Trabajo de Fin de Master
Ingeniería Electrónica, Robótica y Automática

Posicionamiento de un UAV usando marcadores visuales

Autor: Isidro Jesús Arias Sánchez

Tutores: Manuel Vargas Villanueva
Manuel Gil Ortega Linares

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 usando marcadores visuales

Autor:

Isidro Jesús Arias Sánchez

Tutores:

Manuel Vargas Villanueva

Profesor Titular

Manuel Gil Ortega Linares

Catedrático

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 usando marcadores visuales

Autor: Isidro Jesús Arias Sánchez
Tutores: Manuel Vargas Villanueva, Manuel Gil Ortega Linares

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 de traducir]

Índice Abreviado

Índice

1 Introducción

El interés
En la robótica uno de los mayores problemas es la percepción del entorno y su ubicación en él. En concreto, cuando se habla de vehículos aéreos no tripulados (UAV), suelen poder posicionarse en el exterior con una precisión de metros. Sin embargo, cuando se trata de navegar en interiores o con precisión más alta el coste de los componentes puede ser muy elevados. Por ejemplo, en exteriores, se puede utilizar la *navegación cinética satelital en tiempo real* (RTK) o en interiores se pueden utilizar balizas acústicas.

Posibles aplicaciones del posicionamiento con precisión centimétrica: - Despegue y aterrizaje - Trayectorias cerca de obstáculo

Esto junto con una mejora de la duración de las baterías o de su cambio automático, podría llevar a - Recogida y depósito de paquetes de manera autónoma - Vuelta de reconocimiento para aplicaciones de seguridad cuando se detecte un posible intruso.

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

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

2 Estimador de estados

En muchas ocasiones se tienen sensores con un retraso y una frecuencia de actualización muy diferentes entre ellos, por ejemplo una IMU es mucho más rápida que el procesamiento de la imagen de una cámara. 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 pero a una frecuencia mayor, existe un estimador llamado *Filtro de Salida*, el cual sí que utiliza la última medida del acelerómetro y del giróscopo.

2.1 Manejo de medidas retrasadas

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 posición estimada por visión es una fuente muy precisa de posición, pero sin embargo tiene un gran retraso desde que se toma la imagen hasta que se procesa y se genera la medida.

Para explicar un método de cómo afrontar este problema, se va a poner un ejemplo de la ejecución paso a paso del estimador con diferentes sensores. Supongamos que en el 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 2.1). 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 medidas de esta celda situada más a la derecha, son las que son usadas por el EKF. Los estados que este genera y las medidas utilizadas para estimarlos se refieren al *horizonte de tiempo retrasado*. Como se muestra en la figura, el buffer tiene una longitud de 7 celdas, por lo tanto las medidas de la IMU que llegan al EKF siempre serán las que se recogieron hace 30 ms.

Pasan algunos ciclos más hasta que en el instante 50ms 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 2.2). En esta también se encuentran las medidas de la IMU tomadas en el instante 25ms, es decir las que fueron tomadas hace 25 ms (50 ms -25 ms), que coincide con el retraso que tiene la posición del GPS con respecto a la IMU. De esta manera se agrupan las medidas que se refieren al mismo instante físico, es decir, el instante en el que llegaron pero **compensándose su retraso**.

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

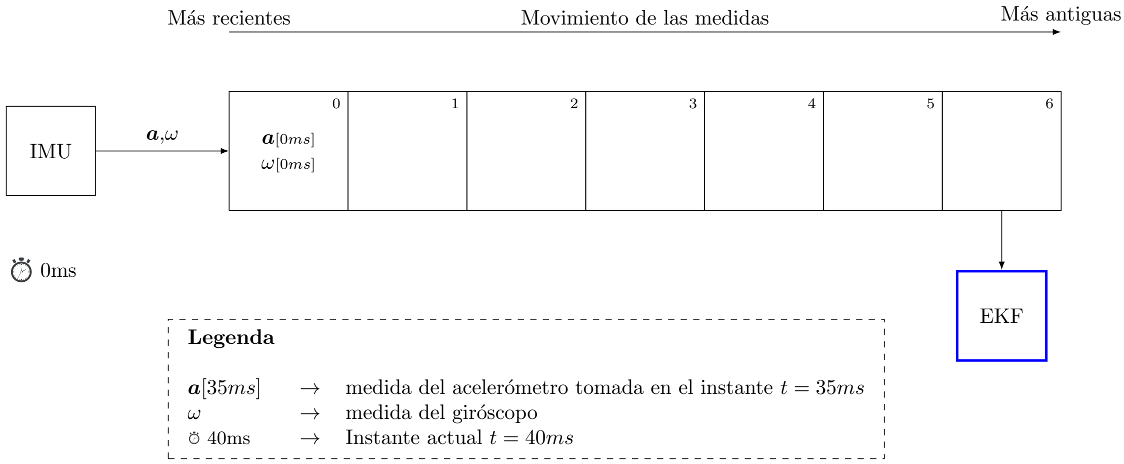

Figura 2.1 Primera medida tomada de la IMU.

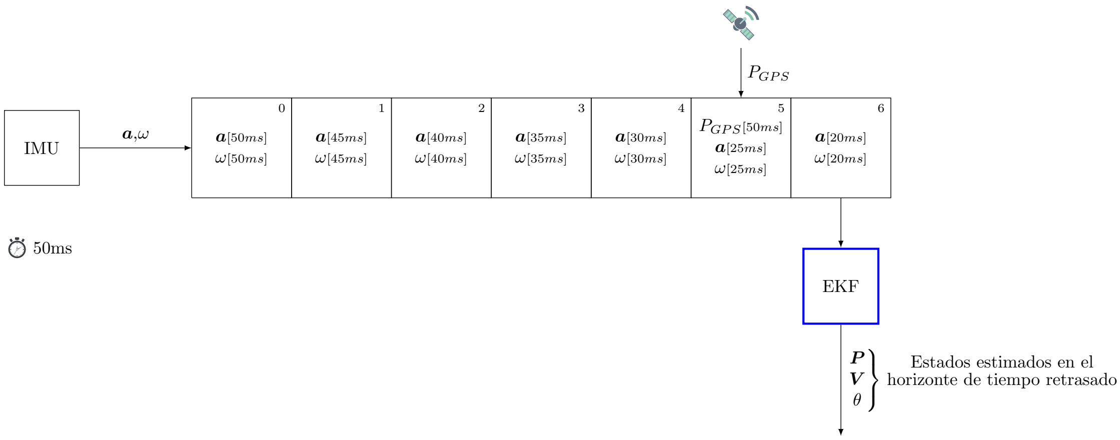

Figura 2.2 Llegada de la medida del GPS.

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.

[añadir detalles de implementación]

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 2.3 como un triángulo) y se le suma a todo el buffer de salida. En el siguiente código se puede ver cómo se ha implentado esto. Se puede ver que la correción 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 aqui se tiene un control proporcional-integral que tiene como señal de control la correción 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.

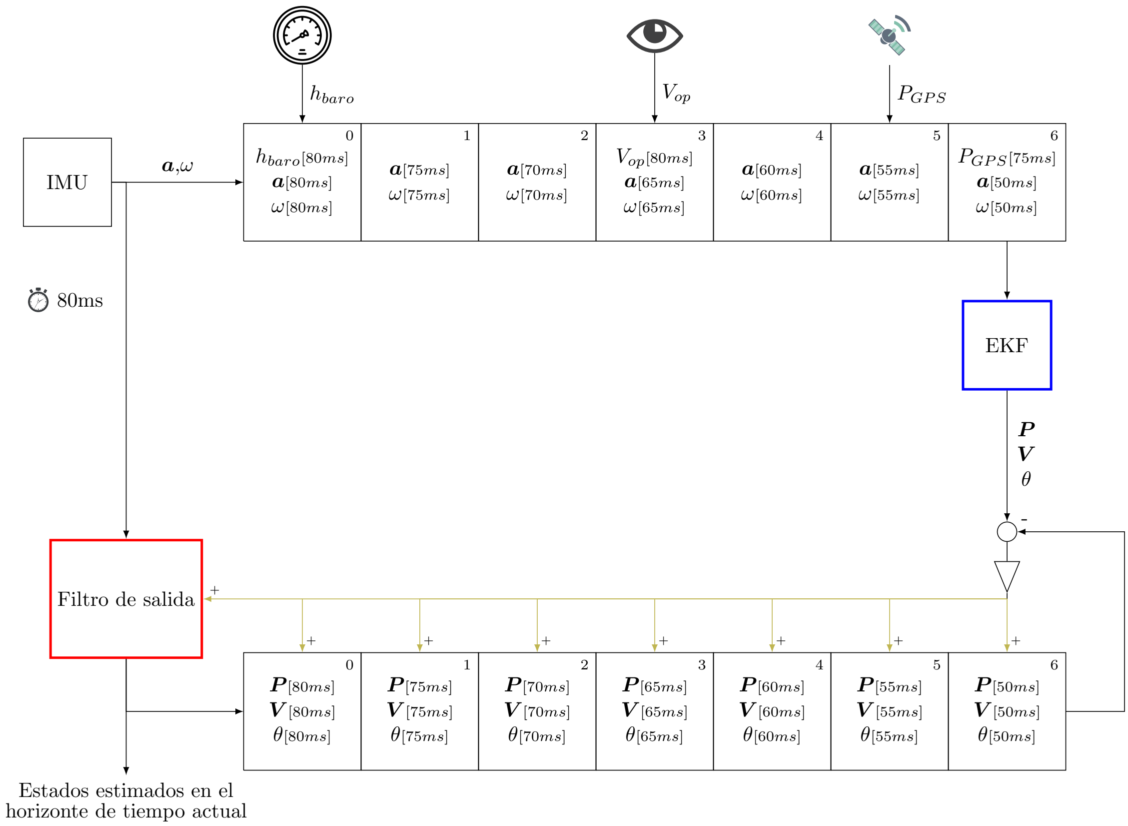

Figura 2.3 Estimador completo.

Código 2.1.1: Correción del buffer de salida. Ubicado en la línea 488 del archivo *Firmware/src/lib/ecl/EKF/ekf.cpp*

```cpp
void Ekf::applyCorrectionToOutputBuffer(float vel_gain, float pos_gain){
        // calculate velocity and position tracking errors
        const Vector3f vel_err(_state.vel - _output_sample_delayed.vel);
        const Vector3f pos_err(_state.pos - _output_sample_delayed.pos);

        _output_tracking_error(1) = vel_err.norm();
        _output_tracking_error(2) = pos_err.norm();

        // calculate a velocity correction that will be applied to the output state history
        _vel_err_integ += vel_err;
        const Vector3f vel_correction = vel_err * vel_gain + _vel_err_integ * sq(vel_gain) * 0.1f;

        // calculate a position correction that will be applied to the output state history
        _pos_err_integ += pos_err;
        const Vector3f pos_correction = pos_err * pos_gain + _pos_err_integ * sq(pos_gain) * 0.1f;

        // loop through the output filter state history and apply the corrections to the velocity and position states
        for (uint8_t index = 0; index < _output_buffer.get_length(); index++) {
                // a constant velocity correction is applied
                _output_buffer[index].vel += vel_correction;

                // a constant position correction is applied
                _output_buffer[index].pos += pos_correction;
        }

```

```
        // update output state to corrected values
        _output_new = _output_buffer.get_newest();
}
```

En el código anterior no ha aparecido la correción 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 correción 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 correción a la orientación estimada en el horizonte de tiempo actual.

Código 2.1.2: Correción de la orientación. Ubicado en el archivo *Firmware/src/lib/ecl/EKF/ekf.cpp*

En la línea 323 se corrige la orientación:

```
        // Apply corrections to the delta angle required to track the quaternion states at the EKF fusion time horizon
        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

```
                // calculate a gain that provides tight tracking of the estimator attitude states and
                // adjust for changes in time delay to maintain consistent damping ratio of ~0.7
                const float time_delay = fmaxf((imu.time_us - _imu_sample_delayed.time_us) * 1e-6f, _dt_imu_avg);
                const float att_gain = 0.5f * _dt_imu_avg / time_delay;

                // calculate a corrrection to the delta angle
                // that will cause the INS to track the EKF quaternions
                _delta_angle_corr = delta_ang_error * att_gain;
```

2.2 EKF para modelo bidimensional

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

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

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

Estados:

$$X = \begin{bmatrix} x \\ y \\ V_x \\ V_y \\ \theta \end{bmatrix} \tag{2.2}$$

Modelo de predicción cinemático:

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

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

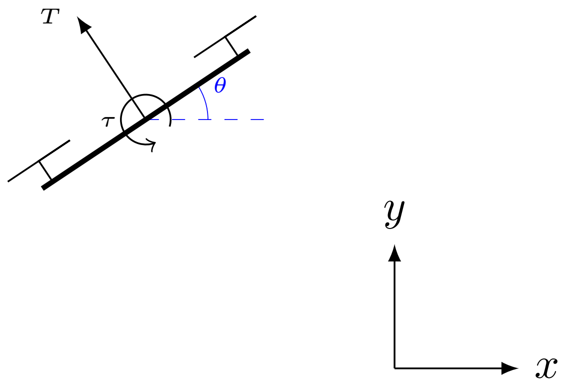

Figura 2.4 Quadrotor en dos dimensiones.

$$\theta_{k+1} = \theta_k + \Delta t \omega \tag{2.5}$$

Jacobiano del modelo de predicción:

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

Jacobiano del acelerómetro y el giróscopo

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

Matriz de covarianzas de la predicción:

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

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 se 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.

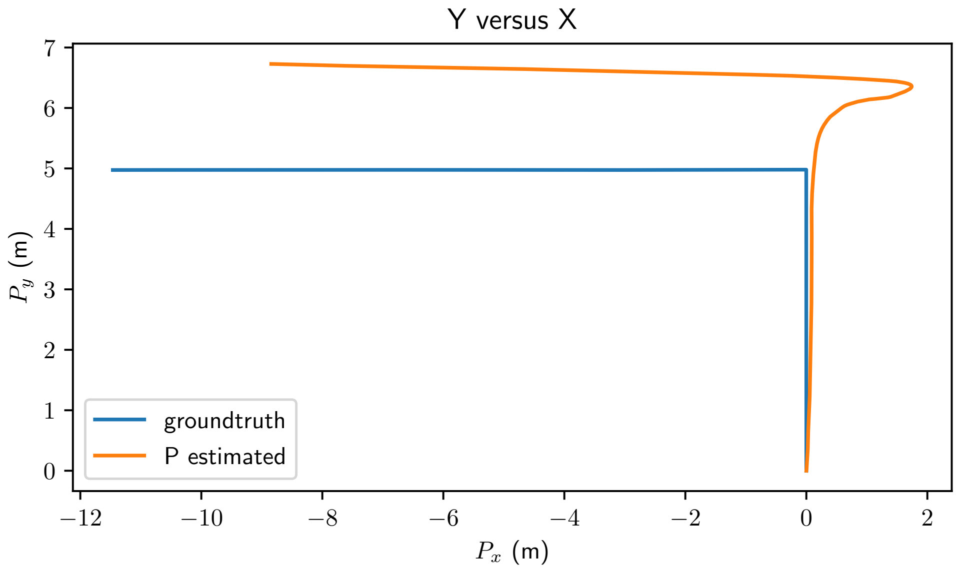

Figura 2.5 EKF no ejecuta la fase de actualización.

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

$$\ddot{\theta} = \frac{\tau}{I} \tag{2.9}$$

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

$$\theta = \theta_{i-1} + \Delta t \dot{\theta} \tag{2.11}$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2.12}$$

$$\boldsymbol{T}_{rot} = R \begin{bmatrix} 0 \\ T \end{bmatrix} \tag{2.13}$$

$$\boldsymbol{F}_g = \begin{bmatrix} 0 \\ -mg \end{bmatrix} \tag{2.14}$$

$$\boldsymbol{a} = \frac{\boldsymbol{T}_r\boldsymbol{ot} + \boldsymbol{F}_g}{m} \tag{2.15}$$

$$\boldsymbol{v} = v_{i-1} + \boldsymbol{a}\Delta t \tag{2.16}$$

$$\boldsymbol{p} = p_{i-1} + \boldsymbol{v}\Delta t \tag{2.17}$$

$$\tag{2.18}$$

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 giróscopo y la del acelerómetro (ver figura 2.5) y en el segundo experimento (2.6) se va a fusionar la medida del GPS. Se puede ver que la fusión del GPS, aunque sea muy ruidosa, mejora en la estimación de la posición ya que no se produce la deriva del primer experimento.

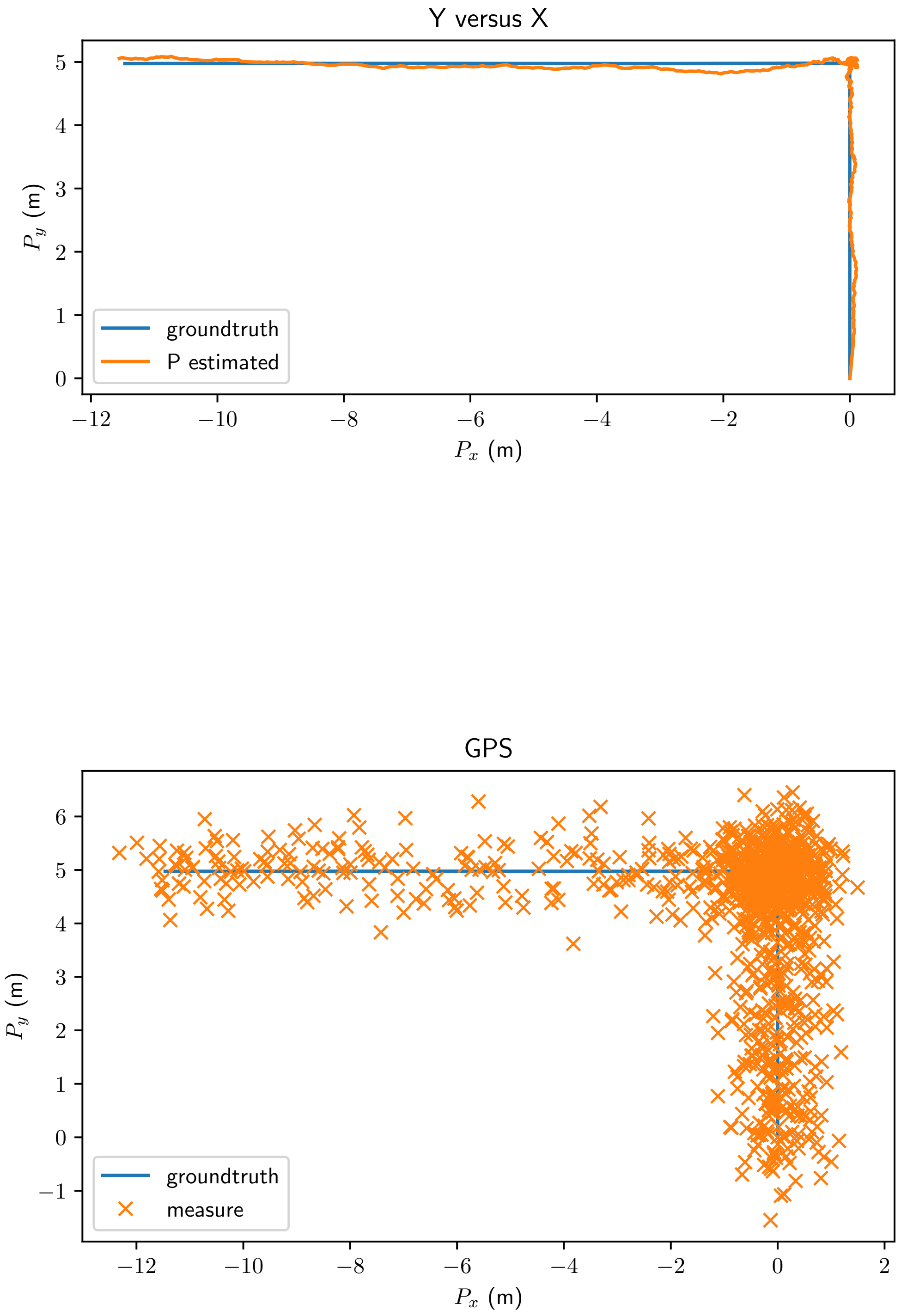

Figura 2.6 Fusionando la medida del GPS.

[Incluir GPS con retraso y comparar]
[Incluir código explicado en anexo]

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 cápitulo se explica que para conseguirlo se ha construido un quadrotor con los componentes necesarios para detectar un marcador visual y cómo se ha programado un ordenador embebido en el quadrotor para que procese dicho marcador.

De manera resumida, el funcionamiento es el siguiente: - Se ha colocado un una cámara en la parte inferior del quadrotor, conectada a un ordenador embebido que procesa las imágenes tomadas - En el suelo se coloca un marcador impreso que es detectado por la cámara - Este le envía al autopiloto la posición y orientación del UAV con respecto a los ejes de coordenadas del del marcador - El autopiloto lo recibe, lo fusiona en su estimador de estados - El estimador 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 (consignas de movimiento en ejes inerciales) 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 aceleración en ejes cuerpo en función de los errores de posición en ejes cuerpo. La desventaja de esto es que se no se utilizan los demás sensores para el posicionamiento, por ejemplo si en un instante falta la medida de la visión, el autopiloto podría tomar otras como el acelerómetro, el GPS, el flujo óptico...

3.1 Mi programa

- Inicialización: - Recoger imagen de la cámara: Este 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 lleva mediante DMA a la RAM. Todo ello sin consumir tiempo de CPU permitiendo que esta haga en paralelo otras operaciones como el procesamiento de imagen. - Detectar marcadores: El objetivo es ubicar los marcadores de la imagen y extraer su identificador. Este proceso está explicado en [1], pero lo que se va a aportar aquí es una referencia directa al código de la librería aruco y a algunas funciones implementadas. - 1. El primer paso es la extración de bordes a partir de la imagen convertida a blanco y negro. - 2. A partir de la imagen binaria del paso anterior, se extraen los contornos usando en *_findMarkerContours()* la función de OpenCV - 3. Se realiza una aproximación poligonal y se quedan aquellos que solo estén compuestos por 4 puntos. - 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. - Inversión de posición y orientación Como se ve en la figura 3.2 hay varios sistemas de referencia que entran en juego y hay que tenerlos presente para tranformar desde lo que aporta la detección de marcadores 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 $R^{Marcador \rightarrow Cámara}$ y $p^{Marcador \rightarrow Cámara}$. En el código 3.1 se puede ver que una vez obtenida dicha matriz de rotación esta se invierte o trasnpone para conseguir la orientación de la cámara vista desde el sistema de referencia del marcador. Dicha matriz se utiliza para expresar la posición del marcador en unos ejes rotados centrados en la cámara y paralelos a los ejes del marcador. Esta posición simplemente se niega para obtener la posición de la cámara vista desde el marcador $p^{Cámara \rightarrow Marcador}$ que es la que se le envía al

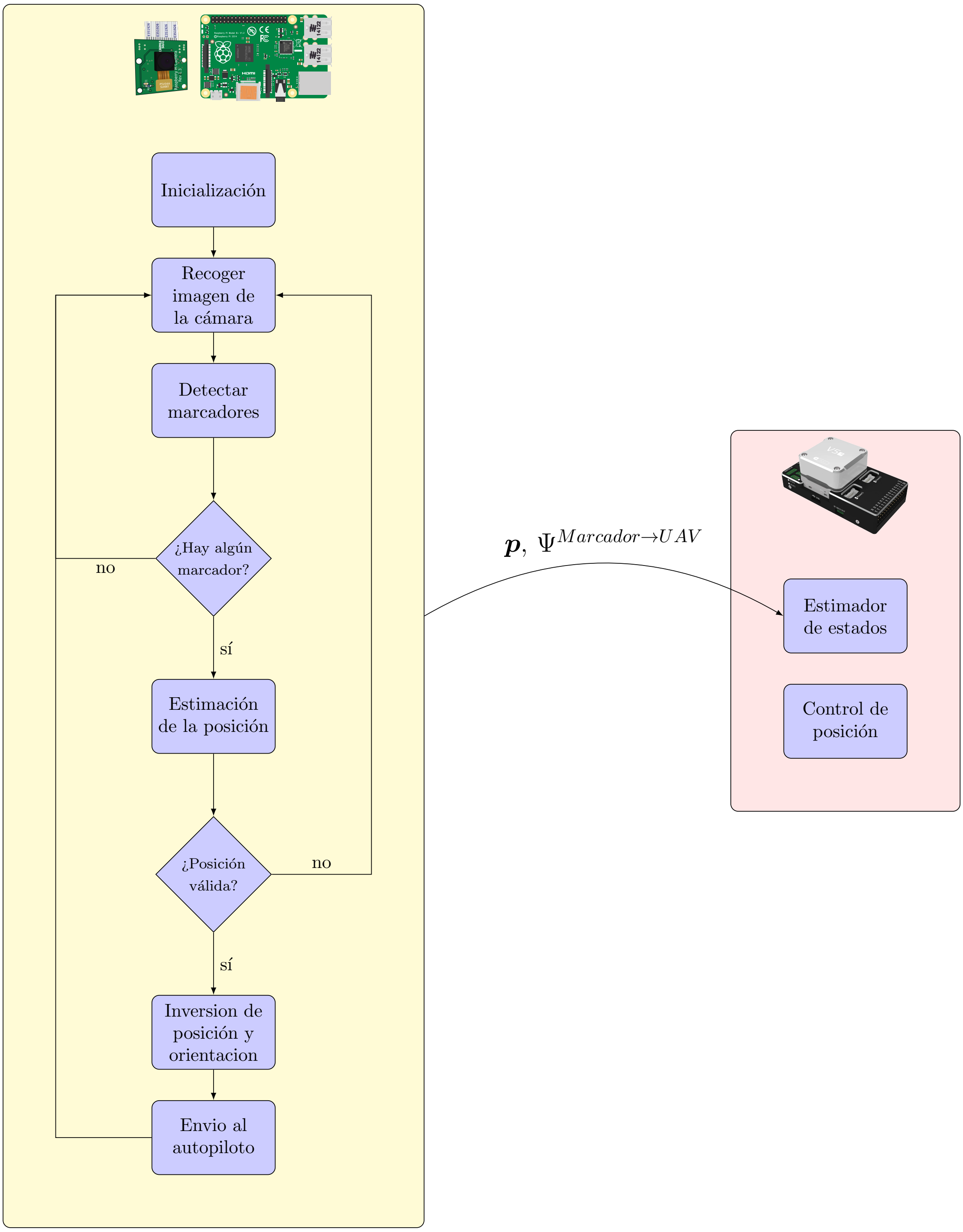

Figura 3.1 A la izquierda: diagrama de flujo del programa que se corre en el ordenador embebido, a la derecha: algunas de las tareas del autopiloto.

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

$$R^{Marcador \to UAV} = \left(R^{Cámara \to Marcador} \; R^{UAV \to Cámara}\right)^T \tag{3.1}$$

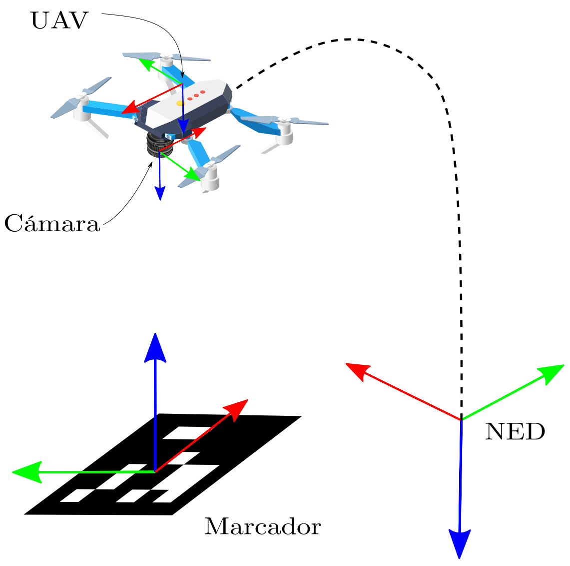

Figura 3.2 Sistemas de referencia presentes en el problema.

Código 3.1.1: Inversión de la posición y orientación ubicada en mi archivo *marker_vison.h*

```
void VisionClass::InvertPose(Eigen::Vector3d &pos, Eigen::Vector3d &eul, Vec3d &rvec, Vec3d &tvec){
    /* @brief Invierte la posición y la rotación. También corrige la posición de la cámara con respecto al UAV
     * @param pos   posición del uav/cámara con respecto al marcador
     * @param eul   orientación del uav con respecto al marcador. El orden de los elementos son 0: roll, 1: pitch, 2: yaw
     * @param rvec  Vector de rotación del marcador con respecto a los ejes de la cámara
     * @param tvec  Vector de translación del marcador con respecto a los ejes de la cámara
     */

    Eigen::Vector3d     pos_marker_in_camera(tvec[0],tvec[1],tvec[2]);

    // Transformación de vector de rotación a matriz de rotación
    cv::Mat                     rot_mat;
    Eigen::Matrix3d             rot_mat_marker_from_camera;
    Rodrigues(rvec,rot_mat);
    cv::cv2eigen(rot_mat, rot_mat_marker_from_camera);

    // La inversa de una matriz de rotación es igual a su traspuesta
    Eigen::Matrix3d     rot_mat_camera_from_marker = rot_mat_marker_from_camera.transpose() ;

    // Se obtiene la posición del marcador en unos ejes paralelos al marcador centrados en la cámara
    Eigen::Vector3d     pos_marker_in_marker_axis = rot_mat_camera_from_marker*pos_marker_in_camera;

    // Si queremos que la posición esté centrada en el marcador y no en la cámara, es necesario negarla
    pos = -pos_marker_in_marker_axis;

    // Aquí debemos de tener en cuenta la rotación de la cámara con respecto al uav. Esta es de 180º alrededor del eje z
    // (de la cámara o del uav, da igual, por tanto da igual postmultiplicar que premultiplicar)
    Eigen::Matrix3d             rot_mat_camera_from_uav;
    rot_mat_camera_from_uav     << -1,    0,    0,
```

```
                                 0,  -1,   0,
                                 0,   0,   1;
    Eigen::Matrix3d             rot_mat_marker_from_uav = rot_mat_marker_from_camera * rot_mat_camera_from_uav;

    // Se obtiene la orientación del uav visto desde el marcador
    Eigen::Matrix3d             rot_mat_uav_from_marker = rot_mat_marker_from_uav.transpose() ;

    // Se obtiene los ángulos de Tait-Bryan en el orden Z-Y-X (ángulos de euler)
    eul = rotationMatrixToEulerAngles(rot_mat_uav_from_marker);
}
```

- Envio al autopiloto La orientación y la posición son enviadas a autopiloto a través del protocolo *Mavlink* Una vez que las recive, 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. Como se puede ver en el código 3.3 esta rotación se le aplica a la posición de la visión. El final la posición que utiliza PX4 para fusionar es la del UAV expresado en unos ejes centrados en el marcador y paralelos a los ejes NED. Que tengan está orientación es importante, que el EKF en su fase de predicción, utilizando el acelerómetro y la orientación estimada, expresa su posición en ejes NED.

Código 3.1.2: Rotación en PX4 de la posición suministrada por la visión

En el archivo *ekf_helper.cpp*:

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

En el archivo *control.cpp*:

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

3.2 Hardware

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

- Raspberry pi 4 Model B. 4 GB de RAM.
- Raspberry Pi Camera Module v2. Campo de visión horizontal de 62 grados, capaz de grabar vídeo con resolución de 1640x1232 a 40fps.
- Cama amortiguadora para el autopiloto.
- Cuav V5+. Autopiloto corriendo PX4. Esquemáticos publicados en Github.
- CUAV NEO V2. Este incluye GNSS, magnetómetro, botón de armado, luces indicadoras y alarma sonora.
- *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.
- Receptor *X8R*. Recibe hasta 16 canales de la emisora, que este caso es una *Taranis Q X7*.
- Módulo de telemetría *Holybro V3*. Permite una comunicación con la estación de control terrestre.
- *DJI 2312E 800KV*. Motor sin escobillas.
- Hélices de fibra de carbono diámetro 9.4 pulgadas y paso de 5 pulgadas. Según el fabricante del motor, con esta hélice se consigue un empuje de 850 gramos alimentado a 14.8 V.
- *Hobbywing XRotor 40A*. Variador de velocidad o ESC. Estos están sobredimensionados ya que fabricante récomienda unos que soporten cómo mínimo una corriente de 20A.

- *Tattu Funfly 1500mAh*. Batería LiPo de 4 celdas.
- Regulador de voltaje para alimentar el autopiloto. Además, lee el voltaje y la corriente que suministra la batería.
- *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.
- *SILABS CP2102*. Puente USB-UART. Se conecta entre el puerto USB del ordenador embebido y el puerto UART del autopiloto.
- *DJI F450*. Chasis de quadrotor de 45 cm de diagonal.
- 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
- *CUAV PX4FLOW 2.1*. Sensor de flujo óptico. También tiene su sofware liberado y el esquemático de una versión anterior.

3.3 Registro de resultados (log)

En este problema hay muchos parámetros que se pueden tocar: - Número de marcadores - Tiempo de exposición de la cámara - Iteraciones máximas de los algoritmos visuales - Mínima calidad permitida en el reconocimento de un marcador

Hay dos maneras de solucionar problemas en ingenieria: - Planificando y haciendo análisis. Esto es lo ideal ya que cada parámetro o configuración está definida a priori tiene su razón y le pueden respaldar las matemáticas - Probando y viendo resultados. A veces el entorno real no es completamente predecible, los modelos no funcionan. O simplemente te has equivocado en los cálculos.

Lo ideal es una fusión de ambos. El estudio teórico, sirve para ver que efecto pueden tener los parámetros, también para darles un valor inicial para hacer las pruebas.

Para verificar el desempeño de la estimación de la posición y orientación, lo ideal sería tener un groundtruth, por ejemplo con GNSS RTK o 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 hayar 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 brúscamente, por lo tanto si al inspeccionar las gráficas temporales de la posición y orientación esto sucede, probablemente se trate de una estimación erronea. Otra forma de verificación es la de ver los ejes del marcador superpuestos en la imagen. Resulta muy fácil de inspeccionar si estos se están moviendo cuando la cámara no cambia de posición.

Funcionalidades implementadas: - Archivo de parámetros - Desactivar la espera de la comunicación del autopiloto - Tomar las imágenes de un vídeo en lugar de la cámara - Guardar la posición y orientación estimadas en un archivo. Representación de estas en una gráfica temporal - Guardar las imágenes de la cámara con los ejes del marcador superpuestos

A pesar de ser un de ejecución rápida, C++ suele ser más dificil para el desarrollador. En cambio Python es un lenguaje que necesita menos líneas de código para hacer lo mismo, tiene una sintaxis más simple y una cantidad enorme de librerias. Por esta razón, en el programa principal escrito en C++ se han hecho llamadas a sripts de python en el arranque y en la finalización del programa, ya que estos son los momentos en los que es menos crítico el tiempo de ejecución. En concreto, en el arranque se genera una carpeta temporal o se elimina su contenido si ya existía antes. Después, durante la ejecución del bucle principal en el que se estiman las medidas, se pueden guardar diversos resultados dicha carpeta, como la estimación de la posición y la orientación en un archivo CSV o las imágenes capturadas con el sistema de coordenadas del marcador superpuesto.

Código 3.3.1: Logging en el archivo *main.cpp*

Script de inicio

```
/* Startup python script for logging */
int res=system("python3 ../python_scripts/startup.py");
if (res!=0){
    cout << "El script de inicio ha fallado con código " << res << endl;
```

```
        exit(1);
    }
```

Script de finalización

```
    res=system("python3 ../python_scripts/shutdown.py");
    if (res!=0){
        cout << "El script de finalización ha fallado con código " << res << endl;
        exit(1);
    }
```

Guardado de la posición y orientación estimada

```
        if (log_file){
            double seconds = getTickCount()/ getTickFrequency() - seconds_init;
            myfile << pos[0] << "," << pos[1] << "," << pos[2] << "," << euler_angles[0] << "," << euler_angles[1] << "," << euler_angles[2] << "," << seconds << "\n";
        }
```

Código 3.3.2: Archivo de perámetros *vision_params.yml*

```
%YAML:1.0
dict_type: 10   # 6x6 256

## Single marker
#marker_length: 0.179
marker_length: 0.2335

## Diamond
diamond: false
# autoscale not implemented yet
autoScale: false

## Charuco
charuco: true
refindStrategy: true
# Charuco Boards
squaresX: 5
squaresY: 7
#squareLength: 0.0365
#markerLength: 0.022
squareLength: 0.0471
markerLength: 0.0282

#squaresX: 4
#squaresY: 6
#squareLength: 0.0601
#markerLength: 0.0362

#squaresX: 3
#squaresY: 4
#squareLength: 0.0465
#markerLength: 0.0765

## Camera
exposure_time: 30
# if fps>40, fov decreases
fps: 40
#video_file: "../videos/vuelo_inclinado.h264"
video_file: "../videos/chess5x7_3.h264"
frame_width: 640
frame_height: 480
#frame_width: 1280
#frame_height: 720
camera_parameters: "../calibration/rpi_v2_camera/cal.yml"
#camera_parameters: "../calibration/hp_camera/cal.yml"

```

```

## General parameters
mav_connect: false
loop_period_ms: 0
open_window: true
wait_key_mill: 1
#TODO: this is mandatory when open_window=true
wait_key: true
show_rejected: false

## Logging
write_images: true
log_file: true
generate_video_from_images: false
```

Apéndice A
Simulador del estimador de estados

Código A.0.1: Simulador del estimador de estados (*main.py*)

```python
#!/bin/env python3
import numpy as np
from numpy.random import randn
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['text.usetex'] = True
import time
from pdb import set_trace


# Parameters
DATA_L=1000
MASS=1 # Kg
G_CONSTANT=9.8 # m/s^2
INERTIA=MASS*0.45**2/12 # Kg.m^2
DT=0.01 # s # Reducir el paso mejora la precisión de la predicción, aunque haya ruido

#ACCEL_NOISE = 0.35 # m/s^2
ACCEL_NOISE = 1 # m/s^2
#GYRO_NOISE = 0.015 # rad/s
GYRO_NOISE = 0.03 # rad/s
#GPS_NOISE= 0.7 # m
GPS_NOISE= 0.1 # m
VISION_NOISE = 0.05 # m

# Plot flags
DRAW_ESTIMATED = True
IMAGE_FOLDER = 'images/'
IMAGE_EXTENSION = 'png'


# Control se realiza sobre los estados reales para acotar más el efecto del estimador
def control_actuators(theta: float, thetad: float ,theta_ref:float, yd_e: float) -> [float,float]:
    # Control gains
    K_height = 2
    K_tilt = 0.2
    Kd_tilt = 0.1
    thrust = MASS*G_CONSTANT/np.cos(theta) + yd_e*K_height
    torque = (theta_ref-theta)*K_tilt -thetad*Kd_tilt
    return thrust, torque

def draw_animation(x,y,theta):
    import numpy as np
    import matplotlib.pyplot as plt
    from matplotlib.animation import FuncAnimation

    fig, ax = plt.subplots()
    xdata, ydata = [], []
```

```
    ln, = plt.plot([], [], 'r')
    ln2, = plt.plot([], [], 'b')


    def init():
        margin=2
        ax.set_xlim(min(x)-margin, max(x)+margin)
        ax.set_ylim(min(y)-margin, max(y)+margin)
        ax.set_aspect('equal')
        return ln,

    def update(frame):
        xdata.append(x[frame])
        ydata.append(y[frame])
        ln.set_data(xdata, ydata)
        c = np.cos(theta[frame])
        s = np.sin(theta[frame])
        rot_mat = np.array([[c, -s], [s, c]])
        p1 = [-0.5,0]
        p2 = [0.5,0]
        p1_rot = rot_mat @ p1
        p2_rot = rot_mat @ p2
        ln2.set_data([p1_rot[0],p2_rot[0]]+x[frame], [p1_rot[1],p2_rot[1]]+y[frame])
        return ln,ln2

    ani = FuncAnimation(fig, update, frames=len(x),
                        init_func=init, blit=True, interval=DT*1e3,repeat=False)
    plt.show()

def main():
    print("-------------------")
    print("Simulador quadrotor")
    print("-------------------")

    # Actuation signals
    thrust = np.ones( DATA_L )*MASS*G_CONSTANT
    torque = np.zeros( DATA_L )

    # translational varibles
    a = np.zeros( (2,DATA_L) )
    v = np.zeros( (2,DATA_L) )
    p = np.zeros( (2,DATA_L) )

    # angular variables. Initialized in zero
    theta = np.zeros( DATA_L )
    thetad = np.zeros( DATA_L )
    thetadd = np.zeros( DATA_L )

    # sensores
    accel = np.zeros( (2,DATA_L) )
    accel_gt = np.zeros( (2,DATA_L) )
    gyro = np.zeros( DATA_L )
    gps = np.zeros( (2,DATA_L) )
    vision = np.zeros( (2,DATA_L) )
    # add optical flow
    op_flow = np.zeros( DATA_L )

    # Setpoints
    yd_ref = np.zeros( DATA_L )
    theta_ref = np.zeros( DATA_L )
    yd_ref[                    :int(DATA_L*0.25)    ] = 2
    theta_ref[int(DATA_L*0.70) :int(DATA_L*0.85)    ] = np.pi/6
    theta_ref[int(DATA_L*0.85) :                    ] = -np.pi/6

    t = np.array(list(range(DATA_L)))*DT


    # Simulate 2 newton law
    # Simular despegue y avance dibujando el suelo
```

```
    for i in range(1,DATA_L): # Pass states are needed, so we start at second
        # Control actuators
        thrust[i], torque[i] =  control_actuators(theta[i-1],thetad[i-1], theta_ref[i], yd_ref[i] - v[1,i-1])

        # Rotational dynamics
        thetadd[i] = torque[i]/INERTIA
        thetad[i] = thetad[i-1] + DT*thetadd[i] # TODO: test trapezoidal integration
        theta[i] = theta[i-1] + DT*thetad[i]

        # Rotation matrix. Transform body coodinates to inertial coordinates
        c = np.cos(theta[i])
        s = np.sin(theta[i])
        rot_mat = np.array([[c, -s], [s, c]])

        # Translational dynamics
        thrust_rot= rot_mat @ np.array([0, thrust[i]])
        gravity_force = np.array([0, -G_CONSTANT])*MASS
        a[:,i] = (thrust_rot+gravity_force)/MASS
        v[:,i] = v[:,i-1] + DT*a[:,i]
        p[:,i] = p[:,i-1] + DT*v[:,i]

        # simulate sensors
        accel_gt[:,i] = np.linalg.inv(rot_mat) @ a[:,i]
        accel[:,i] = accel_gt[:,i] + randn(2)*ACCEL_NOISE # TODO: Habría que multiplicarlo por la inversa de rot_mat?
        gps[:,i] = p[:,i] + randn(2)*GPS_NOISE
        vision[:,i] = p[:,i] + randn(2)*VISION_NOISE
        gyro[i] = thetad[i] + randn(1)*GYRO_NOISE


    # States estimation
    v_est = np.zeros( (2,DATA_L) )
    p_est = np.zeros( (2,DATA_L) )
    theta_est = np.zeros( DATA_L )

    # Matriz de covarianzas
    P_est = np.zeros( (5,5,DATA_L) )

    # Error en la predicción # TODO: calcularlo a partir de los ruidos de los sensores
    Q = np.zeros( (5,5) )

    # Jacobianos de los modelos de observación
    H_vision = np.zeros((2,5))
    H_vision[0,0]=1
    H_vision[1,1]=1
    R_vision = np.diag([VISION_NOISE**2, VISION_NOISE**2])

    H_gps = np.zeros((2,5))
    H_gps[0,0]=1
    H_gps[1,1]=1
    R_gps = np.diag([GPS_NOISE**2, GPS_NOISE**2])

    # Initalization
    for i in range(1,DATA_L):
        # Prediction de los estados
        theta_pred = theta_est[i-1] + DT*gyro[i]
        c = np.cos(theta_pred)
        s = np.sin(theta_pred)
        # TODO: utilizar aquí el predicho ahora o el estimado anterior?
        #c = np.cos(theta_est[i-1])
        #s = np.sin(theta_est[i-1])
        rot_mat = np.array([[c, -s], [s, c]])
        v_pred = v_est[:,i-1] + DT*rot_mat @ accel[:,i]
        p_pred = p_est[:,i-1] + DT*v_est[:,i-1]

        # Predicción de la matriz de covarianzas
        x_pred=np.array([p_pred[0], p_pred[1], v_pred[0], v_pred[1], theta_pred])
        F = np.array([
```

```
                       [1,0,DT,0 ,0],
                       [0,1,0 ,DT,0],
                       [0,0,1 ,0,DT*(-accel[0,i]*np.sin(theta_pred) + accel[1,i]*np.cos(theta_pred) )],
                       [0,0,0 ,1,DT*(-accel[0,i]*np.cos(theta_pred) - accel[1,i]*np.sin(theta_pred) )],
                       [0,0,0 ,0 ,1],
            ])
        G = np.array([
                [0      ,0      ,0],
                [0      ,0      ,0], # que pasaría si desarrollo v(a) aquí?
                [DT*c   ,DT*s   ,0],
                [-DT*s  , DT*c  ,0],
                [0      ,0      ,DT],

            ])
        Q = G @ np.diag([ACCEL_NOISE**2, ACCEL_NOISE**2, GYRO_NOISE**2]) @ np.transpose(G)
        P_pred = np.zeros((5,5))
        P_pred = F @ P_est[:,:,i-1] @ np.transpose(F) + Q

        x_est = x_pred
        p_est[:,i] = x_est[0:2] # Remind slices x:y doesn't include y
        v_est[:,i] = x_est[2:4]
        theta_est[i] = x_est[4]
        P_est[:,:,i] = P_pred

        ### Update
        ## vision
        #innov = vision[:,i] - p_pred[:,i]
        #S_vision = H_vision @ P_pred @ np.transpose(H_vision) + R_vision
        #K_f = P_pred @ np.transpose(H_vision) @ np.linalg.inv(S_vision)
        #x_est = x_est + K_f @ innov
        #p_est[:,i] = x_est[0:2] # Remind slices x:y doesn't include y
        #v_est[:,i] = x_est[2:4]
        #theta_est[i] = x_est[4]
        #p_est[:,i] = x_est[1:2]
        #P[:,:,i] = P_pred + K_f @ H_vision @ P_pred

        ## gps
        innov = gps[:,i] - p_est[:,i]
        S_gps = H_gps @ P_est[:,:,i] @ np.transpose(H_gps) + R_gps
        K_f = P_est[:,:,i] @ np.transpose(H_gps) @ np.linalg.inv(S_gps)
        x_est = x_est + K_f @ innov
        p_est[:,i] = x_est[0:2] # Remind slices x:y doesn't include y
        v_est[:,i] = x_est[2:4]
        theta_est[i] = x_est[4]
        P_est[:,:,i] = P_est[:,:,i] - K_f @ H_gps @ P_est[:,:,i]


    # Plot results
    fig, ax = plt.subplots()
    ax.set_title('X position versus time')
    ax.plot(t, p[0,:], label='P groundtruth')
    if DRAW_ESTIMATED:
        ax.plot(t, p_est[0,:], label='P estimated')
        ax.plot(t, P_est[0,0,:],label='error estimated')
    plt.xlabel('t (s)')
    plt.ylabel('$P_x$ (m)')
    ax.legend()
    plt.savefig(IMAGE_FOLDER + 'x_t.' + IMAGE_EXTENSION)

    fig, ax = plt.subplots()
    ax.set_title('Y position versus time')
    ax.plot(t, p[1,:], label='P groundtruth')
    if DRAW_ESTIMATED:
        ax.plot(t, p_est[1,:], label='P estimated')
        ax.plot(t, P_est[1,1,:],label='error estimated')
    plt.xlabel('t (s)')
```

```
    plt.ylabel('$P_y$ (m)')
    ax.legend()
    plt.savefig(IMAGE_FOLDER + 'y_t.' + IMAGE_EXTENSION)

    fig, ax = plt.subplots()
    ax.set_title('Velocity versus time')
    ax.plot(t, v[0,:],color='tab:red', label='$V_x$ groundtruth', linestyle='--')
    ax.plot(t, v[1,:],color='tab:blue', label='$V_y$ groundtruth', linestyle='--')
    if DRAW_ESTIMATED:
        ax.plot(t, v_est[0,:],color='tab:red', label='$V_x$ estimated')
        ax.plot(t, v_est[1,:],color='tab:blue', label='$V_y$ estimated')
        ax.plot(t, P_est[2,2,:],label='error estimated $V_x$')
        ax.plot(t, P_est[3,3,:],label='error estimated $V_y$')
    plt.xlabel('t (s)')
    plt.ylabel('$V$ (m/s)')
    ax.legend()
    plt.savefig(IMAGE_FOLDER + 'V.' + IMAGE_EXTENSION)

    fig, ax = plt.subplots()
    ax.set_title('Tilt versus time')
    ax.plot(t, theta, label='groundtruth')
    if DRAW_ESTIMATED:
        ax.plot(t, theta_est, label='estimated')
        ax.plot(t, P_est[4,4,:],label='error estimated')
    plt.xlabel('t (s)')
    plt.ylabel(r'$\theta$ (rad)')
    ax.legend()
    plt.savefig(IMAGE_FOLDER + 'theta.' + IMAGE_EXTENSION)

    fig, ax = plt.subplots()
    ax.set_title('Y versus X')
    ax.plot(p[0,:],p[1,:], label='groundtruth')
    if DRAW_ESTIMATED:
        ax.plot(p_est[0,:],p_est[1,:], label='P estimated')
    plt.xlabel('$P_x$ (m)')
    plt.ylabel('$P_y$ (m)')
    ax.legend()
    ax.set_aspect('equal')
    plt.savefig(IMAGE_FOLDER + 'tray.' + IMAGE_EXTENSION)

    # Sensors
    fig, ax = plt.subplots()
    ax.set_title('Acelerometer')
    ax.plot(t, accel_gt[0,:],   color='tab:red',    label='$a_x$ groundtruth',
    ↪ linestyle='--')
    ax.plot(t, accel_gt[1,:],   color='tab:blue',   label='$a_y$ groundtruth',
    ↪ linestyle='--')
    ax.plot(t, accel[0,:],      color='tab:red',    label='$a_x$ measure')
    ax.plot(t, accel[1,:],      color='tab:blue',   label='$a_y$ measure')
    plt.xlabel('t (s)')
    plt.ylabel('a (m/s)')
    ax.legend()
    plt.savefig(IMAGE_FOLDER + 'accel.' + IMAGE_EXTENSION)

    fig, ax = plt.subplots()
    ax.set_title(r'Gyro ($\omega$)')
    ax.plot(t, thetad,label='groundtruth')
    ax.plot(t, gyro,  label='measure')
    plt.xlabel('t (s)')
    plt.ylabel(r'$\omega$ (rad/s)')
    ax.legend()
    plt.savefig(IMAGE_FOLDER + 'gyro.' + IMAGE_EXTENSION)

    fig, ax = plt.subplots()
    ax.set_title('GPS')
    ax.plot(p[0,:],p[1,:],label='groundtruth')
    ax.plot(gps[0,:],gps[1,:],  label='measure', linestyle=" ", marker="x")
    plt.xlabel('$P_x$ (m)')
    plt.ylabel('$P_y$ (m)')
```

```python
    ax.legend()
    ax.set_aspect('equal')
    plt.savefig(IMAGE_FOLDER + 'gps.' + IMAGE_EXTENSION)

    fig, ax = plt.subplots()
    ax.set_title('Elementos diagonales de la matriz de covarianzas')
    ax.plot(t,P_est[0,0,:],label='$P_x$')
    ax.plot(t,P_est[1,1,:],label='$P_y$')
    ax.plot(t,P_est[2,2,:],label='$V_x$')
    ax.plot(t,P_est[3,3,:],label='$V_y$')
    ax.plot(t,P_est[4,4,:],label=r'$\theta$')
    plt.xlabel('$t$ (s)')
    plt.ylabel('m,m,m/s,m/s,rad')
    ax.legend()
    plt.savefig(IMAGE_FOLDER + 'P_est_diag.' + IMAGE_EXTENSION)

    fig, ax = plt.subplots()
    ax.set_title('Matriz de covarianzas')
    # Parece que es diagonal
    ax.plot(t,P_est[0,1,:],label='$P_{est}$[0,1]')
    ax.plot(t,P_est[0,2,:],label='$P_{est}$[0,2]')
    ax.plot(t,P_est[0,3,:],label='$P_{est}$[0,3]')
    ax.plot(t,P_est[0,4,:],label='$P_{est}$[0,4]')
    ax.plot(t,P_est[1,2,:],label='$P_{est}$[1,2]')
    ax.plot(t,P_est[1,3,:],label='$P_{est}$[1,3]')
    ax.plot(t,P_est[1,4,:],label='$P_{est}$[1,4]')
    ax.plot(t,P_est[2,3,:],label='$P_{est}$[2,3]')
    ax.plot(t,P_est[2,4,:],label='$P_{est}$[2,4]')
    ax.plot(t,P_est[3,4,:],label='$P_{est}$[3,4]')
    ax.plot(t,P_est[0,0,:],label='$P_x$', linestyle="--")
    ax.plot(t,P_est[1,1,:],label='$P_y$', linestyle="--")
    ax.plot(t,P_est[2,2,:],label='$V_x$', linestyle="--")
    ax.plot(t,P_est[3,3,:],label='$V_y$', linestyle="--")
    plt.xlabel('$t$ (s)')
    ax.legend()
    plt.savefig(IMAGE_FOLDER + 'P_est' + IMAGE_EXTENSION)


    plt.show()

    draw_animation(p[0,:],p[1,:],theta)

if __name__=="__main__":
    main()
```

Índice de Figuras

Índice de Tablas

Bibliografía

[1] 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*.