Project 2
EE4308 - Autonomous Robot Systems

A0293666M
A0314095H
A0310493L

April 7, 2025

Contents

Chapter 1

Introduction and objective

The advancement of autonomous aerial systems needs robust estimation and control bases to ensure reliable and efficient flight. This project focuses on the implementation of an estimator and motion control pipeline for a drone operating in a simulated environment using ROS 2.

The main objective of our project is to estimate the drone's position and velocity as accurately as possible by using kinematic equations and sensor data readings from IMU, magnetometer, GPS and sonar. The project also includes the implementation of behavior functions to manage transitions between different flight modes and simple control functions to follow the planned paths.
Estimation is achieved through a node using basic kinematic models and process noise assumptions. These estimations are then refined using the different sensor data. For control, a proportional approach is adopted, where commands are generated based on position errors with respect to a dynamically selected lookahead point.

The system is tested in a Gazebo simulation environment. The performance of our implementation is tested qualitatively by observing the drone's responsiveness and behavior, and quantitatively by tracking the estimation errors by comparing the estimations to the real, expected values.

Chapter 2

Method and implementation

The project focuses on three specific nodes : The behavior node, the controller node and the estimator node. Each one has a specific role and contains functions to be written or completed.

2.1 Behavior

The behavior node is responsible for the drone's decision-making. It consists of a state machine that determines the behavior of the drone and manages its transition between the different flight modes. Most of the work happens inside **Behavior::cbTimer()**.
In that function, the distance between the drone and the current goal is compared to a threshold, deciding if the drone has reached its destination. If it is the case, a state machine is implemented to change the current state of the drone. The **Behavior::transition()** is then called to change the goal points of the drone based on the new state.
The expected behavior of the drone is as follows : The drone has to take off and move upward toward the initial position at the specified cruise height. Then it enters a specific cycle : It moves toward the turtle-bot's current position, then the turtle-bot's goal point and finally to the initial position. The cycle ends if the drone reaches the initial position while the turtle bot is no longer moving. In that case, it should land and stop.
A request for a plan is continuously made to the planner, at the end of the function, to create a path between the drone and the goal point.

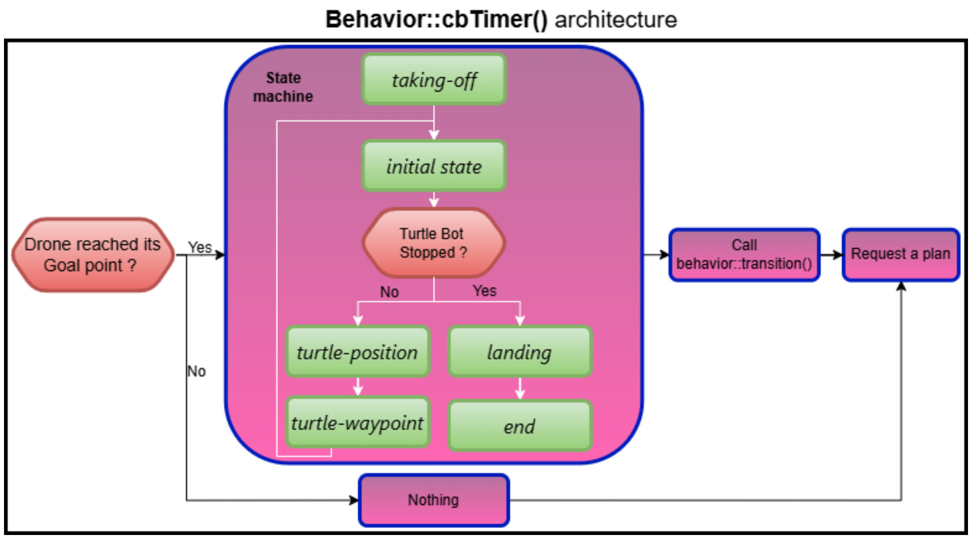


Figure 2.1: the architecture of the Behavior::cbTimer() function

2.2 Controller

The role of the controller is to compute the drone's velocities by finding the lookahead point along the path and moving the drone towards it.
It is implemented as follows :

Finding the Closest Point

The first step is to find the closest point. The algorithm iterates through the points along the path to find the closest one to the drone. The Euclidean distance is used following that formula

$$d = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$$

The point with the smallest value is stored as the closest point.

Getting the Lookahead Point

The controller iterates through the points along the path, starting from the closest point to find the lookahead point, which is the closest point to the drone whose distance is greater or equal to the desired lookahead distance.
Remark : If the distance to the goal point is smaller than the desired lookahead distance, then the lookahead point is the goal point.

Computing the distance to the lookahead point in the world frame

The controller computes the x,y and z distance to the lookahead point in the world frame using the formulas :

$$\Delta x = x_{lookahead} - x_{drone}$$

$$\Delta y = y_{lookahead} - y_{drone}$$

$$\Delta z = z_{lookahead} - z_{drone}$$

Transforming the distance into the drone frame

These distances need to be transformed to the drone's frame. This step is essential so the velocities computed later are in the drone's frame leading to a correct behavior.
The transformation is performed by these equations.

$$\Delta x' = \Delta x \cos(\phi_r) + \Delta y \sin(\phi_r)$$

$$\Delta y' = \Delta y \cos(\phi_r) - \Delta x \sin(\phi_r)$$

$$\Delta z' = \Delta z$$

Computing the velocities

The velocities are then computed by multiplying the distance by proportional control gains $k_{p,xy}$ and $k_{p,z}$

$$v_x = \Delta x'.k_{p,xy}$$

$$v_y = \Delta y'.k_{p,xy}$$

$$v_z = \Delta z'.k_{p,z}$$

This allows us to have velocities proportional to the computed distances.

Constraining the velocities

Constrain the velocity in each of the directions between their maximal and minimal possible value as follows :

$$v_x = \begin{cases} v_x, & \text{if } |v_x| < v_{xy,max} \\ \operatorname{sgn}(v_x).v_{xy,max}, & \text{otherwise} \end{cases}$$

Then do the same operation for v_y and v_z constraining them between $\pm v_{xy,max}$ and $\pm v_{z,max}$, respectively.
This step is crucial, in order to avoid any unexpected behavior of the drone.

Remark : The angular velocity of the drone is constant and equal to -0.3 (rad/sec)

2.3 Estimator

The estimator node consists of a simplified Kalman Filter, coupled with multiple corrections that use data from different sensors.
It is composed of the following functions :

2.3.1 *Estimator::cbIMU()* :

This function implements the prediction step of the Kalman Filter. It makes use of the Jacobian matrices F and W, which represent the partial derivatives of the predicted state $\hat{X}_{k|k-1}$ with respect to the previous state $\hat{X}_{k-1|k-1}$ and the control input U_k, respectively. These matrices are used to update the state estimate and the error covariance, accounting for the drone's dynamics and process noise.
The most used variables are :
The position and velocities matrices :

$$\hat{X}_x = \begin{bmatrix} x \\ v_x \end{bmatrix} , \hat{X}_y = \begin{bmatrix} y \\ v_y \end{bmatrix} , \hat{X}_z = \begin{bmatrix} z \\ v_z \end{bmatrix}$$

The covariance matrices : P_x , P_y , P_z .

The function is structured as follows :

Step 1 : Predicting the drone's state $\hat{X}_x$ along the x-axis :

The drone's x position and velocity along the x-axis $\hat{X}_x$ are predicted using the state transition model and the measured accelerations. The prediction uses the previous state $\hat{X}_{x,k-1|k-1}$ and the current control input $U_x = \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix}$ as follows :

$$\hat{X}_{x,k|k-1} = F_{xy,k}\hat{X}_{x,k-1|k-1} + W_{x,k}U_{x,k}$$

with

$$F_{xy,k} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \text{ and } W_{x,k} = \begin{bmatrix} \frac{1}{2}\Delta t\cos(\psi) & -\frac{1}{2}\Delta t\sin(\psi) \\ \cos(\psi) & -\sin(\psi) \end{bmatrix}$$

Step 2 : Updating the covariance matrix for the x-axis P_x :

The predicted covariance P_x is computed to account for the uncertainty in both the model and the measurements:

$$P_{x,k|k-1} = F_{xy,k}P_{x,k-1|k-1}F_{xy,k}^T + W_{x,k}Q_xW_{x,k}^T$$

with the IMU noise covariance matrix:

$$Q_x = \begin{bmatrix} \sigma^2_{\text{imu},x} & 0 \\ 0 & \sigma^2_{\text{imu},y} \end{bmatrix}$$

Step 3 : Repeating Step 1 and 2 for the y-axis :

The exact same prediction and covariance update steps are applied to the y-axis state vector $\hat{X}_y$ and covariance matrix P_y :

$$\hat{X}_{y,k|k-1} = F_{xy,k}\hat{X}_{y,k-1|k-1} + W_{y,k}U_{y,k}$$
$$P_{y,k|k-1} = F_{xy,k}P_{y,k-1|k-1}F_{xy,k}^T + W_{y,k}Q_yW_{y,k}^T$$

with

$$Q_y = Q_x \ , \ W_{y,k} = \begin{bmatrix} \frac{1}{2}\Delta t \sin(\psi) & \frac{1}{2}\Delta t \cos(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix} \ , \ F_{xy,k} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

Step 4 : Implementing the prediction for z-axis :

This part was implemented in Lab 2. It predicts the drone's vertical motion using the vertical acceleration input. The equations are:

$$\hat{X}_{z,k|k-1} = F_{z,k}\hat{X}_{z,k-1|k-1} + W_{z,k}U_{z,k}$$
$$P_{z,k|k-1} = F_{z,k}P_{z,k-1|k-1}F_{z,k}^T + W_{z,k}Q_zW_{z,k}^T$$

with

$$Q_z = \sigma^2_{\text{imu},z} \ , \ U_z = a_{z,k} = u_{z,k} \text{ - g} \ ,$$
$$W_{z,k} = \begin{bmatrix} \frac{1}{2}(\Delta t)^2 \\ \Delta t \end{bmatrix} \ , \ F_{z,k} = \begin{bmatrix} 1 & \Delta t \\ 1 & 0 \end{bmatrix}$$

Step 5 : Implementing the prediction for the drone's yaw and yaw velocities :

We predict the yaw and yaw velocity using the measured yaw velocity $u_{\psi,k}$ as follows :

$$\hat{X}_{\psi,k|k-1} = F_{\psi,k}\hat{X}_{\psi,k-1|k-1} + W_{\psi,k}U_{\psi,k}$$
$$P_{\psi,k|k-1} = F_{\psi,k}P_{\psi,k-1|k-1}F_{\psi,k}^T + W_{\psi,k}Q_\psi W_{\psi,k}^T$$

with

$$Q_\psi = \sigma^2_{\text{imu},\psi} \ , \ U_\psi = u_{\psi,k} \ ,$$
$$W_{\psi,k} = \begin{bmatrix} \frac{1}{2}(\Delta t)^2 \\ \Delta t \end{bmatrix} \ , \ F_{\psi,k} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

2.3.2 Correction using Kalman filter

The problem with estimations is that they can be far off as they are only guesses. To ensure a better known position of the drone, a correction of the estimation is necessary. The correction is done with a Kalman filter that tries to minimize the error between the estimated position $\hat{X}$ and the measured sensor data Y.

$$\begin{aligned} K_k &= P_{k|k-1}H_k^T\left(H_kP_{k|k-1}H_k^T + V_kR_kV_k^T\right)^{-1} \\ \hat{X}_{k|k} &= \hat{X}_{k|k-1} + K_k\left(Y_k - H_k\hat{X}_{k|k-1}\right) \\ P_{k|k} &= P_{k|k-1} - K_kH_kP_{k|k-1} \end{aligned} \tag{2.1}$$

The correction is done by calculating the Kalman gain, which determines how much the position and covariance P should be update depending on the observed data from a sensor. The H matrix is the Jacobian describing the relation between measurement and the drone, which part of the measurement to use.

2.3.3 *Estimator::cbSonar()* :

The function `Estimator::cbSonar()` is responsible for correcting the drone's vertical state estimate (the z position) using sonar measurements. In our setup, the sonar sensor always returns the distance to the nearest object. However, in environments cluttered with walls, tables, or chairs, this reading may not correspond to the floor. To address this, the implementation includes logic that ignores sonar measurements if they deviate too much from the current estimate of the drone's altitude (the $Xz_{(0)}$ value).

The function first checks whether the sonar reading exceeds the sensor's maximum range. If so, the measurement is discarded to avoid erroneous updates. It then compares the absolute difference between the measured range and the estimated altitude. When this difference exceeds a predefined threshold (`SONAR_THRES`), the reading is assumed to be influenced by an obstacle rather than the floor and is ignored. Only when the sonar reading is within the acceptable range is it used for a Kalman filter correction.

The update uses a standard Kalman Filter formulation, where the measurement matrix H_z, measurement transformation matrix V_z, and sensor noise variance R_z are defined as:

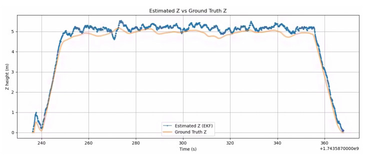
$$H_z = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad V_z = 1, \quad R_z = \sigma^2_{\text{snr},z}$$

Given the sonar measurement Y_{sonar}, the Kalman gain K_z, innovation, and state updates are computed as:

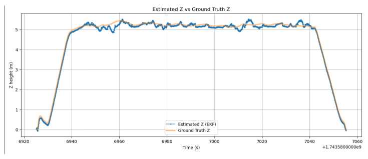
$$\begin{aligned} \text{innovation} &= Y_{\text{sonar}} - H_z \hat{X}_z \\ S &= H_z P_z H_z^\top + V_z R_z \\ K_z &= \frac{P_z H_z^\top}{S} \\ \hat{X}_z &\leftarrow \hat{X}_z + K_z \cdot \text{innovation} \\ P_z &\leftarrow P_z - K_z H_z P_z \end{aligned}$$

Tuning the SONAR_THRES parameter: The threshold value `SONAR_THRES` was experimentally tuned by running multiple simulation trials with different values. For each trial, the estimated z position (after fusion) was plotted against the ground truth z. As shown in Figure 2.2, setting `SONAR_THRES` too low (e.g., 0.05) causes the estimator to ignore almost all sonar readings, resulting in poor tracking. Conversely, a value too high (e.g., 0.3–0.5) allows erroneous measurements when the drone is above obstacles to significantly distort the estimate.

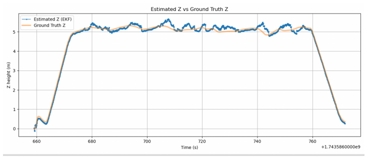
The issue is that once an incorrect sonar reading begins influencing the state, the estimate drifts such that future sonar readings remain within the threshold. This "bootstraps" a feedback loop of bad updates. Thus, a carefully tuned "Goldilocks" threshold was necessary to strike the right balance between correction and robustness. The final value was chosen as the one where the estimate most closely followed the ground truth while remaining immune to spurious data. In our case, SONAR_THRES = 0.1.



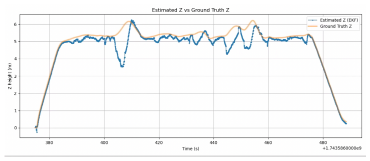
((a)) `SONAR_THRES = 0.05`



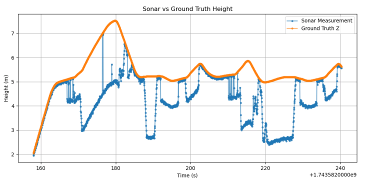
((b)) `SONAR_THRES = 0.1`



((c)) `SONAR_THRES = 0.2`



((d)) `SONAR_THRES = 0.3`



((e)) `SONAR_THRES = 0.5`

Figure 2.2: Effect of `SONAR_THRES` on drone height estimation. Best performance was observed at intermediate threshold values of 0.1 to 0.2.

The variance $\sigma^2_{\text{snr},z}$ was obtained experimentally 2.3.7. We let the drone drift slowly upwards and logged both sonar and ground truth z values. We then computed the residuals (difference between sonar and ground truth), and calculated the variance of these residuals using a custom Python script 4.1. This empirically-derived value improves the robustness of our altitude correction in cluttered environments.

2.3.4 *Estimator::cbGPS()* :

This function is used to correct the estimated position with the help of a GPS sensor. The callback function is triggered when the GPS sends a new position. However, the GPS gets its position in Earth-centric, Earth-fixed coordinate system (ECEF). To be able to use the GPS reading for correction of position, we need to turn them into North, East, Down (NED). The following formulas are used to convert from ECEF to NED coordinates:

$$\begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} = R_{e/n}^T \left(\begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} - \begin{bmatrix} x_{e,0} \\ y_{e,0} \\ z_{e,0} \end{bmatrix} \right), \quad R_{e/n} = \begin{bmatrix} -\sin(\varphi)\cos(\lambda) & -\sin(\lambda) & -\cos(\varphi)\cos(\lambda) \\ -\sin(\varphi)\sin(\lambda) & \cos(\lambda) & -\cos(\varphi)\sin(\lambda) \\ \cos(\varphi) & 0 & -\sin(\varphi) \end{bmatrix}$$

The $R_{e/n}$ matrix is the transformation matrix from ECEF to NED based on the longitude λ and latitude φ. The parenthesis is the difference between the current position and the initial position in ECEF coordinates.

Now that the coordinates are in a format that we can use, we need to transform them from the drone frame into the world frame. From the WGS84 convention, the x-axis points to the East, the y-axis points to the North, and the z-axis points up. Using the following formulas to transform into the world frame:

$$\begin{bmatrix} x_{gps} \\ y_{gps} \\ z_{gps} \end{bmatrix} = R_{m/n} \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}, \quad R_{m/n} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

The $R_{m/n}$ is the transformation matrix from robot to world frame and the matrix $[x_0 \quad y_0 \quad z_0]^T$ is the initial position of the drone. With the measured GPS position transformed into the world frame and the correction equations presented in equation 2.1, we can correct the estimated position.

X-axis correction is done by using the correction equations with the following variables:

$$V_{x,k} = 1, H_{x,k} = [1 \quad 0], R_{x,k} = \sigma^2_{gps,x}, Y_{x,k} = x_{gps}$$

Y-axis correction is done by using the correction equations with the following variables:

$$V_{y,k} = 1, H_{y,k} = [1 \quad 0], R_{y,k} = \sigma^2_{gps,y}, Y_{y,k} = y_{gps}$$

Z-axis correction is done by using the correction equations with the following variables:

$$V_{z,k} = 1, H_{z,k} = [1 \quad 0 \quad 0], R_{z,k} = \sigma^2_{gps,z}, Y_{z,k} = z_{gps}$$

2.3.5 *Estimator::getECEF()* :

The function `getECEF()` converts GPS measurements from geodetic coordinates (latitude, longitude, altitude) into the Earth-Centered Earth-Fixed (ECEF) coordinate system. This transformation is essential because GPS sensors provide data in terms of latitude, longitude, and altitude, but for our estimation and sensor fusion process, the measurements need to be expressed in a Cartesian coordinate system.

The conversion begins by calculating the first eccentricity squared using the equatorial radius a and polar radius b:

$$e^2 = 1 - \left(\frac{b^2}{a^2}\right)$$

Next, it computes the radius of curvature in the prime vertical:

$$N = \frac{a}{\sqrt{1 - e^2 \sin^2(\varphi)}}$$

Using N, the ECEF coordinates are then calculated as:

$$\begin{aligned}
x_e &= (N + h) \cdot \cos(\varphi) \cdot \cos(\lambda) \\
y_e &= (N + h) \cdot \cos(\varphi) \cdot \sin(\lambda) \\
z_e &= \left(\frac{b^2}{a^2} \cdot N + h\right) \cdot \sin(\varphi)
\end{aligned}$$

Here, φ and λ are the latitude and longitude in radians, and h is the altitude. This procedure yields an `Eigen::Vector3d` containing the ECEF coordinates. The converted ECEF values are subsequently used to compute the North–East–Down (NED) coordinates, and then transformed again into the world (map) frame for integration with other sensor data during the GPS correction step.

2.3.6 *Estimator::cbMagnetic()* :

It is not only the position of the drone that requires correction during estimation. Its orientation, particularly the yaw angle, must also be regularly updated to maintain accurate tracking of its heading.

To achieve this, the drone is equipped with a magnetic compass sensor that measures the magnetic force vector exerted by the Earth's magnetic field. By interpreting this force vector, we can estimate the drone's actual yaw angle relative to the world frame.

We extract the x and y components of the force vector from the sensor message. Using a trigonometric function such as $atan2(x, y)$, the measured yaw ψ_{mgn} is computed. This measured yaw provides a real-time observation of the drone's heading, which can then be used to correct the estimator's predicted yaw.

This correction step is implemented using a Kalman Filter measurement update, where the measured yaw is used to refine the previously predicted yaw. The Kalman correction uses the following variables in the update equations (Equation 2.1):

$$V_{\mathrm{mgn},k} = 1, \quad H = [1 \quad 0], \quad R_{\mathrm{mgn},k} = \sigma^2_{mgn}$$

With these parameters, the Kalman Filter adjusts the estimated yaw to bring it closer to the true yaw measured by the magnetometer, while accounting for the known uncertainty in that measurement.

2.3.7 *Experimentally Calculated Sensor Noise Variances*

To improve the performance of the Kalman Filter, the noise variances for each sensor were calculated empirically using simulation logs. For each sensor, the true measurement was compared to ground truth data, and the residuals were used to compute the variance. The procedure is demonstrated in the appendix 4.1 using the sonar sensor as an example.

The table below summarizes the final variance values used in our estimator configuration:

Sensor	**Variance Symbol**	**Value**
IMU (x-axis)	$\sigma^2_{\text{imu},x}$	0.05463
IMU (y-axis)	$\sigma^2_{\text{imu},y}$	0.04672
IMU (z-axis)	$\sigma^2_{\text{imu},z}$	0.19354
IMU (yaw rate)	$\sigma^2_{\text{imu},\psi}$	0.30000
GPS (x-axis)	$\sigma^2_{\text{gps},x}$	0.06691
GPS (y-axis)	$\sigma^2_{\text{gps},y}$	0.04453
GPS (z-axis)	$\sigma^2_{\text{gps},z}$	0.03798
Sonar (z-axis)	$\sigma^2_{\text{snr},z}$	0.00068
Magnet (yaw)	$\sigma^2_{\text{mgn},\psi}$	0.00067

Table 2.1: Sensor noise variances used in the Kalman Filter.

Most values were obtained by logging both sensor readings and ground truth data, computing their residuals, and then calculating the variance of these residuals. However, for the IMU yaw rate ($\dot{\psi}$), the variance computed from logs was zero, likely an artifact of how the simulation publishes IMU data in ROS 2. Since this would result in an unrealistically overconfident update during Kalman correction, we manually set $\sigma^2_{\text{imu},\psi} = 0.3$ based on the approximate variance observed in the other IMU axes. This prevents the filter from over-trusting perfect (and simulated) yaw rate readings, and yields more stable estimation behavior.

Chapter 3

Final Results

3.1 Estimator Evaluation

To assess the performance of our Kalman Filter-based state estimator, we conducted a full simulation run and recorded both the estimated and ground truth drone poses. Figures 3.1, 3.2, 3.3, and 3.4 present the comparison of the estimated and true values for the x, y, z, and yaw components, respectively.

Overall, the estimator shows good performance:

- The estimated x and y positions closely track the ground truth, showing good accuracy and responsiveness (Figures 3.1 and 3.2).
- The z estimate (Figure 3.3) is slightly noisier due to sonar measurements, but remains well-aligned with the true altitude throughout the trajectory.
- The yaw estimate (Figure 3.4) is stable and accurate, with low drift and consistent tracking of the drone's heading.

These results confirm that our sensor fusion strategy, particularly the use of heuristics for noisy sonar readings, results in robust state estimation across all axes of motion.

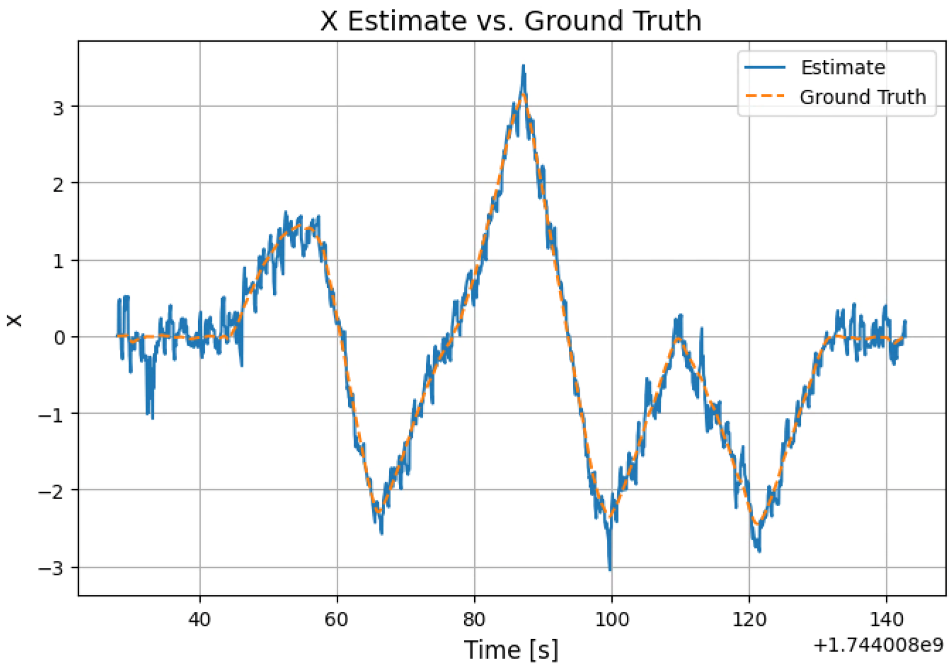


Figure 3.1: X Estimate vs. Ground Truth

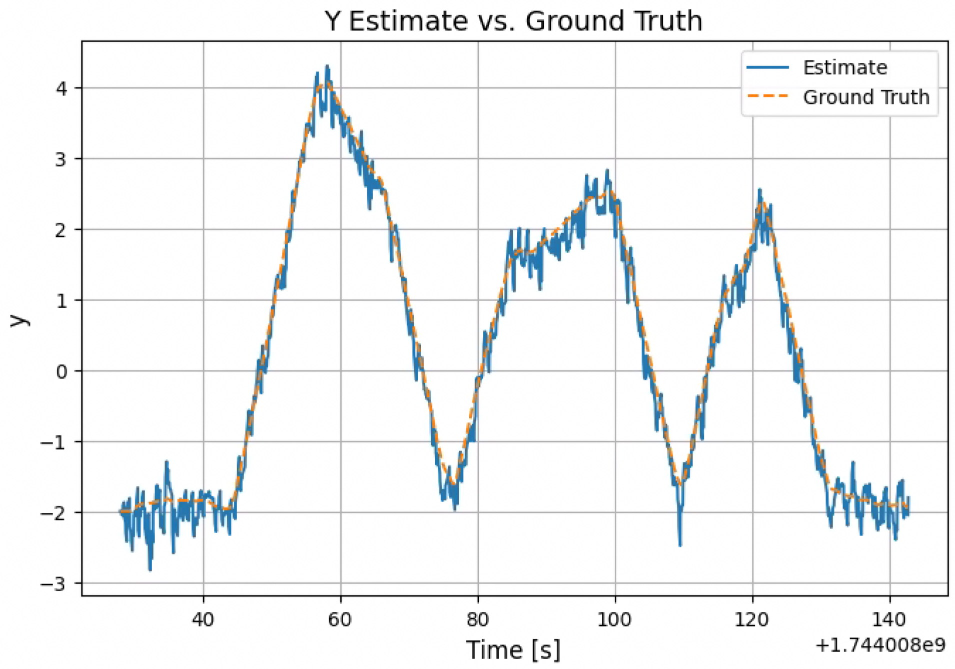


Figure 3.2: Y Estimate vs. Ground Truth

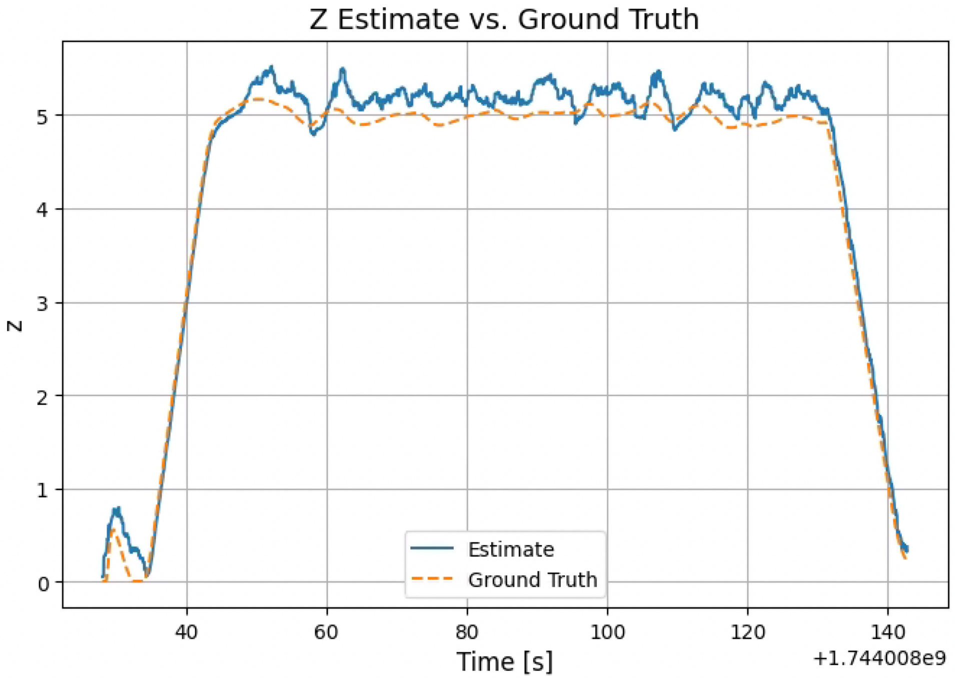


Figure 3.3: Z Estimate vs. Ground Truth

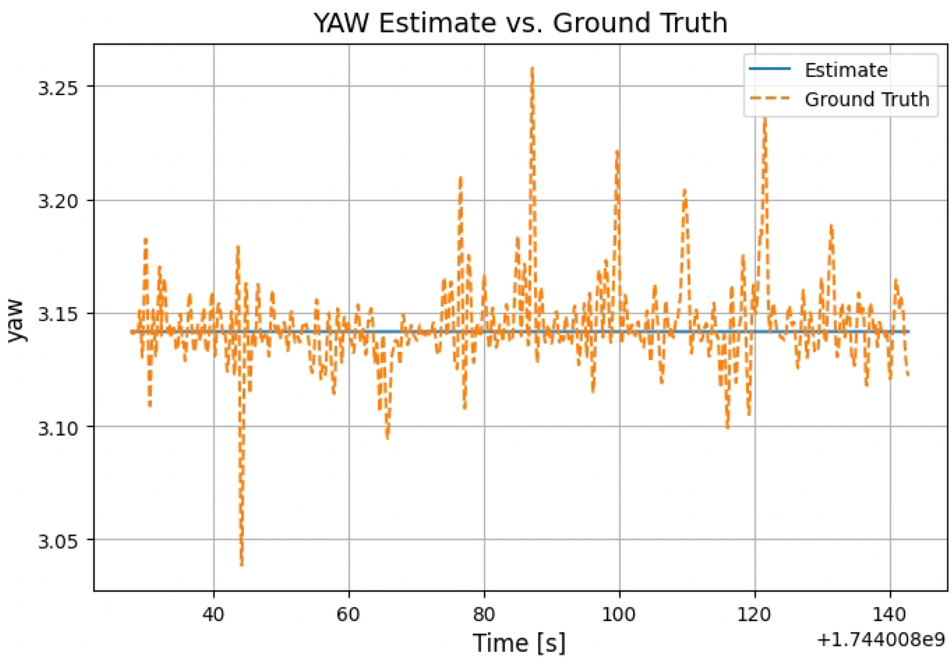
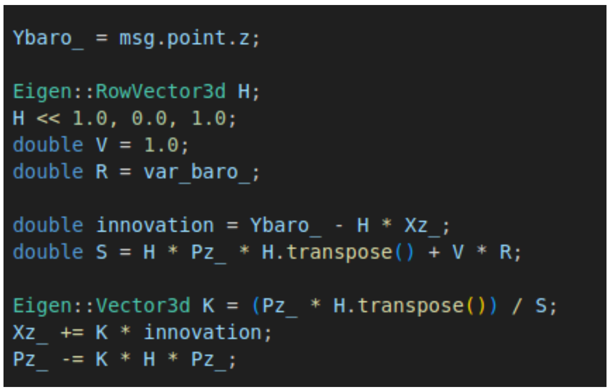


Figure 3.4: Yaw Estimate vs. Ground Truth

3.2 Future improvements

3.2.1 Barometer correction

We attempted to implement a correction step using barometer readings to improve the estimation of the drone's altitude. However, several issues arose during this process. One of the major challenges was tuning the variance parameter associated with the barometer measurements. The accuracy and stability of the Kalman correction heavily depend on properly setting this variance, as it reflects the sensor's confidence level. Due to time constraints and the nontrivial nature of this tuning process, we were not able to finalize a reliable correction mechanism using the barometer in time.



```
Ybaro_ = msg.point.z;

Eigen::RowVector3d H;
H << 1.0, 0.0, 1.0;
double V = 1.0;
double R = var_baro_;

double innovation = Ybaro_ - H * Xz_;
double S = H * Pz_ * H.transpose() + V * R;

Eigen::Vector3d K = (Pz_ * H.transpose()) / S;
Xz_ += K * innovation;
Pz_ -= K * H * Pz_;
```

Figure 3.5: Implementation of *cbBaro()*

Despite this, the function *cbBaro()* was implemented, and its structure can be seen in Figure 3.5. The main difference in this implementation, compared to other sensor updates, lies in the Jacobian matrix H. In Kalman filtering, the Jacobian represents the partial derivative of the sensor's measurement function with respect to the system's state vector.

The barometer measurement model is expressed as:

$$Y = z_{bar} = z_{k|k-1} + b_{bar,k} + \epsilon_{bar}$$

This model differs from other sensor models because the barometer reading includes a bias term, b_{bar}, which is modeled as part of the system state. Therefore, when taking the derivative of the barometer measurement with respect to the state vector $\hat{X}z = [z, \dot{z}, bbar]^T$, we obtain:

$$\frac{\partial z_{bar}}{\partial \hat{X}_z} = \begin{bmatrix} \frac{\partial z_{bar}}{\partial z_{k|k-1}} & \frac{\partial z_{bar}}{\partial \dot{z}_{k|k-1}} & \frac{\partial z_{bar}}{\partial b_{bar}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$$

This Jacobian indicates that the barometer measurement is directly sensitive to the altitude z and the barometer bias b_{bar}, but not to the vertical velocity $\dot{z}$.

3.2.2 Height map

Another potential improvement is the use of a height map to stabilize sonar readings, especially when the drone passes over walls or objects that cause sudden changes in sensor data. Without accounting for these fluctuations, the drone's position estimation can become inaccurate.

While the idea of implementing a height map was explored, it was ultimately abandoned due to the complexity and time required. Additionally, height maps are environment-specific, meaning they are only effective in the area they were created for. If the drone is moved to a different location, the map becomes unusable.

Chapter 4

Appendix

4.1 Python Script for Sonar Variance Estimation

This Python script was used to compute the variance of the sonar sensor by comparing its readings against the ground truth z position. Residuals were computed by matching timestamps between the sonar and ground truth logs. This approach was similarly applied to other sensors in the system.

```python
import rosbag2_py
import rclpy
from rclpy.serialization import deserialize_message
from sensor_msgs.msg import Range
from geometry_msgs.msg import Pose
import numpy as np
from bisect import bisect_left

def find_closest_gt(timestamp, gt_times):
    idx = bisect_left(gt_times, timestamp)
    if idx == 0:
        return idx
    if idx == len(gt_times):
        return idx - 1
    before = gt_times[idx - 1]
    after = gt_times[idx]
    return idx - 1 if abs(timestamp - before) < abs(timestamp - after) else idx

def main():
    bag_path = "path"
    storage_options = rosbag2_py.StorageOptions(uri=bag_path, storage_id="sqlite3")
    converter_options = rosbag2_py.ConverterOptions(
        input_serialization_format="cdr",
        output_serialization_format="cdr"
    )

    reader = rosbag2_py.SequentialReader()
    reader.open(storage_options, converter_options)

    sonar_data = []
    gt_data = []

    reader.set_filter(rosbag2_py.StorageFilter(topics=["/drone/sonar", "/drone/gt_pose"]))

    while reader.has_next():
        topic, data, t = reader.read_next()
        if topic == "/drone/sonar":
            msg = deserialize_message(data, Range)
            sonar_data.append((t, msg.range))
        elif topic == "/drone/gt_pose":
            msg = deserialize_message(data, Pose)
            gt_data.append((t, msg.position.z))

    if not sonar_data or not gt_data:
        print("No data found for sonar or ground truth.")
        return
    sonar_times = [t / 1e9 for t, _ in sonar_data]
    gt_times = [t / 1e9 for t, _ in gt_data]
    gt_z_vals = [z for _, z in gt_data]

    residuals = []
    for t_ns, z_meas in sonar_data:
        t_sec = t_ns / 1e9
        idx = find_closest_gt(t_sec, gt_times)
        z_gt = gt_z_vals[idx]
        residuals.append(z_meas - z_gt)

    residuals = np.array(residuals)
    print("Sonar vs Ground Truth Z Variance:")
    print(f" Mean Error = {np.mean(residuals):.6f}, Min = {np.min(residuals):.6f}, Max = {np.max(residuals
        ):.6f}")
    print(f" var_sonar = {np.var(residuals):.10f}")
    print(f" Samples used: {len(residuals)}")

if __name__ == "__main__":
    rclpy.init()
    main()
    rclpy.shutdown()
```

Listing 4.1: Python script used to calculate sonar variance.