat_uav_from_NED = rot_mat_marker_from_NED *
342     rot_mat_uav_from_marker;
343
344   eul = rotationMatrixToEulerAngles(rot_mat_uav_from_NED);
345   pos = rot_mat_marker_from_NED * -pos_marker_in_marker_axis;
346   #endif
347
348 }
349
350 bool VisionClass::readVisionParameters(string filename) {
351   FileStorage fs(filename, FileStorage::READ);
352   if(!fs.isOpened())
353     return false;
354
355   fs["camera_parameters"] >> calibration_file;
356   fs["video_file"] >> video_file;
357   open_window = (string)fs["open_window"]=="true";
358   show_rejected = (string)fs["show_rejected"]=="true";
359   charuco = (string)fs["charuco"]=="true";
360   refindStrategy = (string)fs["refindStrategy"]=="true";
361   marker_length = (float)fs["marker_length"];

```

```

352     dict_type = (int)fs["dict_type"];
353     exposure_time = (int)fs["exposure_time"];
354     fps = (int)fs["fps"];
355     squaresX = (int)fs["squaresX"];
356     squaresY = (int)fs["squaresY"];
357     marker_length_ch = (float)fs["markerLength"];
358     square_length = (float)fs["squareLength"];
359     squaresY = (int)fs["squaresY"];
360     frame_height = (int)fs["frame_height"];
361     frame_width = (int)fs["frame_width"];
362     diamond = (string)fs["diamond"]=="true";
363     autoScale = (string)fs["autoScale"]=="true";
364     write_images = (string)fs["write_images"]=="true";
365
366     // Print them
367     cout << "Parámetros de la visión:" << endl;
368     cout << "\tEl fichero de calibración es:\t" << calibration_file << endl;
369     cout << "\tArchivo de video:\t" << video_file << endl;
370     cout << "\tActivación de la ventana:\t" << open_window << endl;
371     cout << "\tMostrar rechazados:\t" << show_rejected << endl;
372     cout << "\tTamaño del marcador:\t" << marker_length << endl;
373     cout << "\tTipo de diccionario:\t" << dict_type << endl;
374     cout << "\tTiempo de exposición:\t" << exposure_time << endl;
375     cout << "\tFPS:\t\t\t" << fps << endl;
376     if (charuco){
377         cout << endl;
378         cout << "Charuco:" << endl;
379         cout << "\trefindStrategy:\t\t" << refindStrategy << endl;
380         cout << "\tmarkerLength:\t\t" << marker_length_ch << endl;
381         cout << "\tsquareLength:\t\t" << square_length << endl;
382         cout << "\tsquaresX:\t\t" << squaresX << endl;
383         cout << "\tsquaresY:\t\t" << squaresY << endl;
384     }
385     else if(diamond){
386         cout << endl;
387         cout << "Diamond:" << endl;
388         cout << "\tautoScale:\t\t" << autoScale << endl;
389     }
390
391     cout << endl;
392     return true;
393 }
394
395 bool VisionClass::readDetectorParameters(string filename, Ptr<aruco::DetectorParameters> &params) {
396     FileStorage fs(filename, FileStorage::READ);
397     if(!fs.isOpened())
398         return false;
399     fs["adaptiveThreshWinSizeMin"] >> params->adaptiveThreshWinSizeMin;
400     fs["adaptiveThreshWinSizeMax"] >> params->adaptiveThreshWinSizeMax;
401     fs["adaptiveThreshWinSizeStep"] >> params->adaptiveThreshWinSizeStep;
402     fs["adaptiveThreshConstant"] >> params->adaptiveThreshConstant;
403     fs["minMarkerPerimeterRate"] >> params->minMarkerPerimeterRate;
404     fs["maxMarkerPerimeterRate"] >> params->maxMarkerPerimeterRate;
405     fs["polygonalApproxAccuracyRate"] >> params->polygonalApproxAccuracyRate;
406     fs["minCornerDistanceRate"] >> params->minCornerDistanceRate;
407     fs["minDistanceToBorder"] >> params->minDistanceToBorder;
408     fs["minMarkerDistanceRate"] >> params->minMarkerDistanceRate;
409     fs["cornerRefinementMethod"] >> params->cornerRefinementMethod;
410     fs["cornerRefinementWinSize"] >> params->cornerRefinementWinSize;
411     fs["cornerRefinementMaxIterations"] >> params->cornerRefinementMaxIterations;
412     fs["cornerRefinementMinAccuracy"] >> params->cornerRefinementMinAccuracy;
413     fs["markerBorderBits"] >> params->markerBorderBits;
414     fs["perspectiveRemovePixelPerCell"] >> params->perspectiveRemovePixelPerCell;
415     fs["perspectiveRemoveIgnoredMarginPerCell"] >> params->perspectiveRemoveIgnoredMarginPerCell;
416     fs["maxErroneousBitsInBorderRate"] >> params->maxErroneousBitsInBorderRate;
417     fs["minOtsuStdDev"] >> params->minOtsuStdDev;
418     fs["errorCorrectionRate"] >> params->errorCorrectionRate;
419     return true;
420 }
```

B.4 mavlink_helper.h

```

1 #include <mavSDK/mavSDK.h>
2 #include <mavSDK/plugins/offboard/offboard.h>
3 #include <mavSDK/plugins/mocap/mocap.h>
4 #include <Eigen/Dense>
5
6 using namespace mavSDK;
7 using std::chrono::milliseconds;
8 using std::this_thread::sleep_for;
9
10 #define CONNECTION_URL "serial:///dev/ttyUSB0:921600"
11 // #define UUID 3690507541151037490 // autopilot cube
12 #define UUID 3762846584429098293 // autopilot cuav
13 #define ERROR_CONSOLE_TEXT "\033[31m" // Turn text on console red
14 #define NORMAL_CONSOLE_TEXT "\033[0m" // Restore normal console colour
15
16 class CommunicationClass{
17     public:
18         void send_msg(Eigen::Vector3d pos, Eigen::Vector3d eul);
19         void init();
20     private:
21         void wait_until_discover(MavSDK& dc);
22         shared_ptr<Mocap> mocap;
23         Mocap::VisionPositionEstimate est_pos;
24         MavSDK dc;
25         ConnectionResult connection_result;
26     };
27
28
29 void CommunicationClass::wait_until_discover(MavSDK& dc)
30 {
31     std::cout << "Waiting to discover system..." << std::endl;
32     std::promise<void> discover_promise;
33     auto discover_future = discover_promise.get_future();
34
35     dc.register_on_discover([&discover_promise](uint64_t uuid) {
36         std::cout << "Discovered system with UUID: " << uuid << std::endl;
37         discover_promise.set_value();
38     });
39
40     discover_future.wait();
41 }
42
43 void CommunicationClass::init()
44 {
45     connection_result = dc.add_any_connection(CONNECTION_URL);
46
47     if (connection_result != ConnectionResult::Success) {
48         std::cout << ERROR_CONSOLE_TEXT << "Connection failed: " << connection_result
49             << NORMAL_CONSOLE_TEXT << std::endl;
50         exit(0);
51     }
52
53     bool connected = dc.is_connected(UUID);
54     while(connected==false){
55         connected = dc.is_connected(UUID);
56         cout << "Waiting system for connection ..." << endl;
57         sleep_for(milliseconds(500));
58     }
59     System& system = dc.system(UUID);
60
61     mocap = std::make_shared<Mocap>(system);
62 }
63
64 void CommunicationClass::send_msg(Eigen::Vector3d pos, Eigen::Vector3d eul)
65 {
66     est_pos.position_body.x_m = pos[0];
67     est_pos.position_body.y_m = pos[1];

```

```
68     est_pos.position_body.z_m = pos[2];
69     est_pos.angle_body.roll_rad = eul[0];
70     est_pos.angle_body.pitch_rad = eul[1];
71     est_pos.angle_body.yaw_rad = eul[2];
72     std::vector<float> covariance{NAN};
73     est_pos.pose_covariance.covariance_matrix=covariance;
74     Mocap::Result result= mocap->set_vision_position_estimate(est_pos);
75     if(result!=Mocap::Result::Success){
76         std::cerr << ERROR_CONSOLE_TEXT << "Set vision position failed: " << result <<
77             NORMAL_CONSOLE_TEXT << std::endl;
78 }
```

Índice de Figuras

2.1	Manejo de medidas retrasadas	4
2.2	Quadrotor en dos dimensiones	8
2.3	EKF no ejecuta la fase de actualización	9
2.4	Fusión del GPS con retraso	9
2.5	Experimento con manejo de medidas retrasadas	10
3.1	Componentes del quadrotor	12
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	14
3.3	Sistemas de referencia presentes en el problema	15
3.4	Superposición de los ejes de referencia del marcador y del cuadrilátero que lo rodea (trazado en verde)	18
3.5	Modo altitude	19
3.6	Modo position	20

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	16

Bibliografía

- [1] *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).
- [2] 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>.
- [3] 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>.

