Project 2 Report - EE4308

Drone Simulation With Simplified Kalman Filter

A0315270N
A0315021B
A0314731M

April 7, 2025

Contents

1 Introduction

Goal of this project was to let a quadcopter navigate along multiple waypoints in a virtual environment. To achieve this, we designed controller and estimator plugins. For the controller, we implemented a pure pursuit controller that was similar to the one from lab and project 1. For estimation, we implemented a Kalman filter that uses sonar, GPS, magnetic compass and barometer measurements to predict and correct the quadcopters pose. We the aimed to improve the pure pursuit controller from lab 1 using simple heuristics. Using this controller and an improved Kalman filter algorithm, the drone should take off, follow a moving turtlebot and fly to the turtlebots goal points in an alternating fashion and then land again.

2 Background and Related Work

In this section we discuss the relevant existing algorithms and their limitations.

2.1 Behavior Node: State Machine

The Behavior Node is a simple state machine that mainly determines the next waypoint of the drone. As illustrated in Figure 1, the drone starts in the state `TAKEOFF` and then transits into the state `INITIAL`. Both of these states have the the same waypoint which is located at the initial air position on cruise height perpendicularly above the initial ground position. After reaching the initial air position, the state transits to `TURTLE_POSITION` which describes the behavior of the drone when it is moving towards the turtlebot. Once the turtlebot is reached, the new state becomes `TURTLE_WAYPOINT` that sets the new waypoint for the quadcopter at cruising height above the next waypoint, respectively goal, of the turtlebot. This cycle is repeated as many times as possible in the given time limit and after this limit is reached, the drone lands at the initial ground position again (`LANDING` state). After landing, the state machine switches to the `END` state which describes the state when the drone prepares stopping the movement after the landing is completed.

2.2 Controller Node: Holonomic Pure Pursuit

Our controller node implements a holonomic variant of the pure pursuit algorithm to guide the quadcopter along a 3D path including multiple waypoints. Unlike the non-holonomic TurtleBot in Project 1, the drone can move freely

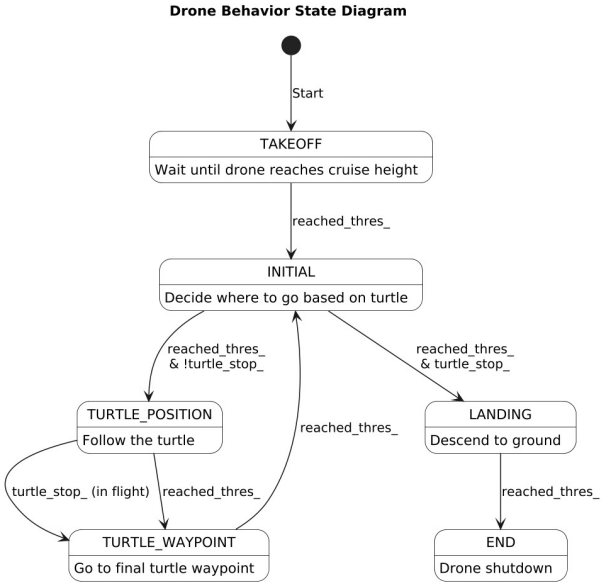

Figure 1: State Machine for Behavior Node

in x, y, and z, so we omit any curvature computation and directly compute velocity setpoints in the body frame.

Key Steps of our Algorithm

1. **Enable Check & Path Validity**
   - If the controller is disabled, the callback returns immediately.
   - If the path is empty, the drone is commanded to hover (zero velocity) and the callback returns.
2. **Look-Ahead Point Selection**
   - Identify the closest point on the current path to the drone's position.
   - From that index, search forward to select the look-ahead point at a fixed distance along the path. If the drone is sufficiently close to the waypoint, we select the last point of the path.
3. **Velocity Computation in Body Frame**
   - Compute the 3D error vector $\mathbf{e} = [dx,\, dy,\, dz]^{\mathsf{T}}$ between the drone and the look-ahead point in the world frame.

- Rotate $\mathbf{e}$ into the drone's body frame using the current yaw ψ:

$$dx_{\text{body}} = dx\cos\psi + dy\sin\psi,$$
$$dy_{\text{body}} = -dx\sin\psi + dy\cos\psi.$$

- Apply a proportional control law on each axis:

$$v_x = K_p\, dx_{\text{body}}, \qquad v_y = K_p\, dy_{\text{body}}, \qquad v_z = K_{p,z}\, dz.$$

4. **Command Limiting**
   - Enforce maximum speed constraints in the horizontal plane and on the vertical axis.
5. **Publish**
   - Send constrained (v_x, v_y, v_z) commands along with the desired yaw rate to the flight controller.

2.3 Estimator Node: Kalman Filter

This node implements a simplified Extended Kalman Filter (EKF) to estimate the drone's position, velocity, and yaw based on IMU measurements (for prediction) and multiple sensors (for correction). The key states we track are $\hat{\mathbf{X}}$, which includes $\left[x, \dot{x},\, y, \dot{y},\, z, \dot{z},\, \psi, \dot{\psi},\, b_{bar}\right]$ depending on whether barometer bias is included. The general flow of the filter is:

1. **Predict** the state and covariance forward in time using the latest IMU measurements.
2. **Correct** the predicted state whenever new sensor data arrives (GPS, sonar, compass, barometer).

Below, each stage is discussed in detail.

2.3.1 Prediction

The prediction step propagates the state from time step $k-1$ to k using a (possibly linearized) process model:

$$\hat{\mathbf{X}}_{k|k-1} \;=\; f\big(\hat{\mathbf{X}}_{k-1|k-1},\, \mathbf{U}_k\big),$$

where $\hat{\mathbf{X}}$ is the estimated state, and $\mathbf{U}_k$ is the input (e.g., IMU acceleration, yaw rate). In an EKF, f is approximated using first-order Taylor expansions with Jacobians $\mathbf{F}_k$ and $\mathbf{W}_k$:

$$\hat{\mathbf{X}}_{k|k-1} \;=\; \mathbf{F}_k\, \hat{\mathbf{X}}_{k-1|k-1} \;+\; \mathbf{W}_k\, \mathbf{U}_k.$$

The corresponding covariance update is:

$$\mathbf{P}_{k|k-1} \;=\; \mathbf{F}_k\,\mathbf{P}_{k-1|k-1}\,\mathbf{F}_k^T \;+\; \mathbf{W}_k\,\mathbf{Q}_k\,\mathbf{W}_k^T,$$

where $\mathbf{Q}_k$ is the process noise. For uncorrelated noise sources,

$$\mathbf{Q}_k = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}.$$

1.1 Coordinate Transforms Because the IMU measures accelerations in the drone's local frame, we must rotate them into the world frame. For a drone yaw angle ψ:

$$\begin{bmatrix} a_{x,k} \\ a_{y,k} \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} u_{x,k} \\ u_{y,k} \end{bmatrix},$$

where $(u_{x,k}, u_{y,k})$ are the IMU accelerations in the drone frame.

1.2 Motion Model for $(x, \dot{x})$ Focusing on the x-axis:

$$\begin{bmatrix} x_{k|k-1} \\ \dot{x}_{k|k-1} \end{bmatrix} = \begin{bmatrix} x_{k-1|k-1} \;+\; \dot{x}_{k-1|k-1}\,\Delta t \;+\; \frac{1}{2}(\Delta t)^2\,a_{x,k} \\ \dot{x}_{k-1|k-1} \;+\; a_{x,k}\,\Delta t \end{bmatrix}.$$

In matrix form:

$$\hat{\mathbf{X}}_{x,k|k-1} = \mathbf{F}_{x,k}\,\hat{\mathbf{X}}_{x,k-1|k-1} + \mathbf{W}_{x,k}\,\mathbf{U}_{x,k},$$

where

$$\mathbf{F}_{x,k} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}, \quad \mathbf{W}_{x,k} = \begin{bmatrix} \frac{1}{2}(\Delta t)^2 & \Delta t \end{bmatrix}.$$

The y-axis updates $(y, \dot{y})$ follow the same approach.

1.3 Vertical Motion (z-Axis) Similarly for the z-axis, except that we also incorporate gravity G

$$a_{z,k} = u_{z,k} - G,$$

so that the position and velocity update remain in the same constant-acceleration form (we subtract G from $u_{z,k}$).

1.4 Yaw States Yaw ψ is updated using the IMU's yaw rate $u_{\psi,k}$:

$$\psi_{k|k-1} \;=\; \psi_{k-1|k-1} + \dot{\psi}_{k-1|k-1}\,\Delta t, \qquad \dot{\psi}_{k|k-1} \;=\; \dot{\psi}_{k-1|k-1} + u_{\psi,k}\,\Delta t.$$

Its Jacobian structure parallels the other axes (position–velocity style).

2.3.2 Correction

Whenever new sensor data arrives, the filter refines the prediction via:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\,\mathbf{H}_k^T\big(\mathbf{H}_k\,\mathbf{P}_{k|k-1}\,\mathbf{H}_k^T + \mathbf{V}_k\,\mathbf{R}_k\,\mathbf{V}_k^T\big)^{-1},$$

$$\hat{\mathbf{X}}_{k|k} = \hat{\mathbf{X}}_{k|k-1} \;+\; \mathbf{K}_k\big(\mathbf{Y}_k - \mathbf{H}_k\,\hat{\mathbf{X}}_{k|k-1}\big),$$

$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} \;-\; \mathbf{K}_k\,\mathbf{H}_k\,\mathbf{P}_{k|k-1}.$$

Here, $\mathbf{Y}_k$ is the sensor measurement, $\mathbf{H}_k$ is the measurement Jacobian, $\mathbf{V}_k$ is any transform for noise, and $\mathbf{R}_k$ is the sensor covariance. If no measurement arrives, there is no correction step.

2.1 Sonar (Altitude) Sonar gives the drone's height. Since it measures z in the same frame:

$$\mathbf{Y}_{snr,z,k} = z_{snr} = z_{k|k-1} + \varepsilon_{snr}, \quad \mathbf{H}_{snr,z,k} = \begin{bmatrix}1 & 0\end{bmatrix}, \quad \mathbf{R}_{snr,z,k} = \sigma^2_{snr,z}.$$

2.2 GPS (Global Position) GPS data in latitude/longitude/altitude must be converted to ECEF and then rotated into the local world frame:

$$\begin{bmatrix} x_{gps} \\ y_{gps} \\ z_{gps} \end{bmatrix} = \mathbf{R}_{m/n}\left(\mathbf{R}_{e/n}^T\left(\begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} - \begin{bmatrix} x_{e,0} \\ y_{e,0} \\ z_{e,0} \end{bmatrix}\right)\right) + \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}.$$

After this transform, each axis is directly comparable to the filter's states. For example:

$$x_{gps} = x_{k|k-1} + \varepsilon_{gps,x} \implies \mathbf{H}_{gps,x} = \begin{bmatrix}1 & 0\end{bmatrix}, \quad \mathbf{R}_{gps,x} = \sigma^2_{gps,x}.$$

Similarly for y and z.

2.3 Magnetic Compass (Yaw) A magnetometer provides heading:

$$\psi_{mgn} = \psi_{k|k-1} + \varepsilon_{mgn}, \quad \mathbf{H}_{mgn,\psi} = \begin{bmatrix}1 & 0\end{bmatrix}, \quad \mathbf{R}_{mgn,\psi} = \sigma^2_{mgn,\psi}.$$

2.4 Barometer (Optional Bias) To handle barometer bias, we augment our state as $\hat{\mathbf{X}}_z = \begin{bmatrix} z, & \dot{z}, & b_{bar}\end{bmatrix}^T$. The barometer measurement is:

$$z_{bar} = z_{k|k-1} \;+\; b_{bar,k} \;+\; \varepsilon_{bar}, \quad \mathbf{H}_{bar,z} = \begin{bmatrix}1 & 0 & 1\end{bmatrix}.$$

Estimating b_{bar} improves long-term altitude stability.

2.3.3 Summary

In summary, the drone's full 3D motion and yaw are tracked by continuously predicting with IMU accelerations and adjusting estimates with various sensor measurements. This ensures robust, real-time estimation for navigation and control, while compensating for noise and drift in the sensor data.

2.4 Problems and Limitations

Limitations of the Pure Pursuit Controller

The standard pure pursuit controller, which relies solely on a proportional gain to fly the drone towards a look-ahead point, exhibits several practical shortcomings. Firstly, high proportional gains, which are necessary for rapid convergence, tend to induce overshoot and oscillations around the path, as there is no mechanism to dampen the response. Secondly, any measurement noise or abrupt changes in the reference trajectory directly translate into equally abrupt velocity commands, leading to jerky changes in velocity.

Limitations of the Estimator

We noticed the following problems with the estimator:

1. **Sensor Anomalies and Obstacles**

   When flying over obstacles such as walls or furniture, the downward-facing sonar sensor reports a significantly lower altitude than the drone's actual and desired cruise height. This causes the estimator to believe the drone is too low, resulting in an abrupt correction that makes the quadcopter shoot upward. This leads to choppy and unstable flight behavior, especially in cluttered environments.

2. **High Sensitivity to Sensor Noise in the Absence of Ground Truth**

   We observed a large degradation in estimator performance when switching from ground-truth data to onboard sensor measurements (i.e., using the data from estimator.cpp instead of the odom topic). This indicates that the Kalman filter is highly sensitive to sensor noise and biases, which leads to drift and reduced accuracy over time in the absence of reliable reference data.

3. **Assumption of Uncorrelated Sensor Measurements**

   The standard Kalman filter algorithm treats each sensor measurement as an independent input, assuming no correlation between them. In practice, multiple sensors (e.g., sonar and barometer) may provide estimates of the same state variable, such as altitude, which means their measurements are correlated. If these sensors have different biases or noise characteristics, their conflicting updates may cause instability or oscillations in the estimated state. Without proper handling of cross-sensor covariance, the estimator may overreact to one sensor's measurement, only to be corrected by another shortly after, leading to inefficiencies and potential instability.

4. **Linear Assumptions in a Nonlinear System**

   The standard Kalman filter we implemented assumes a linear system model, which does not correspond perfectly accurately with the quadcopter dynamics, especially during aggressive maneuvers or in the presence of nonlinear sensor models (e.g., sonar affected by surface angles). This mismatch can lead to suboptimal estimation and control.

5. **Static Noise Covariance Matrices**

   The process and measurement noise covariances are fixed in the basic Kalman filter. However, in real-world scenarios, sensor noise can vary significantly depending on the environment (e.g., barometer drift due to temperature changes or sonar noise near objects). Therefore, a static noise model may reduce the filter's robustness.

Barometer and GPS Measurement Issues

We noticed that even though the barometer measurement data has low variances, the actual data it produced varied a lot between different runs. For example, after instantiation of run 1, the barometer output corresponded to -12 meters in altitude and after run 2 the output was +7 meters in altitude. The interesting thing is that even though these measurements differ by such a big amount, the variances during each run remain very small. This made us question whether we can consider the barometer data useful in any way.

We had a similar problem with the GPS. The GPS data is also very inaccurate and the default assumed noise variances for the x and y direction were too low by orders of magnitude, giving the GPS measurements too big of an impact on the estimation of the drone's pose.

3 Proposed Improvements

In this section we describe our simple solutions to improve the existing algorithms.

3.1 Controller Improvements

In our pure pursuit controller, we've augmented the classic proportional-only scheme with two key features: a **derivative term** and a **low-pass filter** to improve tracking performance and command smoothness.

Derivative Term

We introduced this term to ensure **damping** and **stability** in our controller: By reacting to the rate of change of the tracking error, the derivative term counteracts rapid error excursions and reduces overshoot when using high proportional gains. We want to use a high proportional gain such that the drone reaches the waypoints faster. The derivative error is given by

$$d_{error}(t) \approx \frac{e(t) - e(t - \Delta t)}{\Delta t},$$

where $e(t)$ is the current body frame-error, e.g. dx_{body}. In code our implementation looks like this:

```
double der_x = (dx_body - prev_error_x_) / dt;
// ...
x_vel = kp_xy_ * dx_body + kd_xy_ * der_x;
```

Here, k_d scales the derivative of the error, adding a velocity-dependent "braking" action.

Low-Pass Filter

The purpose of the low-pass filter is to **smoothen the velocity commands**: The low-pass filter filters out high-frequency fluctuations in the computed velocity commands. By reducing big velocity command differences, the overall flight behavior is smoothened and less jerky. If this project were to be implemented on actual hardware and thested in a physical environment this would also improve hardware longevity (because the actuator movements are less aggressive). Mathematically the discrete first order low-pass filter looks as follows:

$$v_{filt}[n] = \alpha v_{filt}[n-1] + (1-\alpha)v_{raw}[n],$$

where $\alpha \in [0, 1]$ is the smoothing factor, v_{raw} is the newly computed velocity command and v_{filt} is the filter output. In code our implementation looks like this:

```
x_vel = alpha_ * prev_x_vel_ + (1.0 - alpha_) * x_vel;
prev_x_vel_ = x_vel;
```

This operation is applied separately on each axis.

Combined Control Law

Putting it all together, for the x-axis in body frame:

1. **Error**: $e_x = x_{goal} - x_{pos}$
2. **Derivative**: $\dot{e}_x \approx (e_x - e_{x,prev})/\Delta t$
3. **Raw Command**: $v_{x,raw} = K_p e_x + K_d \dot{e}_x$
4. **Filtered Command**: $v_{filt}[n] = \alpha v_{filt}[n-1] + (1-\alpha) v_{raw}[n]$

By implementing these enhancements, our controller achieves faster convergence (thanks to a larger K_p) while maintaining stability and producing smooth velocity commands.

3.2 Estimator Improvements

3.2.1 χ^2-Gating to Reject Extreme Outliers

As mentioned in 2.4, our biggest issue with the estimator was sensor noise and obstacles below the drone. At first we adjusted the algorithm to only update the sonar measurements if there is no obstacle, respecively wall or furniture, underneath the drone. However, this did not work as well as we hoped it would and the issue remained. To tackle this problem, we modified the algorithm to include a statistical test that detects and rejects outliers in measurements such that the overall algorithm is less sensitive to sensor noise and obstacles. To handle these sudden anomalies in sensor data, we implemented an outlier rejection mechanism based on the Mahalanobis distance. If the distance between the predicted and actual measurement exceeds a threshold derived from the χ^2-distribution, the measurement is considered an outlier and is ignored. This approach is particularly useful to reject noise and in situations where the drone suddenly flies over an obstacle, such as a wall. In such cases, the predicted measurement (based on smooth flight dynamics) deviates significantly from the actual sonar reading, which drops sharply due to the presence of the obstacle. The statistical test interprets

this discrepancy as an outlier and discards the measurement. As a result, the estimator continues to rely on the most recent valid sonar reading—taken before the wall—while subsequent measurements affected by the obstacle are consistently rejected. This prevents abrupt corrections based on unreliable data, but it also introduces a temporary lag in altitude updates by the sonar. Since we also have other sensors to estimate the altitude and the walls are generally thin, this should not pose a big problem. Mathematically, our approach for the sonar sensor looks as follows:

As described in lab 1, the innovation is given by

$$\mathbf{\Gamma} = \mathbf{Y}_k - \mathbf{H}_k \hat{\mathbf{X}}_{k|k-1}$$

The innovation covariance can then be calculated as

$$\mathbf{S} = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{V}_k \mathbf{R} \mathbf{V}_k^T$$

Note that in this case $\mathbf{V} = \mathbf{V}^T = 1$ like in lab 1. Using these two equations, the squared Mahalanobis distance is given by

$$d^2 = \mathbf{\Gamma}^T \mathbf{S}^{-1} \mathbf{\Gamma}$$

If $d^2 > \chi^2_\alpha(n)$, then we reject the measurement. $\chi^2_\alpha(n)$ is the chi-square value for confidence level α and n degrees of freedom. We used a 95% confidence level and $n = 1$ degrees of freedom.

We observed a **massive improvement** with the χ^2-gating over our initial implementation when it comes to flying over obstacles.

Additionally to the χ^2-gating, we implemented a student's t-distribution for the sonar sensor data. We multiplied the Kalman gain with a weight, that further punishes outliers. The weight is calculated as follows:

$$w_t = (v + n)/(v + d^2)$$

However, after profiling the sensor measurement noise distributions (next section), we observed that the sonar sensor noise corresponds to a gaussian distribution pretty well, wo we discarded the student's t-distribution again.

3.2.2 Profiling Sensor Measurement Noise Variances

Initially, the sensor noise variances were more or less randomly guessed which is why we implemented a python script to calculate more accurate estimates for the variances. We just let the drone stand still, tracked a large number

of measurements from all the sensors and then calculated all the variances. The process was very similar to how we calculated the sonar sensor noise variance in lab 2. The results of these calculations can be found in Table 2 in section 5 "Results and Discussion".

We also compared the noise distribution of the sonar sensor to the different statistical distributions, such as Gaussian, Student's T and Gamma distributions to find out the most accurate representation of the noise. We observed that a Gaussian (as used in the standard implementation) corresponded best to the sensor data noise.

By calculating all these variances, we found out that the initial estimates for the GPS noise variances in x and y directions were too low by orders of magnitude. When we adjusted these parameters, we saw that the arrows for the estimated (x, y) position of the drone jumped around a lot less in the simulation and basically appeared to be right underneath the drone all the time.

4 Methodology

In this section we explain the experimental setup and parameter tuning methodologies.

4.1 Experimental Setup

Simulation-Based Testing

Similarly to project 1, we used the maps from Lab 1 that we created utilizing Gazebo and RViz to simulate navigation in a controlled virtual environment. Briefly summarized, we mapped a simulated world using Cartographer and tele-operated the virtual TurtleBot to generate an occupancy grid in Lab 1. This map was then used in Project 2 to evaluate and fine-tune our estimation and control algorithms for a quadcopter to fly over a set of waypoints. By iterating within the simulation, we adjusted parameters and verified the controller's ability to follow planned paths efficiently.

As a first step we tested the controller node with setting `ground_truth = true` for the behavior and the estimator nodes. The reason for this being that testing and tuning the estimator by observing simulation runs would be impossible with a controller that does not work. When we had a working controller we could start tuning parameters.

4.2 Define Performance Metrics

To tune our parameters efficiently during the simulation-based testing, we had to define the metrics that determine the performance of our algorithms. These include:

- **Accuracy:** The controller and estimator must ensure that the quadcopter follows the planned trajectory closely and reaches the intended waypoints with minimal position error.
- **Robustness and Repeatability:** The system must work reliably across multiple runs.
- **Smoothness:** We visually evaluated the motion of the drone to avoid jerky or oscillatory behavior. This was particularly relevant when tuning the low-pass filter and derivative terms in the controller.
- **Speed / Cycle Count:** We aimed to maximize the number of times the drone could complete the full waypoint path within the fixed time limit of the simulation.
- **Tracking Behavior:** While the drone must follow the TurtleBot, it should not "hover behind" for extended periods. The controller needed to be aggressive enough to reach the TurtleBot fast enough, while maintaining stable flight.

These goals were evaluated using a mix of visual inspection in RViz and Gazebo, as well as logging and analyzing relevant telemetry data. We manually adjusted control gains (e.g., K_p, K_d, and low-pass filter factor α) and observed the resulting behavior in the simulation.

In Table 1 there is an overview over all of our final controller parameters.

Kalman Filter Tuning Methodology

The main thing we did to tune the Kalman filter was to calculate the noise variances of all the sensors that are used in our pipeline and to adjust the noise covariance matrices accordingly. The method was already extensively described in section 3.2.2 and the noise variances are summarized in Table 2.

Parameter	Value
`lookahead_distance`	1.0
`max_xy_vel`	2.0
`max_z_vel`	1.0
`yaw_vel`	0.3
`kp_xy`	0.8
`kp_z`	0.5
`kd_xy`	0.4
`kd_z`	0.2
`alpha`	0.3

Table 1: Controller Parameters

5 Results and Discussion

As mentioned in section 3.2.2., our results for the sensor noise calculations can be seen in Table 2.

When we used the exact values that we calculated, we noticed that by using the larger GPS noise variances, the drone's estimated position started lagging behind the drone very substantially. One likely explanation is that the estimator does not use the IMU's directly measured orientation but instead integrates angular velocity to obtain yaw. Double integrating noisy measurements can lead to substantial lag over time. Because the variance of the IMU is much smaller than the variance of the GPS, the estimator trusts the measurements of the IMU much more than the measurements of the GPS. A solution to this problem was found by lowering the variance of the GPS or by making the variance of the IMU higher. Both will achieve similar results by making GPS data more trustworthy. With this, the measurements of the GPS will lead to a correction in the drift and the lag.

With the GPS variances `var_gps_x` and `var_gps_y` decreased to 10^{-7}, the lag is less, but the estimation is jumping around quite a bit. With the low-pass filter this problem could be diminished. Another test with the variance of the IMU `var_imu_x` and `var_imu_y` increased to 10^{-1} also reduced the lag but the controller gain `kp_xy` had to be decreased to 0.6 and could not be as aggressive as before. The accuracy in this test was better. However, in the end, we still decided to decrease the GPS variance because this approach was more reliable, the drone was faster and the negative effects of the noisy estimation got minimized well by the low-pass filter.

ROS Parameters	Variance
var_imu_x	5.47745e-05
var_imu_y	1.44979e-05
var_imu_z	1.43154
var_imu_a	5.95763e-10
var_gps_x	3.98755e-02
var_gps_y	3.95451e-02
var_gps_z	3.79362e-02
var_baro	2.64803e-02
var_sonar	5.72784e-04
var_magnet	7.06693e-04

Table 2: Sensor Noise Variances

6 Conclusion and Recommendations

In this project, we successfully designed and implemented controller and estimator plugins to enable a quadcopter to navigate along waypoints over an obstacle-filled environment in simulation. To do so, we used an improved pure pursuit algorithm and a Kalman filter to predict and correct the drone's estimated position.

The pure pursuit algorithm introduces noticeable improvements over regular pure pursuit by incorporating a derivative term for damping and a low-pass filter for velocity command smoothing. These additions allow us to use higher proportional gains for faster convergence without inducing oscillations or choppy motions, resulting in smoother, more stable path tracking. This algorithm works very well in simulation.

The Kalman filter provides robust state estimation. Through iterative tuning of the noise covariances and by using a χ^2-Gating method to reject measurement outliers, the filter achieves rapid convergence to the true state while effectively rejecting sensor noise.

Overall, the project was successful, leaving room for further experiments to increase the performance of our algorithms even more. Next steps could include:

- **Sensor fusion resp. Cross-Sensor Covariance Handling:** Address the assumption of uncorrelated sensor measurements in the standard Kalman filter. Sonar, barometer, and other altitude sensors often provide overlapping information with different biases and noise characteristics. By explicitly modeling their cross-sensor covariances in the

measurement covariance matrix R, we could prevent conflicting updates.

- **Height-Aware Goal Adjustment:** Incorporate a precomputed height map of the environment so that sonar measurements can be weighted appropriately. For example, when flying over a three-meter wall, the goal altitude could be dynamically reduced to two meters instead of using a fixed five-meter clearance, reducing unnecessary climb maneuvers. This way we can rely more on the sonar measurements again and do not have to reject as many measurements as we currently are.
- **Real-World Testing:** Test the algorithms on real hardware in a physical environment to see how robust our algorithms that we developed in simulation are.