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:

Agradecimientos

Estos 4 años han sido mejores gracias al apoyo de algunas personas. Especialmente quiero agradecérselo a mis padres, Mercedes y José, a mi tía Antonia, a mi abuela Mercedes, a mi hermano Héctor, a Candi, mi novia, a mis amigos de Sevilla, Ángel, Carlos, David, Fernando, Jorge y Samuel, a los de Cádiz, Daniel, Juan Luís, Javier Mena, Javier Sanz, Pepe y Pedro, y por último y no menos importante, a mi tutores Manuel Vargas y Manuel Gil, y al resto de los profesores que me han sabido enseñar y de los que tanto he aprendido. Gracias a todos.

Isidro Arias Sánchez
Sevilla, 2019

Resumen

En este proyecto se va a desarrollar el estudio de un vehículo aéreo no tripulado que tiene dos hélices o rotores orientables, denominado tiltrotor, del ingles *tilt* que significa inclinar.

El fundamento de este trabajo es contribuir en el estudio de una nave convertible en la que los rotores, al ser inclinables, funcionan con las ventajas de dos modelos de aeronaves diferentes, una del tipo helicóptero que lo dota de alta maniobrabilidad, y otra de tipo aeroplano que le permite recorrer largas distancias.

Entendiendo la importancia de la simulación en la construcción de vehículos aéreos, se opta por la elección de dos herramientas diferentes en el ámbito de la simulación. Se trata de comprobar y testar las coincidencias entre ellas, de manera que se corrija y disminuya la posibilidad de errores en el proceso.

Por último, se añade al proyecto la fabricación de un tiltrotor para verificar los modelos utilizados en la simulación.

Abstract

The basis of this work is to contribute to the study of a convertible aerial vehicle in which the rotors, being tiltable, operate with the advantages of two different aircraft models, one of the helicopter type that gives it high maneuverability, and one of the airplane type that allows traveling long distances.

Understanding the importance of simulation in the design of aerial vehicles, we have chosen two different tools in the field of simulation. It is about verifying and testing the coincidences between them, so that it corrects and reduces the possibility of errors in the process.

Finally, a prototype will be built in order to verify the models used in the simulation.

Índice Abreviado

Resumen III
Abstract V
Índice Abreviado VII

1 Estimador de estados **1**
1.1 Manejo de medidas retrasadas 1
1.2 EKF para modelo bidimensional 2
1.3 Simulación del quadrotor y del estimador 4

Apéndice A Simulador del estimador de estados **7**

Índice de Figuras 13
Índice de Tablas 15

Índice

Resumen III
Abstract V
Índice Abreviado VII

1 Estimador de estados **1**
1.1 Manejo de medidas retrasadas 1
1.2 EKF para modelo bidimensional 2
1.3 Simulación del quadrotor y del estimador 4

Apéndice A Simulador del estimador de estados **7**

Índice de Figuras 13
Índice de Tablas 15

1 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.

1.1 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 1.1). 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 1.2). 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

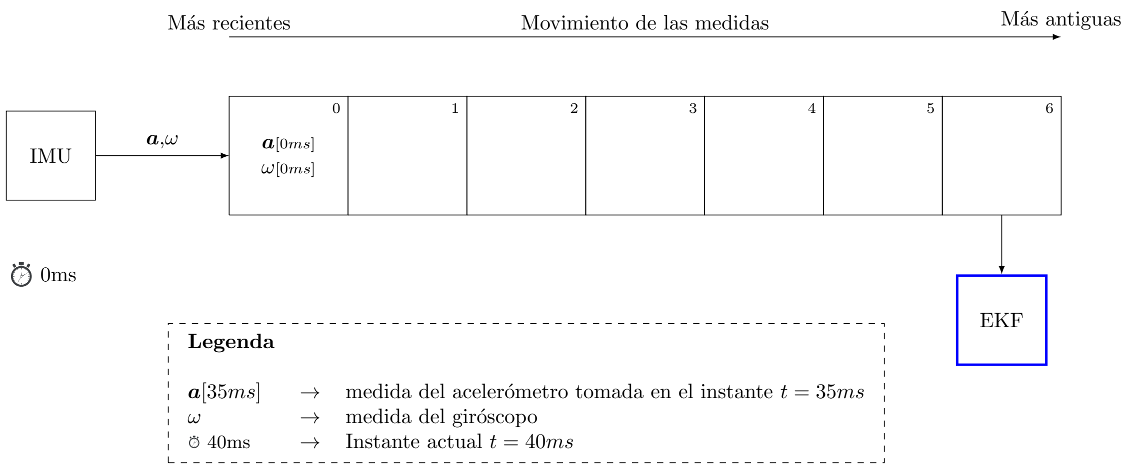

Figura 1.1 Primera medida tomada de la IMU.

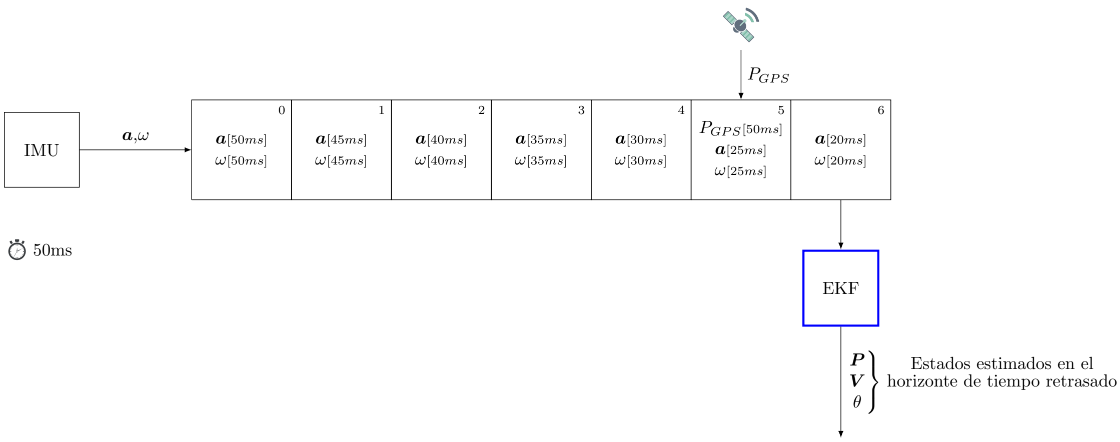

Figura 1.2 Llegada de la medida del GPS.

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 oros 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]

1.2 EKF para modelo bidimensional

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

$$X_{k+1} = f(X_k) \tag{1.1}$$

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

Estados:

$$X = \begin{bmatrix} x \\ y \\ V_x \\ V_y \\ \theta \end{bmatrix} \tag{1.2}$$

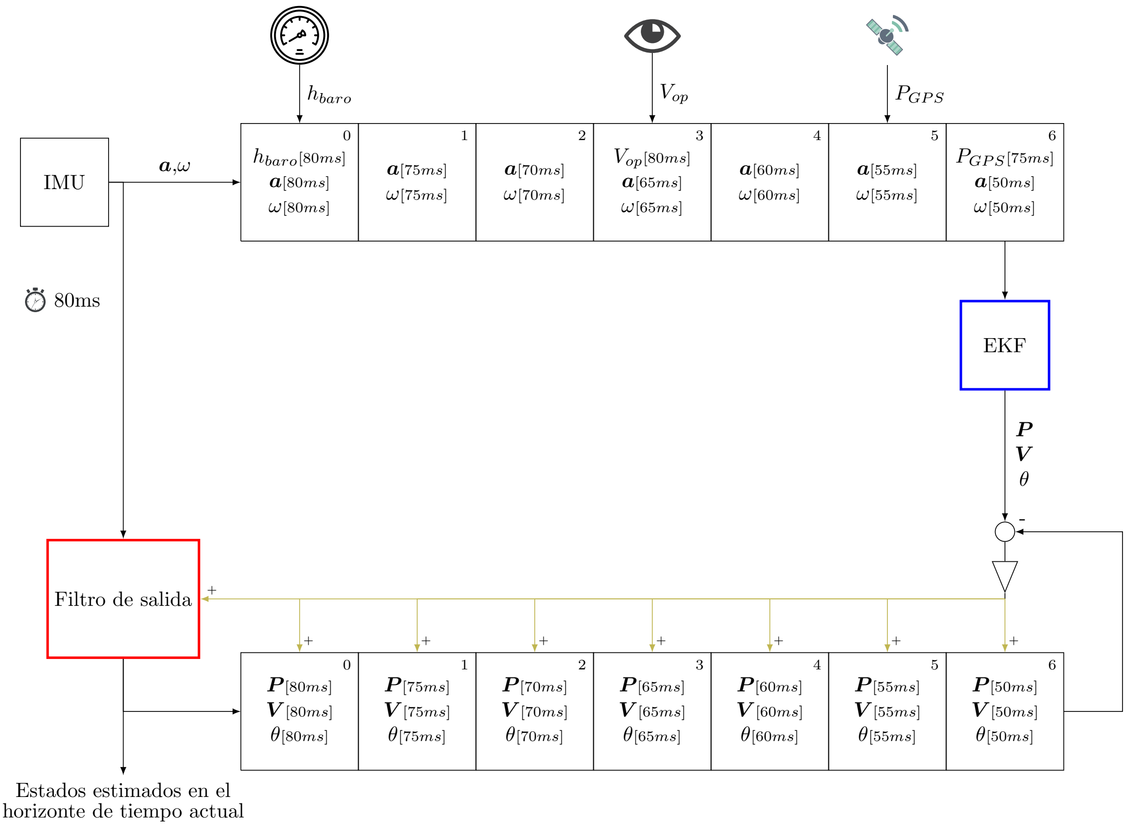

Figura 1.3 Estimador completo.

Modelo de predicción cinemático:

$$\begin{bmatrix} x \\ y \end{bmatrix}_{k+1} = \begin{bmatrix} x \\ y \end{bmatrix}_{k} + \begin{bmatrix} V_x \\ V_y \end{bmatrix}_{k} \Delta t \tag{1.3}$$

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix}_{k+1} = \begin{bmatrix} V_x \\ V_y \end{bmatrix}_{k} + \Delta t \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \boldsymbol{a} + \begin{bmatrix} 0 \\ -m\,g \end{bmatrix} \Delta t \tag{1.4}$$

$$\theta_{k+1} = \theta_k + \Delta t\,\omega \tag{1.5}$$

Jacobiano del modelo de predicción:

$$F = \left.\frac{\partial f}{\partial X}\right|_{X_{k-1}} = \begin{bmatrix} 1 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & \Delta t\left(-a_x \sin\theta_{k-1} + a_y \cos\theta_{k-1}\right) \\ 0 & 0 & 0 & 1 & \Delta t\left(-a_x \cos\theta_{k-1} - a_y \sin\theta_{k-1}\right) \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.6}$$

Jacobiano del acelerómetro y el giróscopo

$$G = \frac{\partial f}{\partial \boldsymbol{a},\omega} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \Delta t\cos\theta & \Delta t\sin\theta & 0 \\ -\Delta t\sin\theta & \Delta t\cos\theta & 0 \\ 0 & 0 & \Delta t \end{bmatrix} \tag{1.7}$$

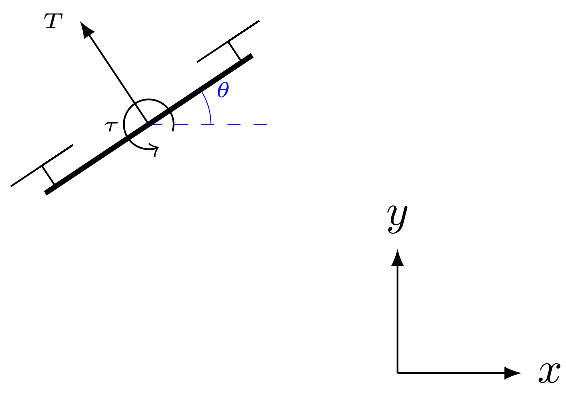

Figura 1.4 Quadrotor en dos dimensiones.

Matriz de covarianzas de la predicción:

$$Q = G \begin{bmatrix} \sigma_a^2 & 0 & 0 \\ 0 & \sigma_a^2 & 0 \\ 0 & 0 & \sigma_\omega^2 \end{bmatrix} G^T \tag{1.8}$$

1.3 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.

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

$$\ddot{\theta} = \frac{\tau}{I} \tag{1.9}$$

$$\dot{\theta} = \dot{\theta}_{i-1} + \Delta t \ddot{\theta} \tag{1.10}$$

$$\theta = \theta_{i-1} + \Delta t \dot{\theta} \tag{1.11}$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{1.12}$$

$$\boldsymbol{T}_{rot} = R \begin{bmatrix} 0 \\ T \end{bmatrix} \tag{1.13}$$

$$\boldsymbol{F_g} = \begin{bmatrix} 0 \\ -mg \end{bmatrix} \tag{1.14}$$

$$\boldsymbol{a} = \frac{\boldsymbol{T} + \boldsymbol{F_g}}{m} \tag{1.15}$$

$$\boldsymbol{v} = v_{i-1} + \boldsymbol{a}\Delta t \tag{1.16}$$

$$\boldsymbol{p} = p_{i-1} + \boldsymbol{v}\Delta t \tag{1.17}$$

$$\tag{1.18}$$

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 1.5) y en el segundo experimento (1.6) se va

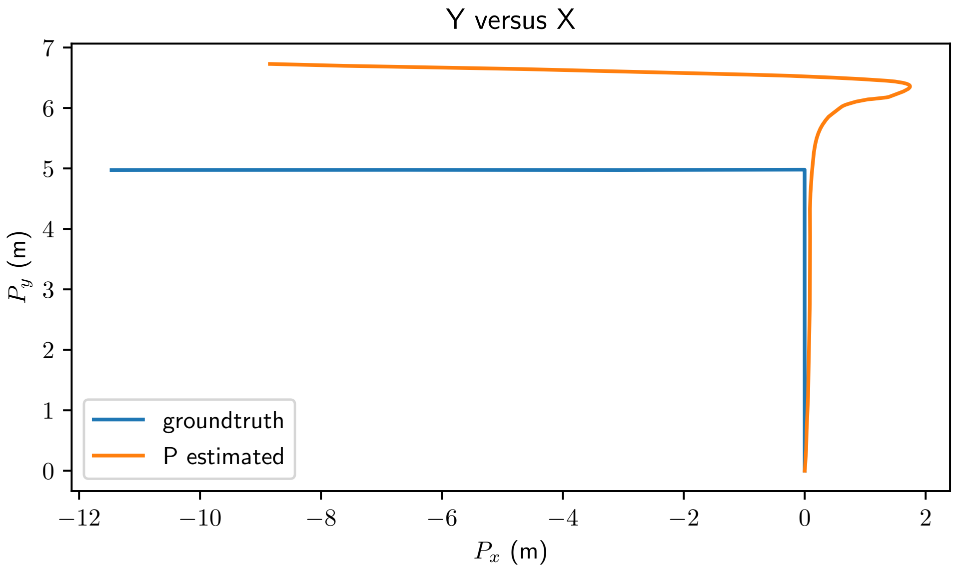

Figura 1.5 EKF no ejecuta la fase de actualización.

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.

[Incluir GPS con retraso y comparar]

[Incluir código explicado en anexo]

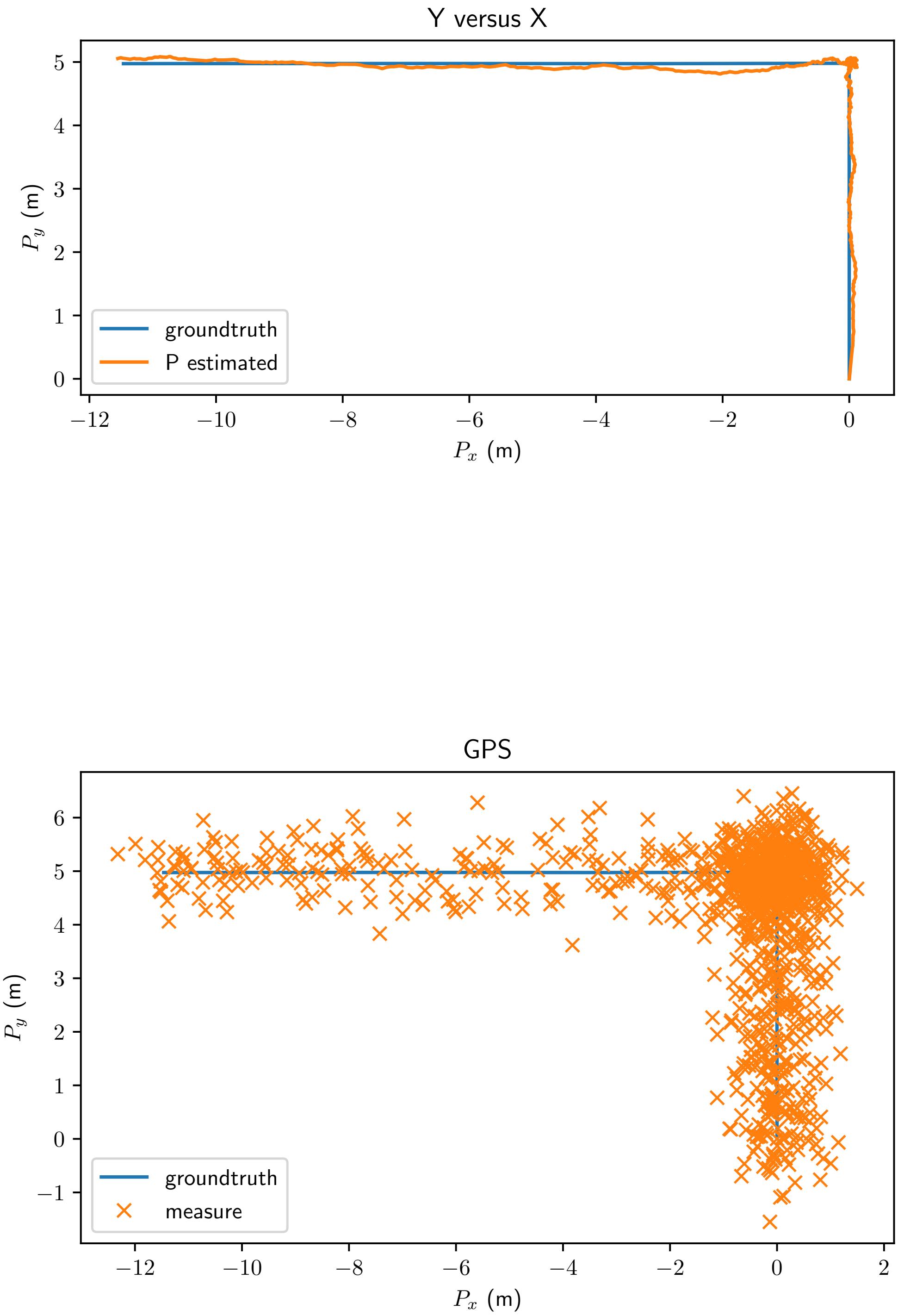

Figura 1.6 Fusionando la medida del GPS.

Apéndice A
Simulador del estimador de estados

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

```
#!/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)')
```

```
    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

1.1	Primera medida tomada de la IMU	2
1.2	Llegada de la medida del GPS	2
1.3	Estimador completo	3
1.4	Quadrotor en dos dimensiones	4
1.5	EKF no ejecuta la fase de actualización	5
1.6	Fusionando la medida del GPS	6

Índice de Tablas