Development of a Control Algorithm with Kalman Filter enhancement to control the altitude of a Quadcopter.

A red and white logo

Description automatically generated

Ben Russell

40178580

13/04/2024

# Abstract

# **Introduction**

In this lab, we embark on the task of designing an estimation and control system for the "Bzzz" quadcopter, as depicted in **Figure 1**. The objective is to develop a robust system that enables the quadcopter to maintain a constant altitude, regardless of external factors and disturbances. To achieve this, we leverage a combination of sensors, including a time-of-flight (ToF) sensor, a barometer, and a global navigation satellite system (GNSS) sensor.

A drone with instructions on it

Description automatically generated with medium confidence

Figure 1 Picture illustrating the Bzzz quadcopter.

The ToF sensor provides measurements of the distance from the ground with a standard error of ±6 cm, while the barometer estimates altitude with an error of ±25 cm. Additionally, the GNSS sensor, coupled with a base GNSS station, offers precise altitude estimates with a standard error of ±5 cm. These sensors serve as crucial components in our estimation framework, allowing us to accurately infer the state of the quadcopter.

To model the dynamics of the quadcopter, we consider the forces acting upon it, namely the weight \(mg\) and the force exerted by the propellers \(F\_{\text{prop}}\). Through experimentation, we establish a model for the lifting force *F*prop​= *ατ*+*β*, where *α >* 0 and *β* < 0 are constants dependent on the battery's charge level. Utilizing this model, we derive the quadcopter's dynamical equations, enabling us to predict its vertical acceleration and velocity.

With our system model in place, we turn our attention to estimation. Employing the Kalman filter, we aim to estimate the unknown parameters *α*and *β* while simultaneously inferring the quadcopter's state variables, including altitude and velocity. Leveraging sensor measurements from the GNSS, ToF, and barometer sensors, we construct an estimation system capable of accurately tracking the quadcopter's state despite uncertainties and disturbances.

Furthermore, we design a control system that utilizes the altitude estimates obtained from our Kalman filter to enable the quadcopter to maintain a constant altitude. Our controller adjusts the throttle reference signal sent to the quadcopter's motors, ensuring precise altitude control in various scenarios, including sudden changes in altitude, GNSS signal interruptions, battery drainage, and sensor biases.

In this report, we present our proposed estimation and control system, along with experimental results demonstrating its effectiveness in diverse real-world scenarios. Additionally, we discuss the rationale behind our parameter choices and provide justifications for our design decisions. Through rigorous testing and analysis, we aim to validate the performance and reliability of our estimation and control system for altitude hold in quadcopters.

# **Methodology**

The first step was to design a Kalman filter that was able to effectively predict the system behaviour over time. Kalman filters use the systems dynamic model, known control inputs to the system (Throttle speed), and multiple sequential measurements (sensors) to form an estimate of the system’s varying quantities.

The Kalman filter is implemented as a class on Python. Within the class, we initialise all the state variables. These matrices A, G, C, Q, P0, Q and R define the system matrices and initial conditions. A represents the system dynamics from one state to another and is selected to have a simple linear relationship between the velocity and position states. G is the control input matrix which maps the control input to the change in state of the system. Here, it accounts for the effect of the control input (Throttle adjustment) on the position and velocity of the quadcopter. G is chosen to capture the quadratic relationship between control input and position. C is the output matrix which specifies how the state variables are mapped to the measurements obtained from the sensors. In this case, it indicates that only the position component of the state is measured directly. P0 is the initial state covariance matric which represents the uncertainty in the initial state of the system. It is a measure of the confidence in the initial estimate of the quadcopters position and velocity. In this work, we select P0 such that it reflects higher uncertainty in the position and lower uncertainty in the velocity.

The process noise covariance Q represents the variance of the random disturbances affecting the system dynamics. It accounts for uncertainty and disturbances in the system model not captured by the state transition matric A and control input matric G. A higher value of Q indicates higher process noise. R is the measurement noise covariance and represents the covariance of noise in the sensor measurements. It captures the uncertainty and errors associated with sensor readings. A lower value of R indicates high confidence in the sensor measurements.

The initial state of the system is assumed to be 0, meaning the initial velocity and altitude is 0 (Starts from rest).

The Kalman filter has three primary functions that encompass the entire algorithm. These are 1) State Update , 2) Output update, 3) Measurement update and 4) Time update. The mechanisms of these functions are given below:

State Update

The state update mechanism utilises the A, G and Q matrices discussed above. That is, the State dynamics matric, control input matrix, and variance in the system model. It also takes in the output value and adjusted throttle value from the controller.

Output

The output matric is multiplied by the output.

Measurement Update

Time Update

CONTROLLER

A PID controller is used to control the system. Again a class is used in Python and the mechanisms are :

class KalmanFilter:

def \_\_init\_\_(self, A, G, C, Q, R, P0, initial\_state):

self.A = A

self.G = G

self.C = C

self.Q = Q

self.R = R

self.sigma\_pred = P0

self.z\_pred = initial\_state

def state\_update(self, z, control\_input):

# Ensure w is a vector with the same number of rows as G has columns

w = np.random.normal(0, np.sqrt(self.Q), size=(self.G.shape[1], 1))

control\_effect = np.array([[0], [control\_input]]) # Assuming control affects the velocity

z\_next = self.A @ z + self.G @ w + control\_effect

return z\_next.reshape(-1, 1)

def output(self, z):

v = np.random.normal(0, np.sqrt(self.R))

y = self.C @ z + v

return y

def measurement\_update(self, y, z\_pred, sigma\_pred):

F = self.C @ sigma\_pred @ self.C.T + self.R

output\_error = y - self.C @ z\_pred

K = sigma\_pred @ self.C.T @ np.linalg.inv(F)

z\_corrected = z\_pred + K @ output\_error

sigma\_corrected = sigma\_pred - K @ self.C @ sigma\_pred

return z\_corrected, sigma\_corrected

def time\_update(self, z\_meas\_update, sigma\_meas\_update):

z\_predicted = self.A @ z\_meas\_update

sigma\_predicted = self.A @ sigma\_meas\_update @ self.A.T + self.Q \* (self.G @ self.G.T)

return z\_predicted, sigma\_predicted

# **Results**

# **Conclusion**