Planar Quadrotor:

Extended Kalman Filter Design

Arseni Lysak, Tsimafei Iliusenka, John Elvin Ndahiro, Pablo Calderon Mateo

Aeronautical Systems Integration

January 19, 2026

Introduction

The aim of this project is to implement an Extended Kalman Filter (EKF) to estimate the movement of a 2D planar quadrotor (fig. 1) utilizing noisy sensor measurements.

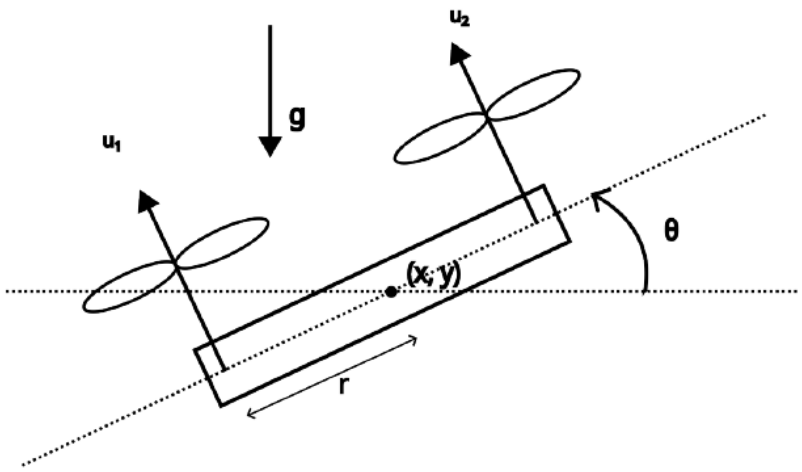

Figure 1. Diagram of the Planar Quadrotor System.

System modeling

The movement of the quadrotor is governed by the following nonlinear equations of motion:

$$m\ddot{x} = -(u_1 + u_2)\sin\theta \tag{1}$$

$$m\ddot{y} = (u_1 + u_2)\cos\theta - mg \tag{2}$$

$$I\ddot{\theta} = r(u_1 - u_2) \tag{3}$$

where:

- m: mass of the quadrotor
- I: moment of inertia
- r: distance from the center of mass to the rotors
- g: gravitational acceleration
- u_1, u_2: rotor thrust forces

This would imply that the system state is represented by a six-element vector $\mathbf{x}$, and the control input is represented by a two-element vector $\mathbf{u}$:

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ \theta \\ \dot{\theta} \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{4}$$

Quadcopter definition

The quadrotor system parameters were initialized with a mass $m = 0.5$ kg, a rotor arm length $r = 0.15$ m, and a moment of inertia $I = 0.005$ kg $\cdot$ m^2. The simulation utilizes a time step of $dt = 0.01$ s under standard gravitational acceleration $g = 9.81$ m/s^{2}2. The initial conditions for the simulation were set with an orientation angle $\theta = 0$ rad, and baseline rotor thrust forces $u_1 = 5$ N and $u_2 = 5$ N.

Four scenarios for control inputs were chosen:

1. Flying up. For the simplest case, both rotors are set to 3 N.
2. Recovery from horizontal flight. To do this, quadrotor system must rotate to have significant vertical thrust component. Three control stages are:
    a) Initiate rotation. For 1.1 s create rotational moment by 0.05 N thrust difference between the rotors (4.05 N, 4.0 N).
    b) Stop the rotation. For the next 1.1 s we slow down the rotation with the opposing thrust moment (4.0 N, 4.05 N).
    c) Increase altitude. For the rest of the simulation time we pull up with same thrust of 4 N on both rotors.
3. 360 degree roll. Most complicated control-wise, due to the choice of long rotation. Altitude loss is significant. Four control stages are:
    a) Initiate rotation. A small thrust difference is applied for the first 2 seconds (4.04 N, 4 N).
    b) Let it rotate. For a second, the system is set to 0 thrust to let it rotate freely.
    c) Stop the rotation. For another 2 seconds, we slow down the rotation, only using small thrust to avoid increasing falling speed (0 N, 0.04 N).
    d) Recover lost altitude. After 5 seconds of accurate controls, we set both rotors to their maximum (5 N) to avoid collision with the ground.
4. Straight fall recovery. Almost identical to the first case, however slightly higher thrust is assumed (3.2 N on both rotors).

Small thrust variation of the rotor is inherently periodic and is recreated using simple trigonometric functions (cosine for rotor u1, sine for rotor u2).

Moreover, a unique set of initial conditions was defined for each case:

1. Zero velocity in both directions, system is 1 m above the ground, no rotation.
2. System is assumed to have horizontal speed of 3 m/s at 10 m altitude, rotors' thrust is parallel to the ground.
3. To make the long rotation possible, high vertical speed and altitude (5 m/s and 50 m respectively) are assumed. No initial rotational speed or system tilt.
4. Negative horizontal and vertical velocities (-1 m/s and -3 m/s respectively), altitude of 15 m. Quadrotor system is initially tilted down by around 18 degrees to align rotor thrust direction with total velocity vector.

Clean simulation

Equations 1-3 represent perfect conditions as they do not account for process noise. The state vector for this ideal trajectory is designated as `state.clean` in the code. This ensures the prediction adheres to the actual physical dynamics.

The clean simulation utilizes first-order Euler integration to propagate the system state forward in time. At each time step t, the state increments are computed based on the previous state vector $\mathbf{x}_{t-1}$ and the current control inputs. The positional and angular updates are derived linearly from the previous velocities:

$$\Delta x = \dot{x}_{t-1}\Delta t, \quad \Delta y = \dot{y}_{t-1}\Delta t, \quad \Delta\theta = \dot{\theta}_{t-1}\Delta t \tag{5}$$

The changes in linear and angular velocities are calculated by discretizing the dynamic equations of motion (Eq.1-3). These updates account for the non-linear coupling introduced by the quadrotor's orientation θ:

$$\Delta\dot{x} = -\frac{(u_1 + u_2)}{m}\sin(\theta_{t-1})\Delta t \tag{6}$$

$$\Delta\dot{y} = \left(\frac{(u_1 + u_2)}{m}\cos(\theta_{t-1}) - g\right)\Delta t \tag{7}$$

$$\Delta\dot{\theta} = \frac{r}{I}(u_1 - u_2)\Delta t \tag{8}$$

Finally, the state vector is updated by adding these increments to the previous state, $\mathbf{x}_t = \mathbf{x}_{t-1} + \Delta\mathbf{x}$, and the ideal sensor output is computed as $\mathbf{y}_t = C\mathbf{x}_t$.

Real-World Trajectory

The real-world trajectory (`state.real`) differs from the perfect trajectory solely through the introduction of process and measurement noise. It is assumed that both process and measurement noise follow a **Gaussian** distribution, which is implemented in the simulation using the MATLAB function `randn`. The stochastic properties are defined by their variances: the process noise variance is set to 0.003^2, while the measurement noise variance is set to 0.01^2.

The governing equations 1-3 remain unchanged, with noise injected during the state update step. The true state evolution is modeled by adding a stochastic process noise

vector $\mathbf{w}$ to the deterministic dynamics:

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \Delta\mathbf{x} + \mathbf{w}_{t-1} \tag{9}$$

Subsequently, the simulated sensor measurements $\mathbf{z}_t$ are generated by applying the measurement matrix C to the updated state, with the addition of measurement noise $\mathbf{v}_t$:

$$\mathbf{z}_t = C\mathbf{x}_t + \mathbf{v}_t \tag{10}$$

Extended Kalman Filter Design

The Linear Kalman filter addresses the general problem of trying to estimate the state of a discrete-time controlled process that is governed by a linear stochastic difference equation. A Kalman filter that linearizes about the current mean and covariance is referred to as an extended Kalman filter or EKF.

The EKF is employed because the quadrotor system is nonlinear—motion is dependent on the sine and cosine of the angle θ. The EKF linearizes the system at each time step using the current optimal estimate prior to applying standard Kalman update equations.

The working principle of the filter is optimally weighting sensor measurements against the predicted values in accordance with the ideal mathematical model (fig. 2).

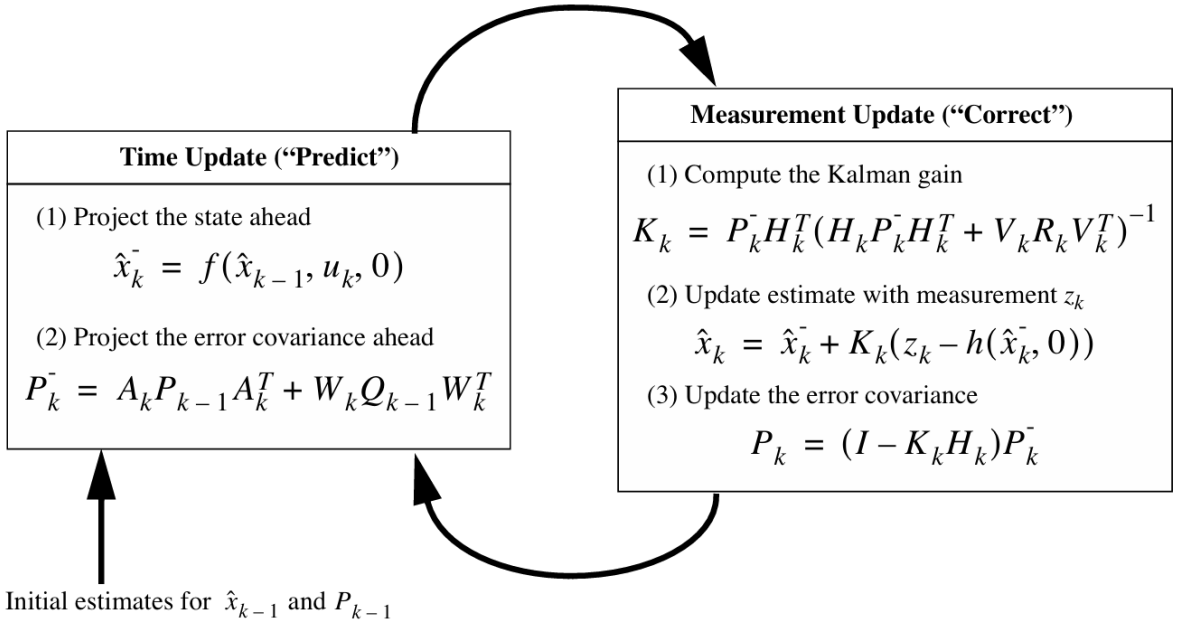

Figure 2. A complete picture of the operation of the extended Kalman filter. Source: Greg Welch and Gary Bishop, *An Introduction to the Kalman Filter*, Technical Report TR 95-041, Updated March 11, 2002 (University of North Carolina at Chapel Hill, 2002).

Jacobian for Covariance Propagation

To estimate a process with non-linear difference and measurement relationships,the system is modeled using non-linear functions f and h:[1]

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{w}_k \tag{11}$$

$$\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k \tag{12}$$

where:

- $\mathbf{x}_k$ is the state vector at time step k,
- $\mathbf{z}_k$ is the measurement vector,

1. Greg Welch and Gary Bishop, *An Introduction to the Kalman Filter*, Technical Report TR 95-041, Updated March 11, 2002 (University of North Carolina at Chapel Hill, 2002).

- $\mathbf{u}_k$ is the control input vector,
- $\mathbf{w}_k$ and $\mathbf{v}_k$ are the process and measurement noise vectors.

To linearize the estimate for covariance propagation, the Jacobian matrices $\mathbf{F}$ and $\mathbf{H}$ have to be computed:

- $\mathbf{F}_k$ is the Jacobian of the dynamics function f with respect to the state $\mathbf{x}$, evaluated at the previous estimate $\hat{\mathbf{x}}_{k-1|k-1}$: $[\mathbf{F}_k = \frac{\partial f}{\partial \mathbf{x}}\big|\, \hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k]$
- $\mathbf{H}_k$ is the Jacobian of the measurement function h with respect to the state $\mathbf{x}$, evaluated at the predicted state $\hat{\mathbf{x}}_{k|k-1}$: $[\mathbf{H}_k = \frac{\partial h}{\partial \mathbf{x}}\big|\, \hat{\mathbf{x}}_{k|k-1}]$

To update the error covariance, the state transition Jacobian A is derived from the partial derivatives of the dynamics, which are defined by equations 1 - 3:

$$\frac{\partial \ddot{x}}{\partial \theta} = -\frac{\cos\theta}{m}(u_1 + u_2) \tag{13}$$

$$\frac{\partial \ddot{y}}{\partial \theta} = -\frac{\sin\theta}{m}(u_1 + u_2) \tag{14}$$

The resulting Jacobian A is:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{-\cos(\theta)(u_1+u_2)}{m} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{-\sin(\theta)(u_1+u_2)}{m} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{15}$$

The discrete-time state transition matrix F used in the EKF is defined as $F = I + A\Delta t$, where I represents the identity matrix and $A\Delta t$ represents the change over the time step.

$$F = \begin{bmatrix} 1 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \frac{-\cos(\theta)(u_1+u_2)}{m}\Delta t & 0 \\ 0 & 0 & 1 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 1 & \frac{-\sin(\theta)(u_1+u_2)}{m}\Delta t & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{16}$$

The measurement matrix H (referred to as C in code) is defined as:

$$H = C = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{17}$$

The implementation follows a standard **predict–correct cycle** (fig. 2), executing in real-time within the simulation loop.

Prediction Step

First, the states are updated, and the estimate is stored as `state.estimate`.

Following the state acquisition for the current timestep, the F matrix is updated based on the current angle (`theta_estimate`) and thrust (`control_input`). This

dependency characterizes the Extended Kalman Filter. The error covariance is then predicted:

$$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \tag{18}$$

where Q represents the confidence in the physics model.

Correction Step

The estimate is subsequently corrected using the latest noisy sensor data (`output.real`). The measurement residual—the discrepancy between sensor readings and predictions—is computed:

The measurement residual, also known as the innovation, quantifies the discrepancy between the actual sensor measurements and the predicted output based on the a priori state estimate. It is computed as:

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - C\hat{\mathbf{x}}_k^- \tag{19}$$

where $\mathbf{z}_k$ represents the noisy sensor measurement vector at time step k, and $C\hat{\mathbf{x}}_k^-$ represents the predicted measurement derived from the current state estimate.

The Kalman gain K is calculated to determine the weighting between sensor data and physics predictions:

$$S = CP_k^- C^T + R \tag{20}$$

$$K = P_k^- C^T S^{-1} \tag{21}$$

Process noise covariance Q and measurement noise covariance R were defined as $I \times 0.003^2$ and $I \times 0.01^2$, respectively. This effectively means that the filter is "perfectly tuned."

Finally, the state and covariance matrices are updated to obtain the a posteriori estimates. The state estimate is corrected by adding the measurement residual, weighted by the Kalman gain K:

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + K_k \tilde{\mathbf{y}}_k \tag{22}$$

Subsequently, the error covariance matrix P is updated to reflect the reduced uncertainty following the incorporation of the measurement:

$$P_k = (I - K_k C) P_k^- \tag{23}$$

Running Mean Filter

In order to compare the performance of the EKF, the running mean filter was implemented. With the window size of 10, it simply averages the values from the last 10 time steps when processing sensor measurements. It then uses this data, to predict the change in states.

The running mean filter implementation employs a hybrid estimation strategy. First, a sliding window average is applied to the noisy sensor measurements to attenuate high-frequency jitters. For a window size N, the smoothed measurement vector $\bar{\mathbf{z}}_t$ is computed as the arithmetic mean of the past N observations:

$$\bar{\mathbf{z}}_t = \frac{1}{N} \sum_{k=t-N+1}^{t} \mathbf{z}_k \tag{24}$$

These smoothed measurements are directly substituted into the state vector for the observable variables (y, θ, and $\dot{\theta}$), bypassing the dynamic model for these specific degrees of freedom. For the remaining unobserved states (x, $\dot{x}$, and $\dot{y}$), the filter utilizes the system dynamics (Eq. 1-2) for propagation. Crucially, these dynamic updates rely on the smoothed orientation angle θ_t rather than the previous state estimate. The horizontal position is updated kinematically, while the velocity increments are derived from the thrust projections:

$$\Delta x = \dot{x}_{t-1}\Delta t \tag{25}$$

$$\Delta \dot{x} = -\frac{(u_1 + u_2)}{m}\sin(\theta_t)\Delta t \tag{26}$$

$$\Delta \dot{y} = \left(\frac{(u_1 + u_2)}{m}\cos(\theta_t) - g\right)\Delta t \tag{27}$$

Finally, these computed increments are applied to update the estimates for the horizontal position and linear velocities.

Results & Visualization

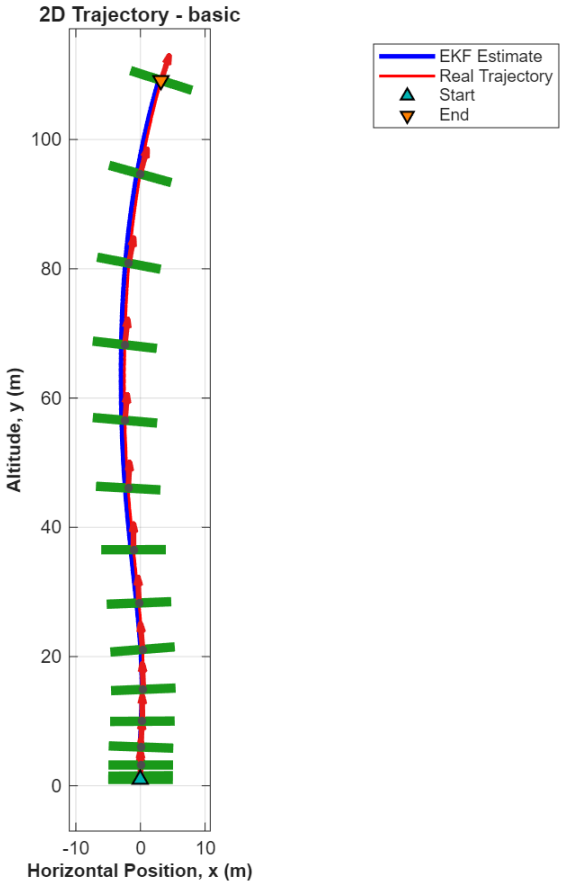

Figure 3. 2D trajectory for the basic hover scenario, comparing EKF estimate against the real noisy path.

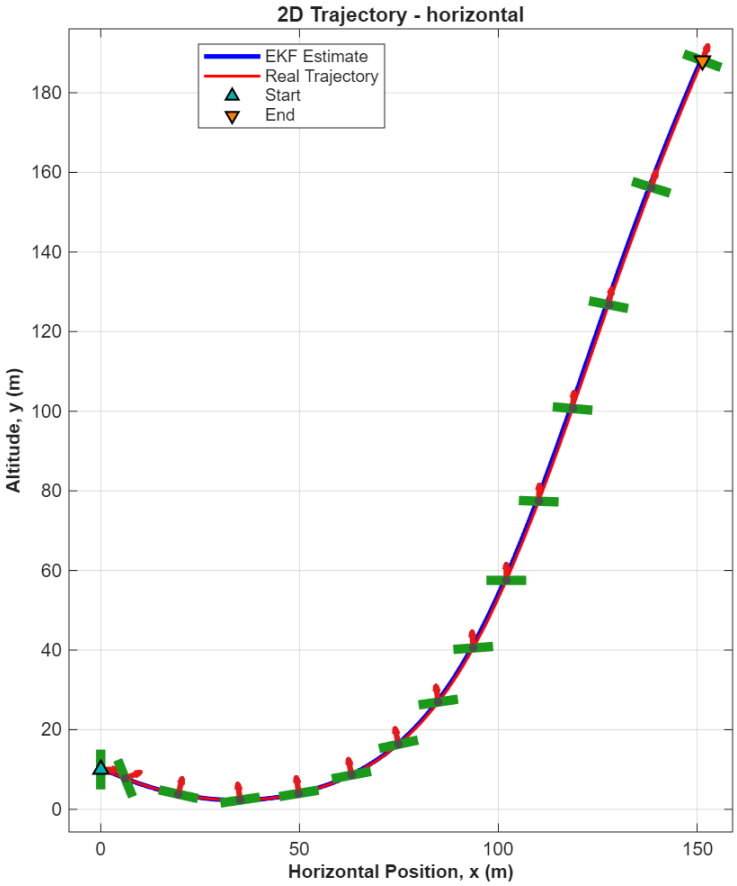

Figure 4. 2D trajectory for the horizontal recovery scenario, showing EKF performance during lateral motion.

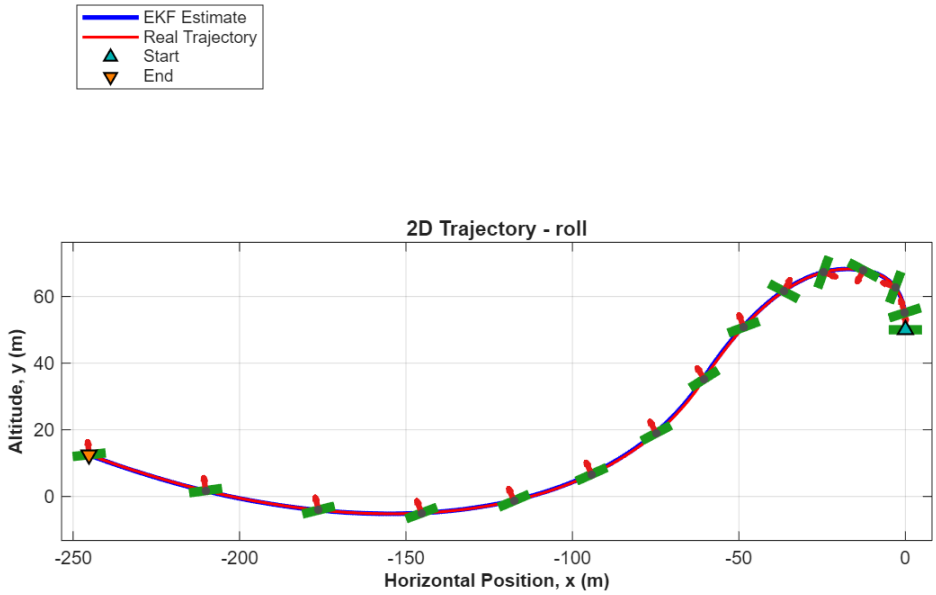

Figure 5. 2D trajectory for the 360° roll scenario, illustrating EKF tracking during aggressive rotational motion.

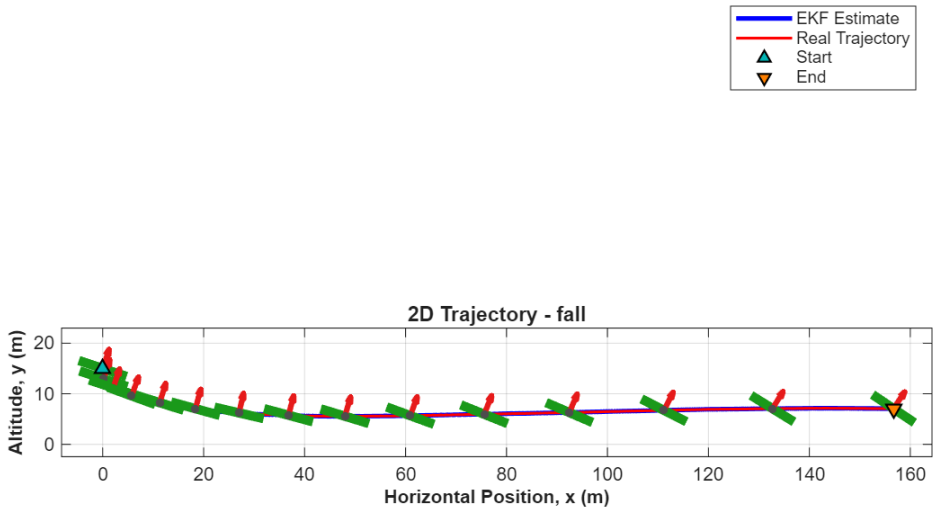

Figure 6. 2D trajectory for the straight fall recovery scenario, demonstrating EKF accuracy during descent.

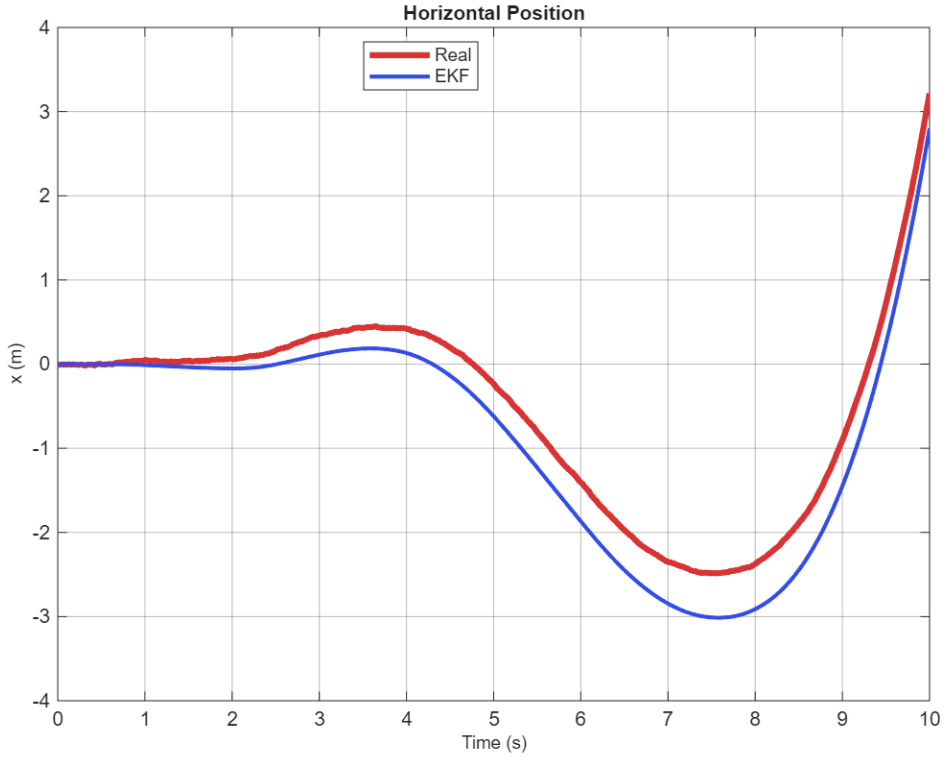

Figure 7. Horizontal position (x) over time for the basic scenario, where the small lateral displacement is caused by slight thrust asymmetry that induces a small rotation of the vehicle.

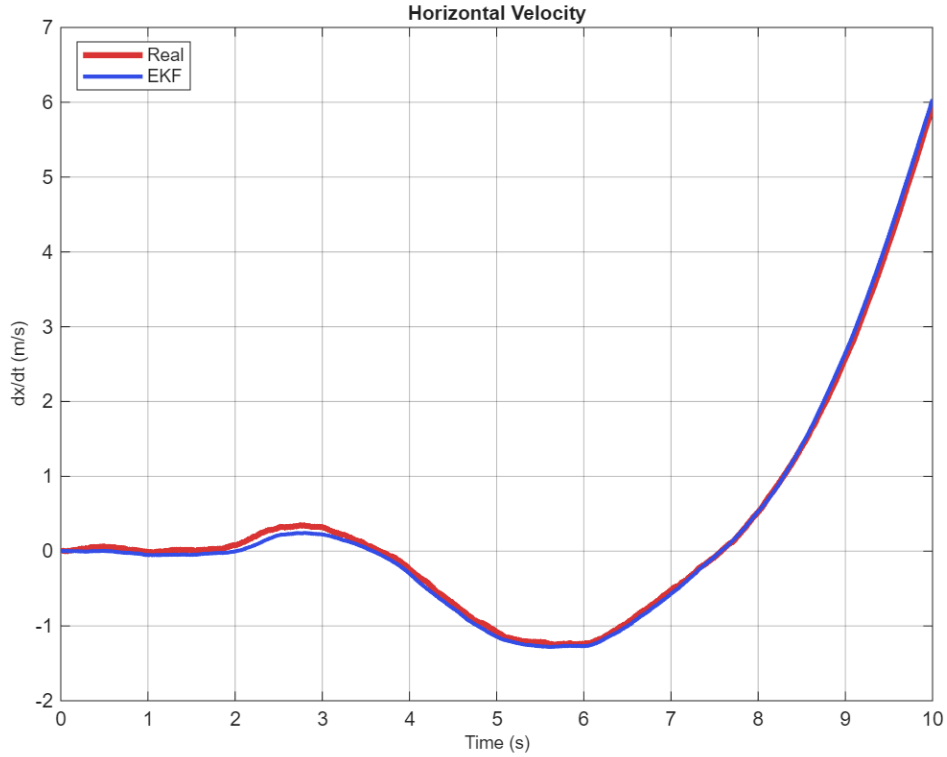

Figure 8. Horizontal velocity ($\dot{x}$) over time for the basic scenario, oscillating around zero and indicating minor alternating horizontal accelerations.

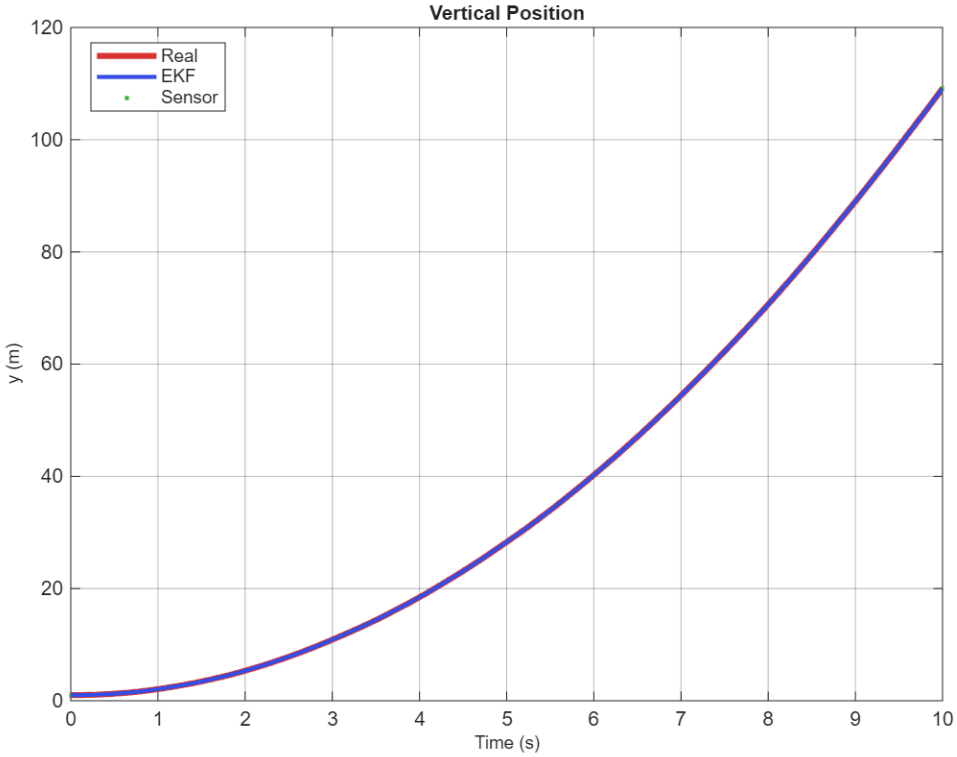

Figure 9. Vertical position (y) over time for the basic scenario, showing continuous ascent with no reversal in direction.

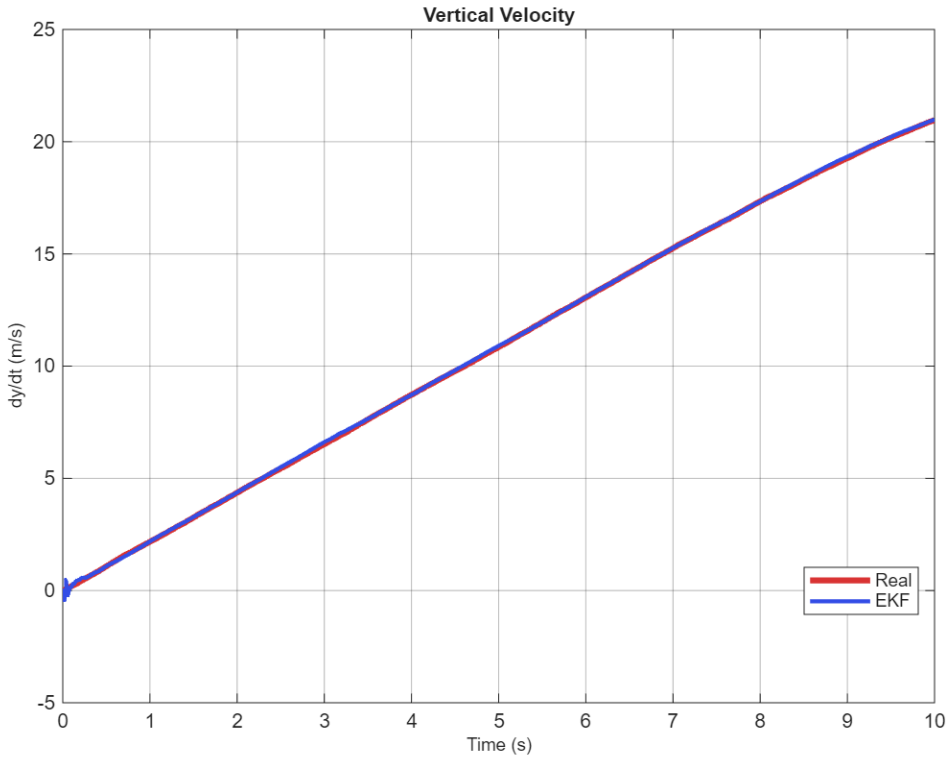

Figure 10. Vertical velocity ($\dot{y}$) over time for the basic scenario, remaining positive and increasing smoothly, confirming upward motion.

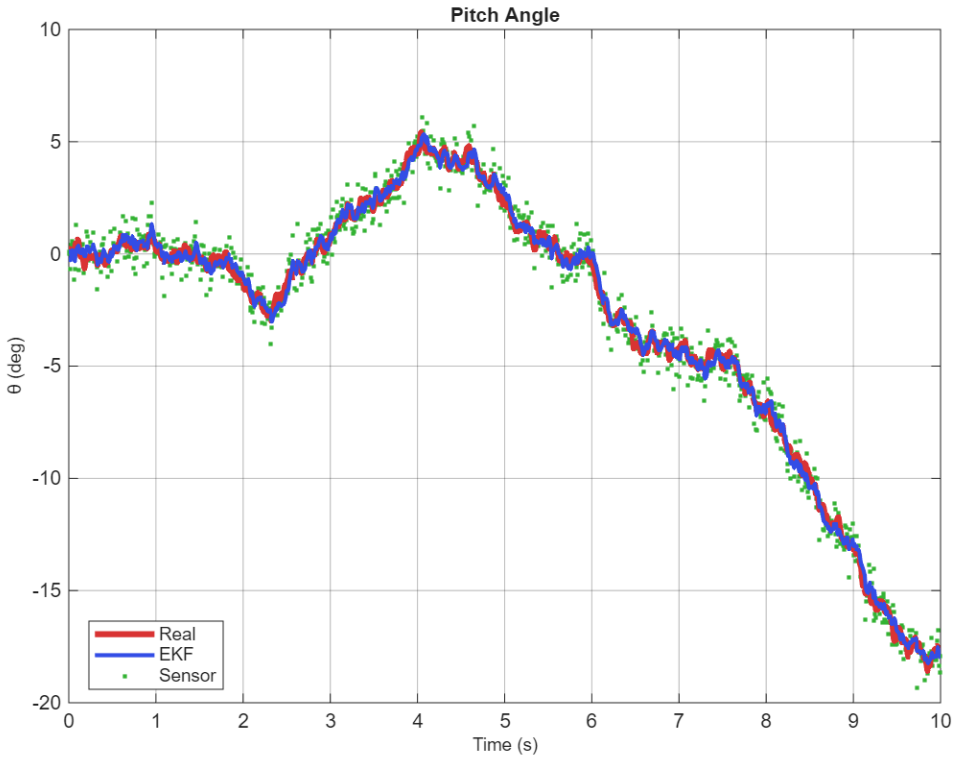

Figure 11. Pitch angle (θ) over time for the basic scenario, where positive and negative values represent alternating nose-up and nose-down orientations.

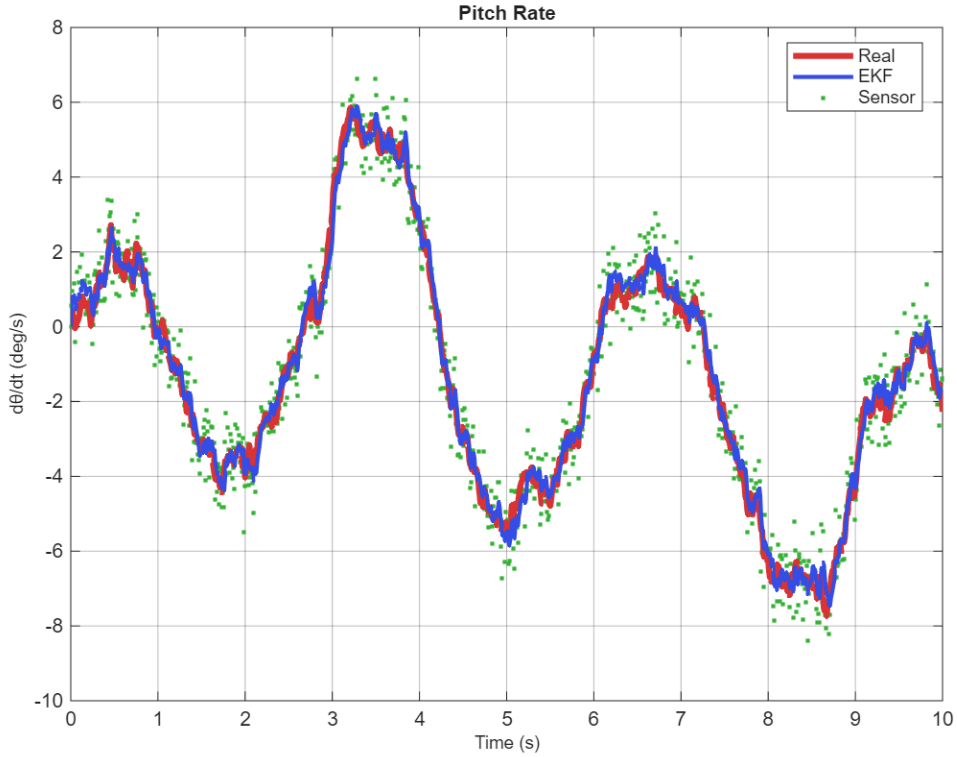

Figure 12. Pitch rate ($\dot{\theta}$) over time for the basic scenario, with zero crossings corresponding to changes in rotational direction.

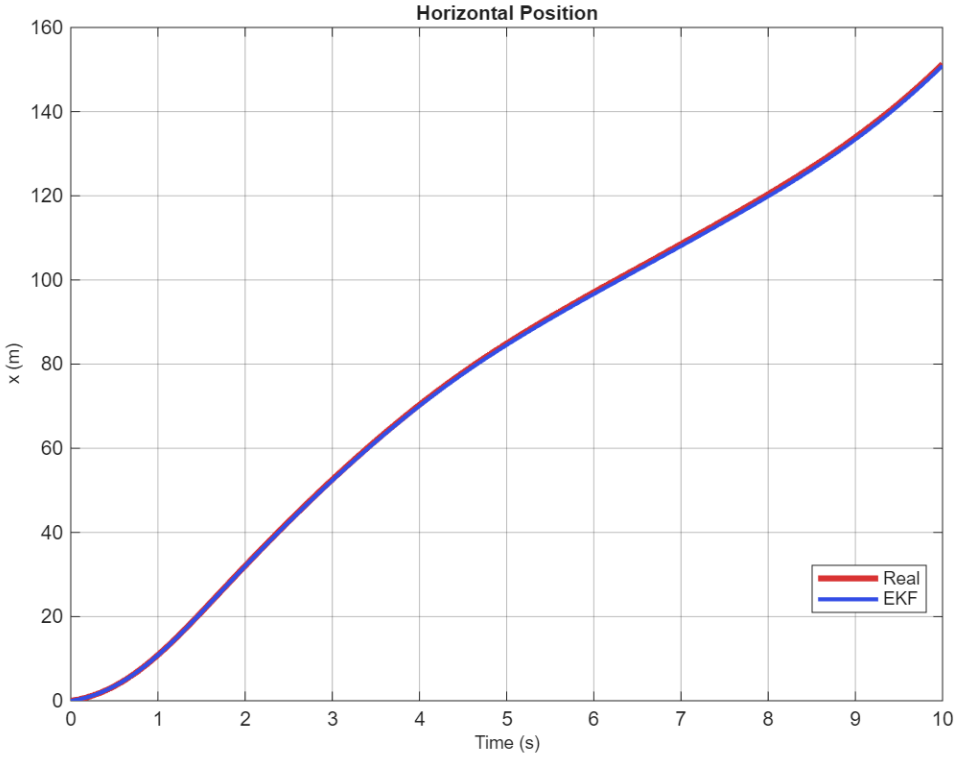

Figure 13. Horizontal position (x) over time for the horizontal scenario, showing sustained lateral movement in a single direction.

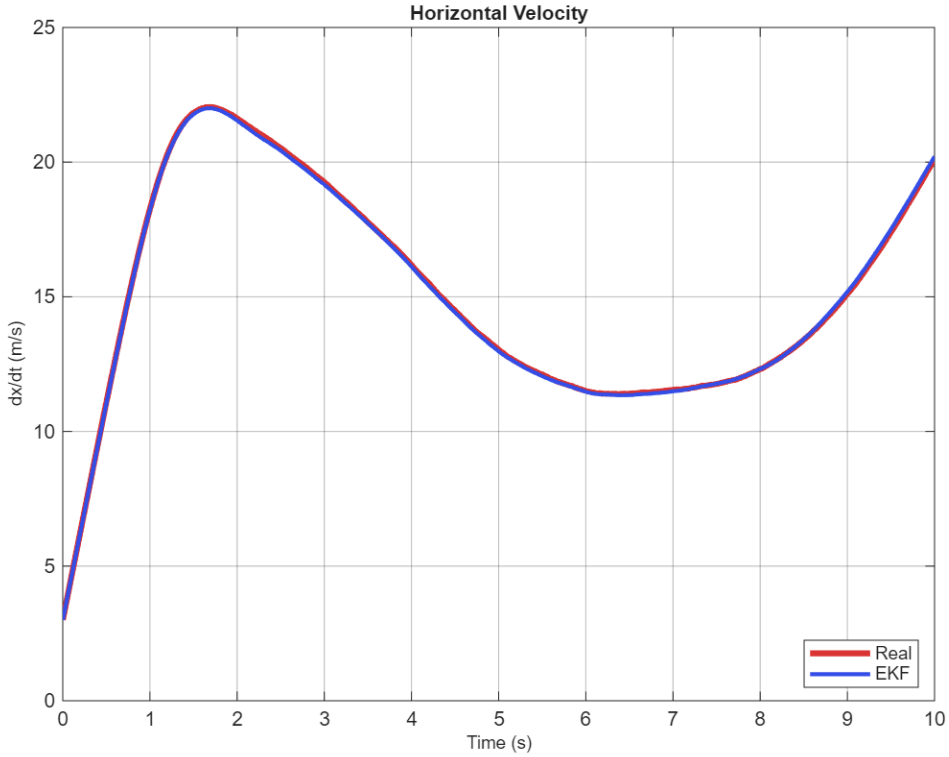

Figure 14. Horizontal velocity ($\dot{x}$) over time for the horizontal scenario, where variations in slope indicate acceleration and deceleration phases.

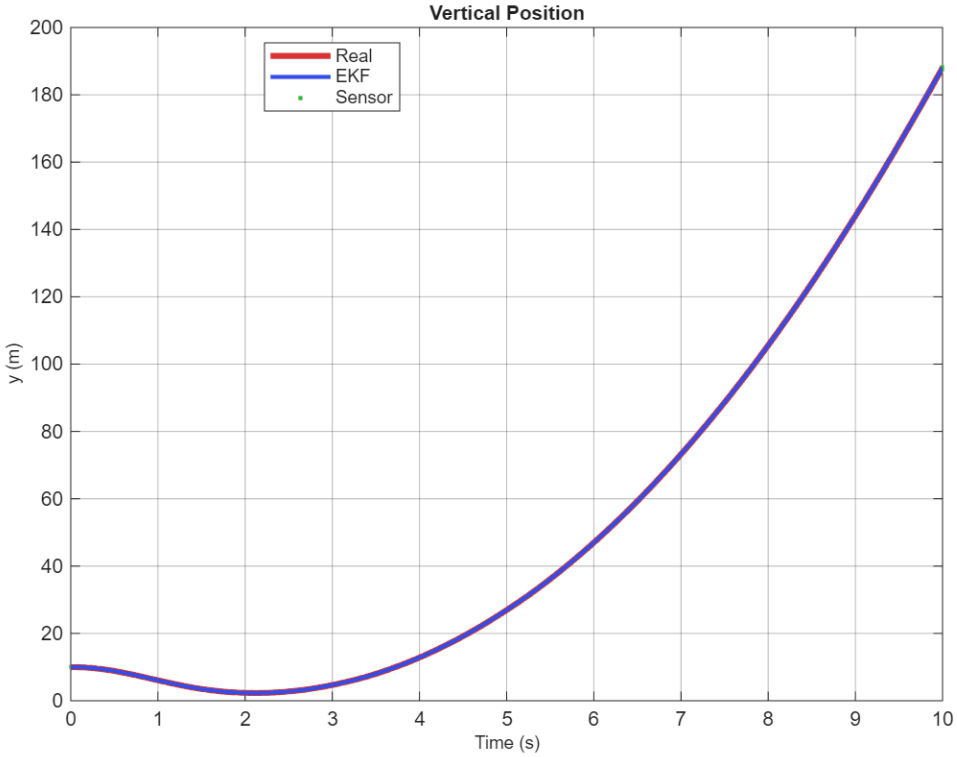

Figure 15. Vertical position (y) over time for the horizontal recovery scenario, decreasing initially then increasing

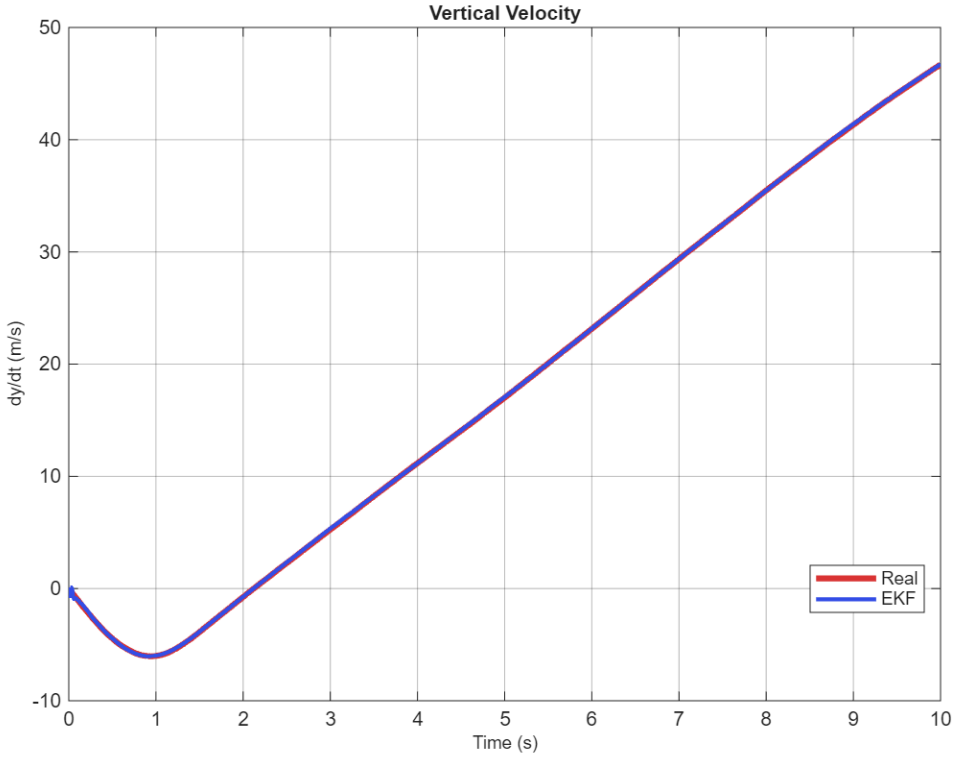

Figure 16. Vertical velocity ($\dot{y}$) over time for the horizontal recovery scenario, where negative values indicate descent and the zero crossing marks direction reversal.

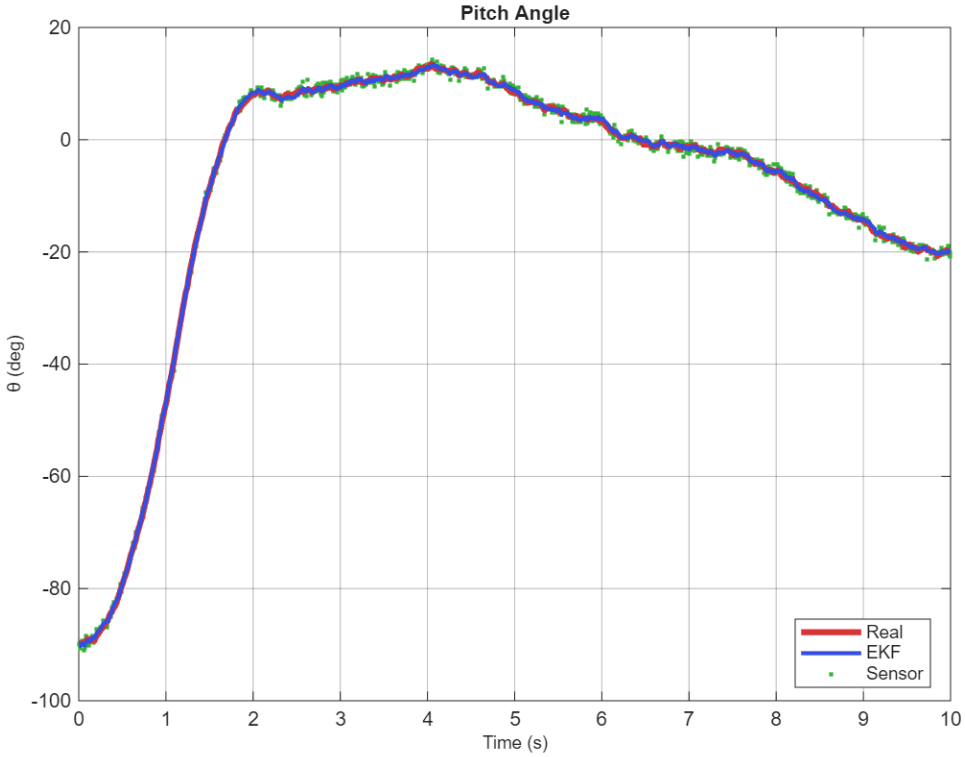

Figure 17. Pitch angle (θ) over time for the horizontal recovery scenario, showing rapid attitude change followed by gradual stabilization.

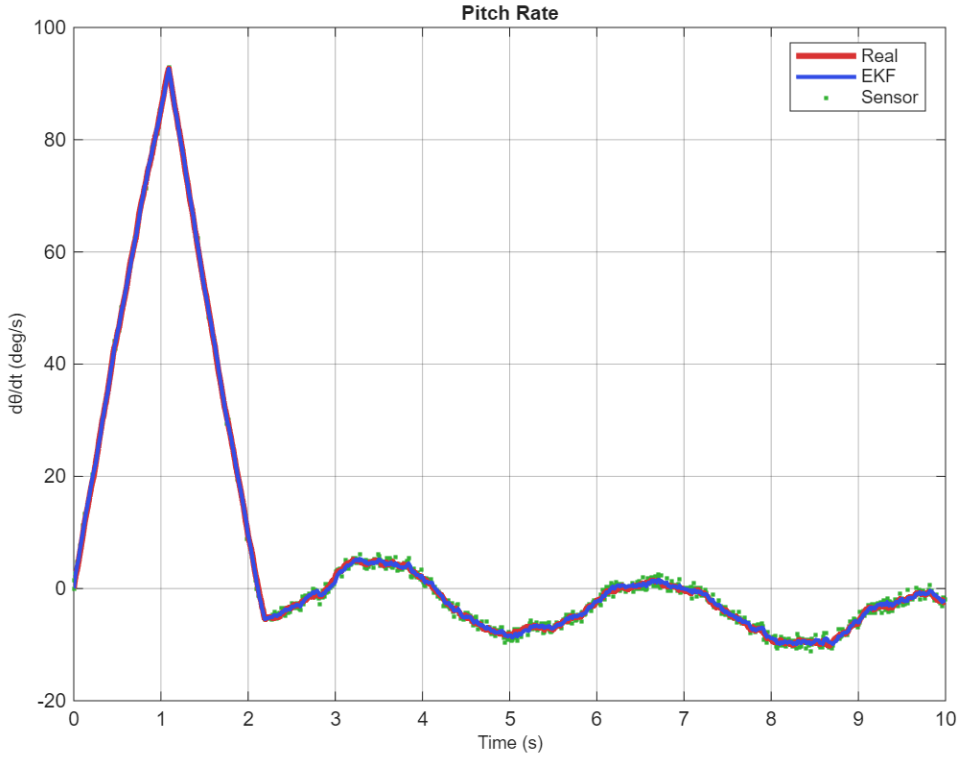

Figure 18. Pitch rate ($\dot{\theta}$) over time for the horizontal recovery scenario, characterized by high initial rotation rates that later reduce in magnitude.

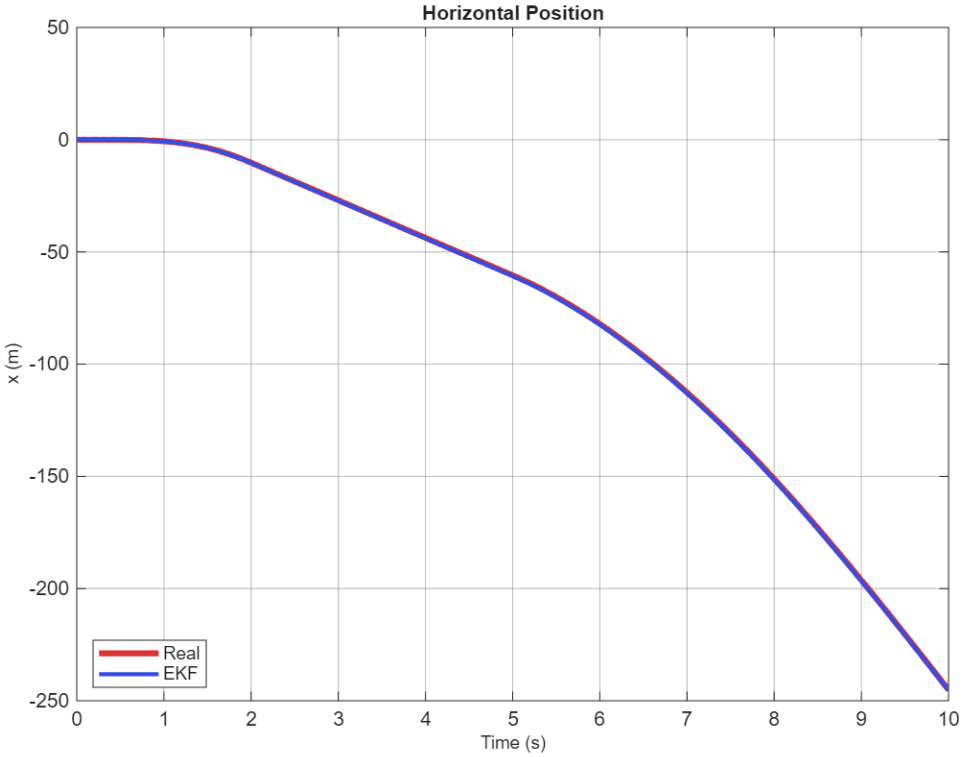

Figure 19. Horizontal position (x) over time for the roll scenario, decreasing continuously, indicating motion opposite to the reference direction.

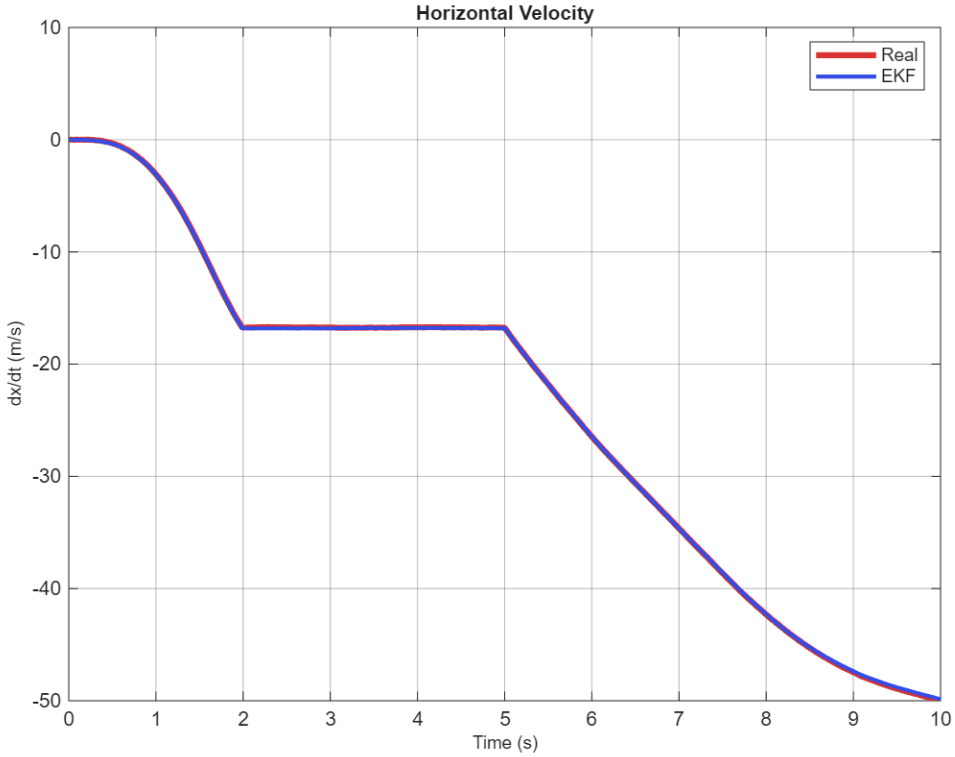

Figure 20. Horizontal velocity ($\dot{x}$) over time for the roll scenario, remaining negative throughout, confirming persistent backward motion.

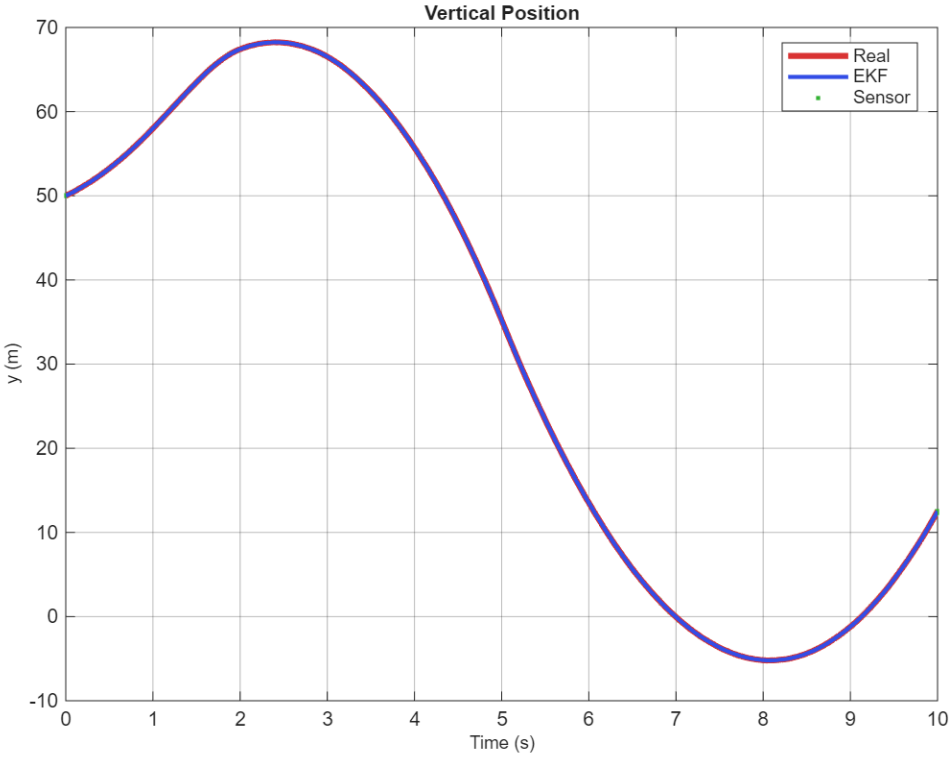

Figure 21. Vertical position (y) over time for the roll scenario, showing alternating ascent and descent with clear extrema.

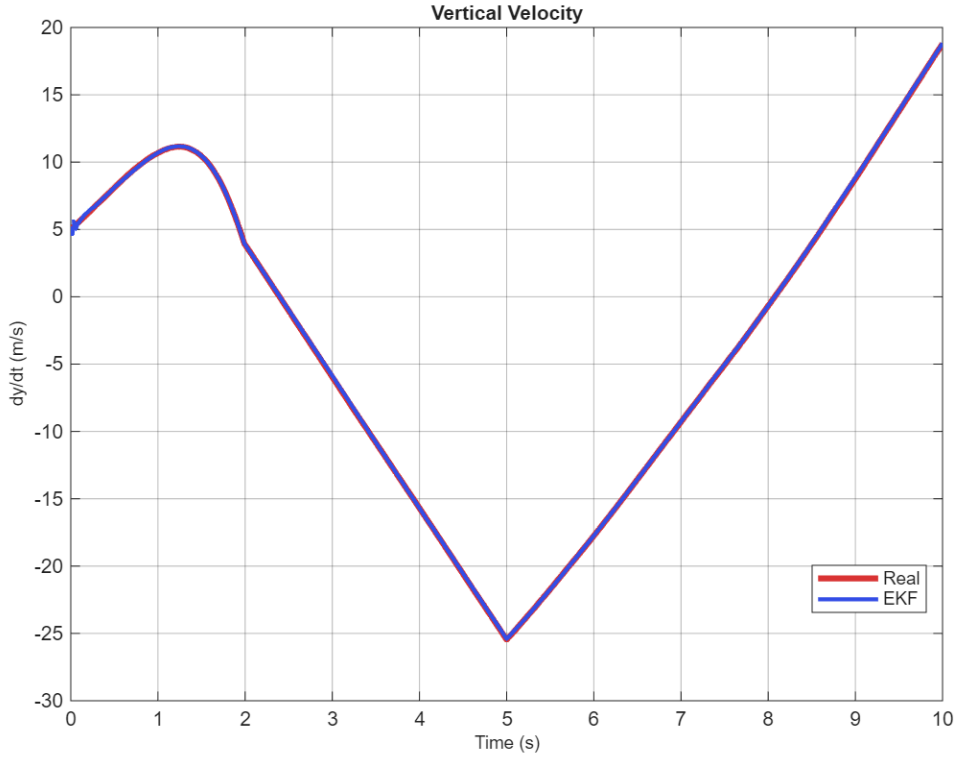

Figure 22. Vertical velocity ($\dot{y}$) over time for the roll scenario, where multiple zero crossings indicate repeated reversals in vertical motion.

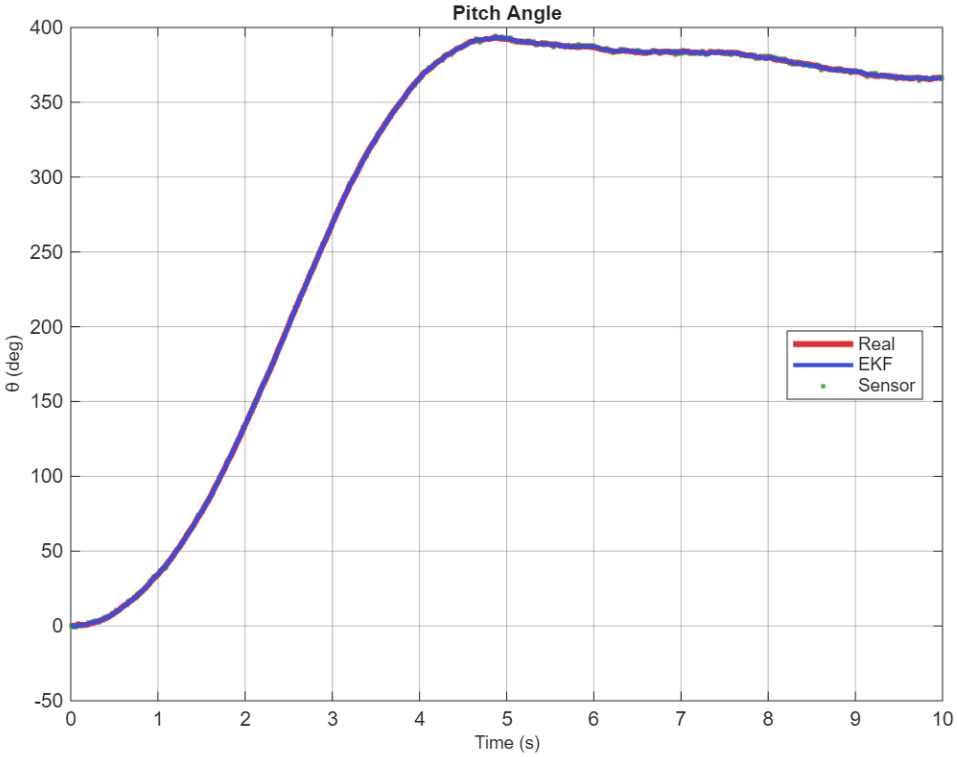

Figure 23. Pitch angle (θ) over time for the roll scenario, exceeding 360°, confirming completion of a full rotational maneuver.

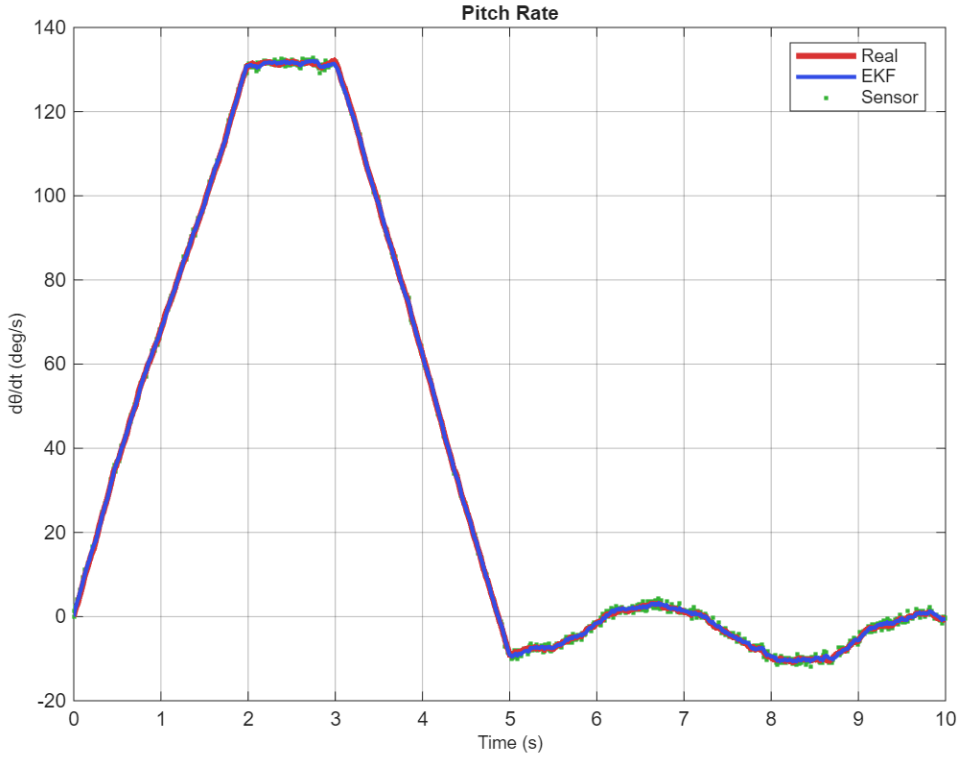

Figure 24. Pitch rate ($\dot{\theta}$) over time for the roll scenario, with large positive values indicating rapid rotation and negative values marking deceleration.

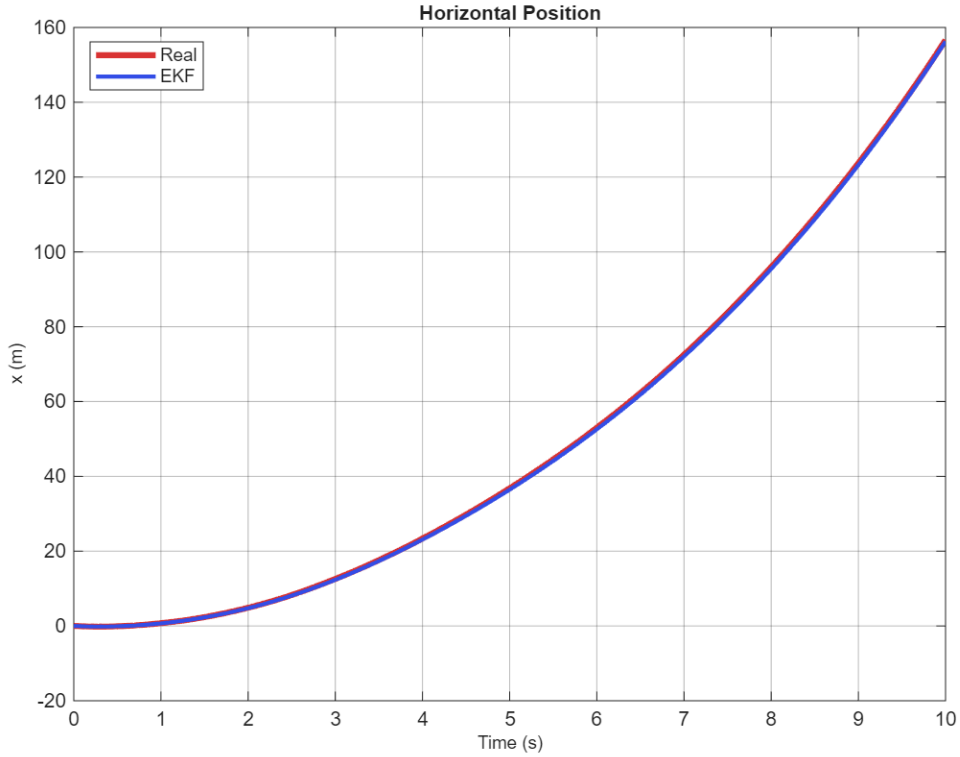

Figure 25. Horizontal position (x) over time for the fall scenario, showing continuous forward motion during descent.

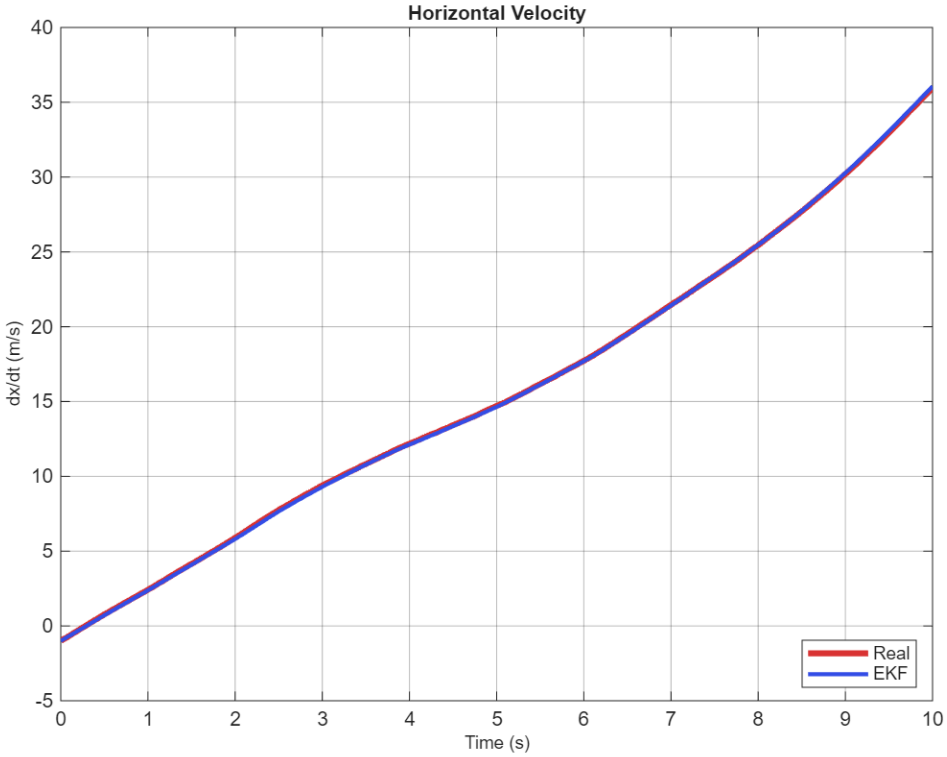

Figure 26. Horizontal velocity ($\dot{x}$) over time for the fall scenario, increasing almost linearly, indicating constant horizontal acceleration.

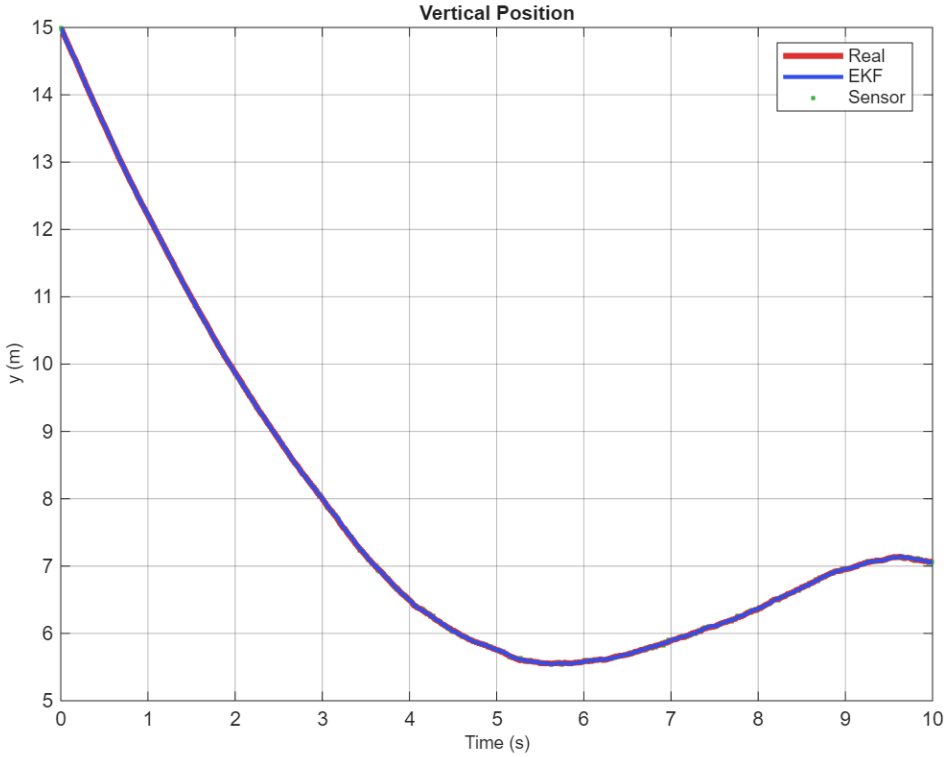

Figure 27. Vertical position (y) over time for the fall scenario, decreasing to a minimum before slight recovery, marking the lowest altitude reached.

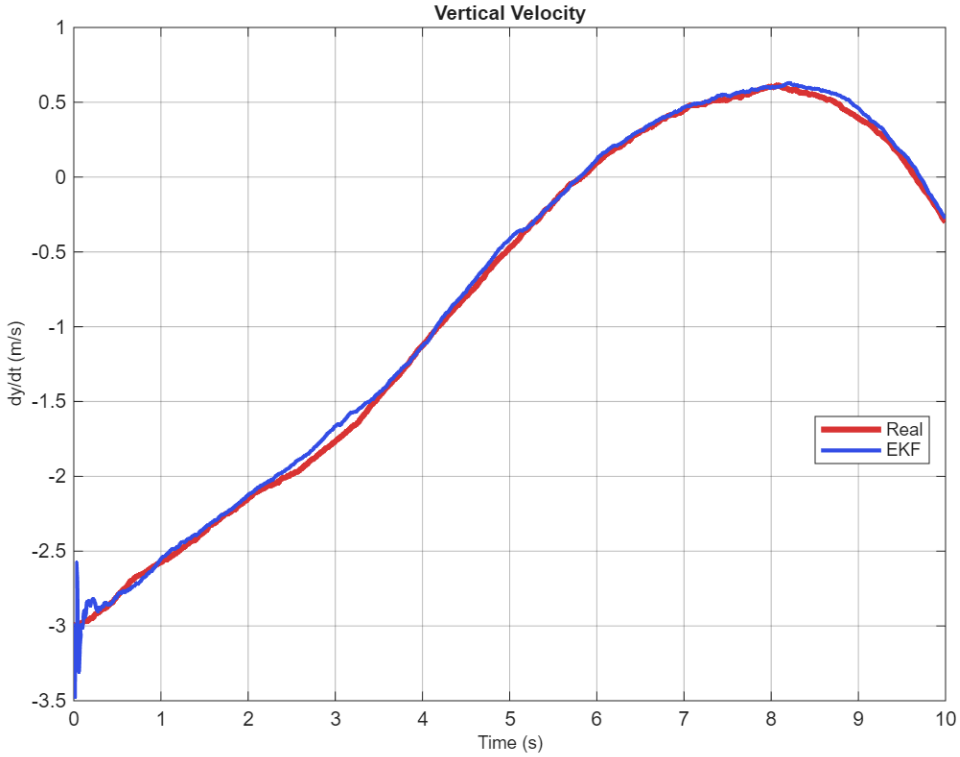

Figure 28. Vertical velocity ($\dot{y}$) over time for the fall scenario, negative during free fall and approaching zero as recovery thrust is applied.

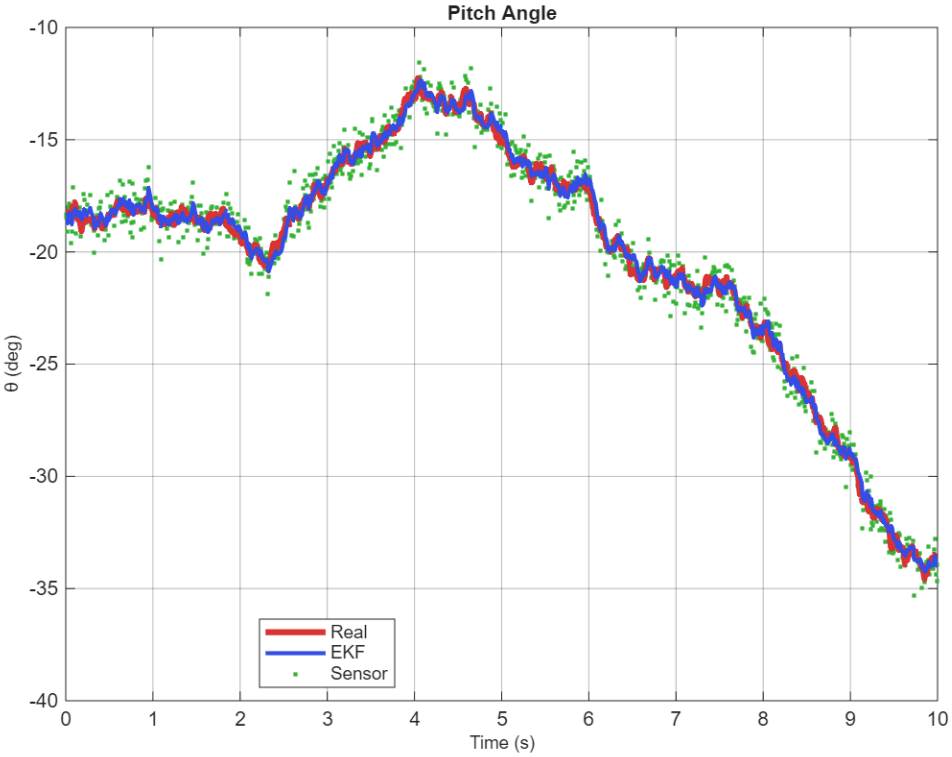

Figure 29. Pitch angle (θ) over time for the fall scenario, showing progressive nose-down rotation caused by loss of vertical stability.

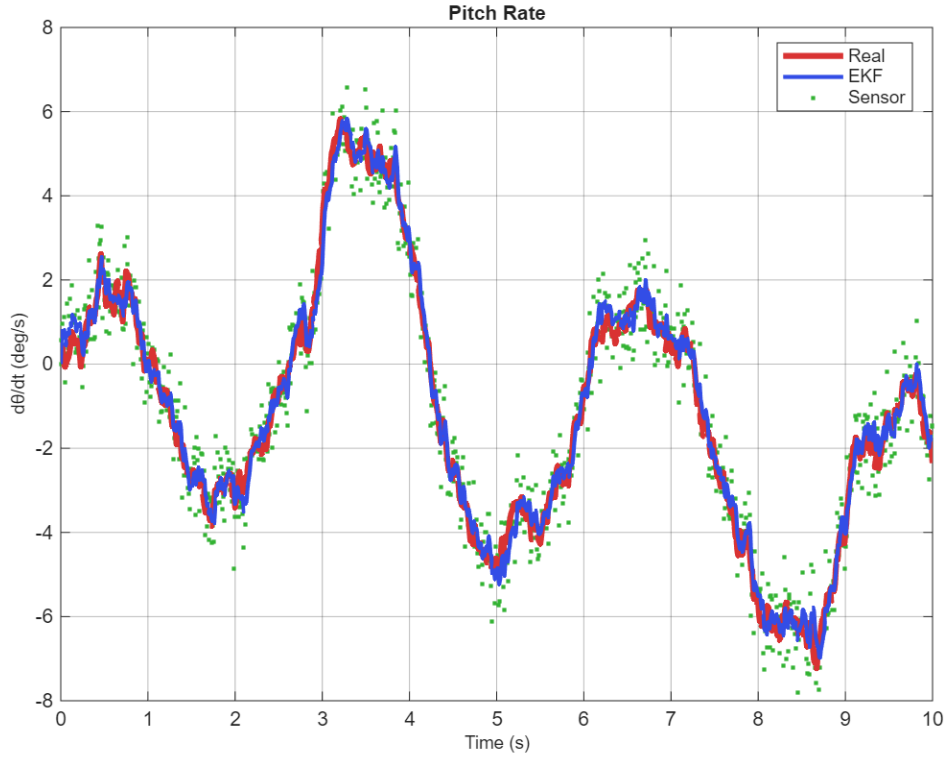

Figure 30. Pitch rate ($\dot{\theta}$) over time for the fall scenario, oscillating with sign changes that reflect alternating corrective rotational actions.

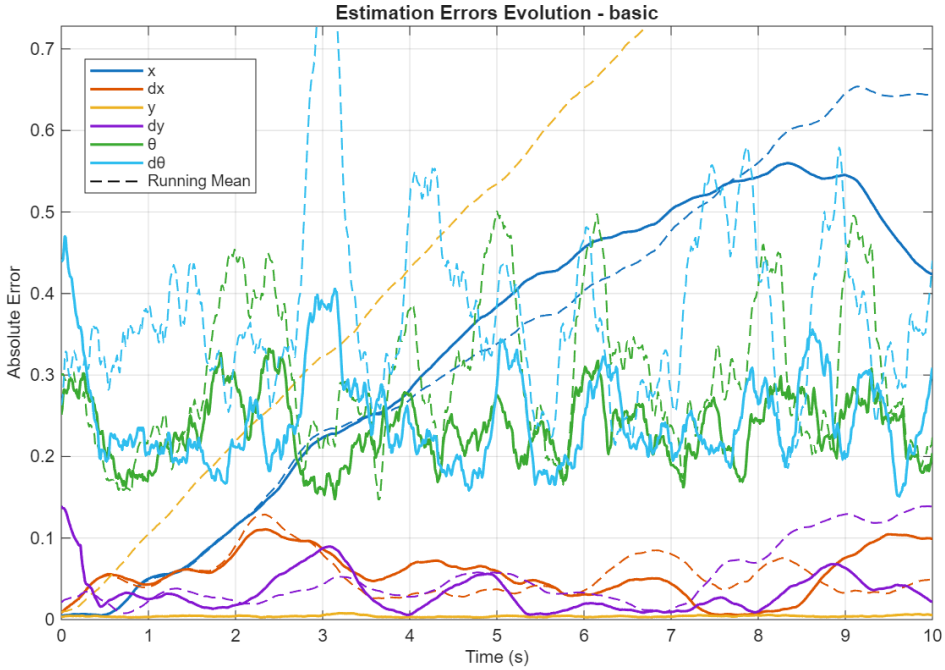

Figure 31. Absolute estimation errors for all states over time (basic scenario), comparing EKF and running mean filters. While some states show varying benefit of EKF over running mean (x, $\dot{x}$, $\dot{y}$), and others show significant error spikes of running mean (θ, $\dot{\theta}$), y in particular displays constant growth of its error, while EKF keeps almost constant.

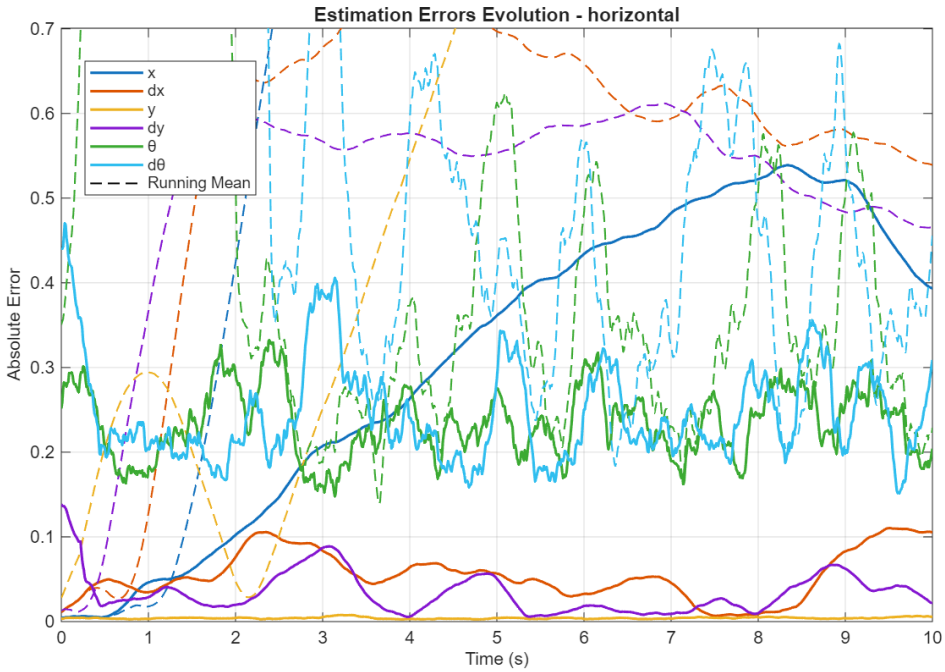

Figure 32. Absolute estimation errors for all states over time (horizontal recovery scenario). With a more control-heavy case, running mean errors show significant increase, unbound in cases of x, y and θ.

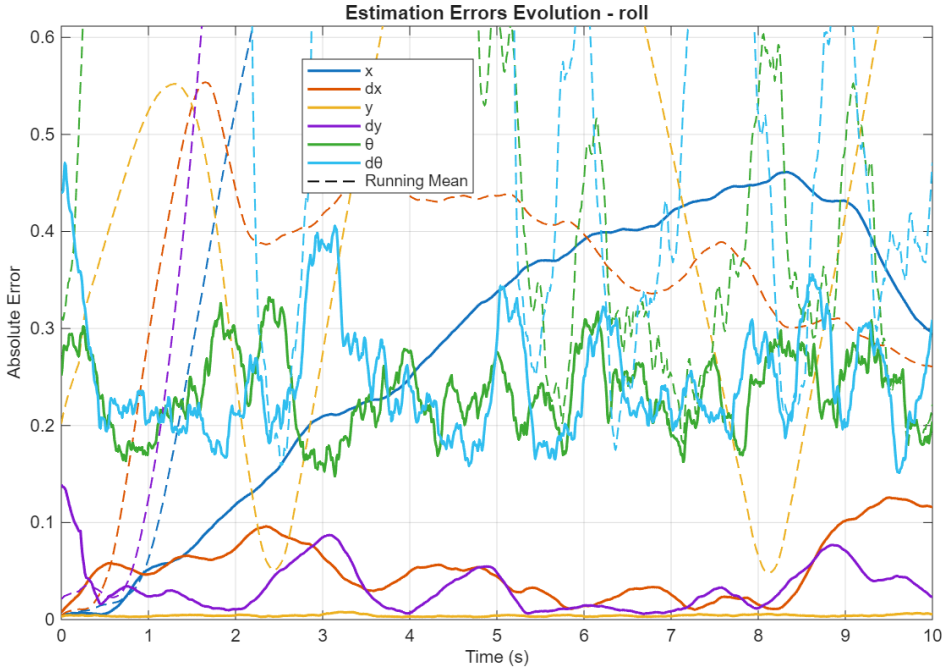

Figure 33. Absolute estimation errors for all states over time (roll scenario). This challenging simulation shows unreliability of running mean for complicated system movement.

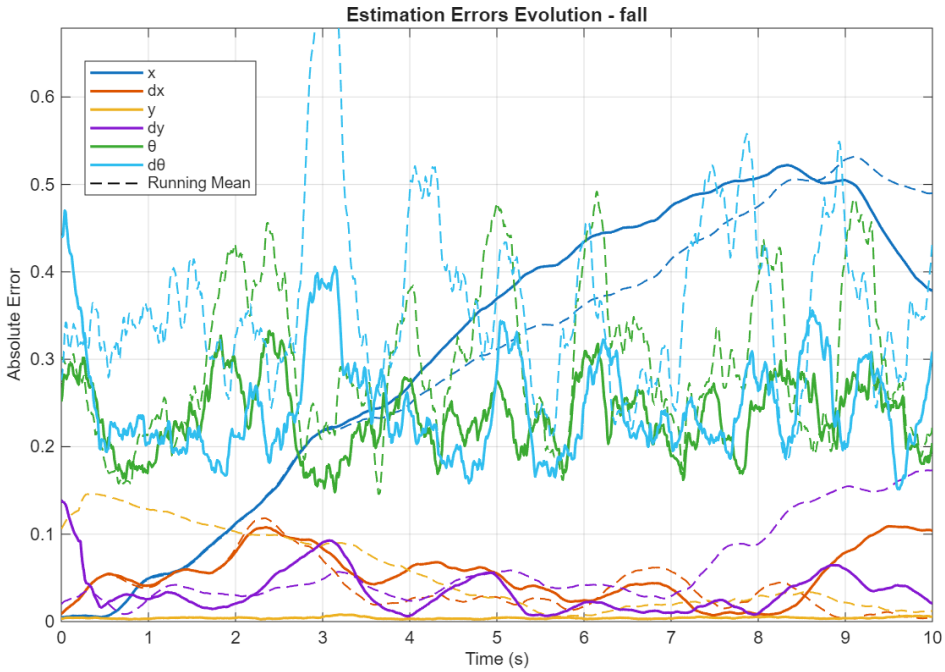

Figure 34. Absolute estimation errors for all states over time (fall recovery scenario). Not explicitly rotating system, just like the basic scenario, shows varying accuracy of running mean. Although y error is now bounded, error spikes of θ and $\dot{\theta}$ can't be ignored.

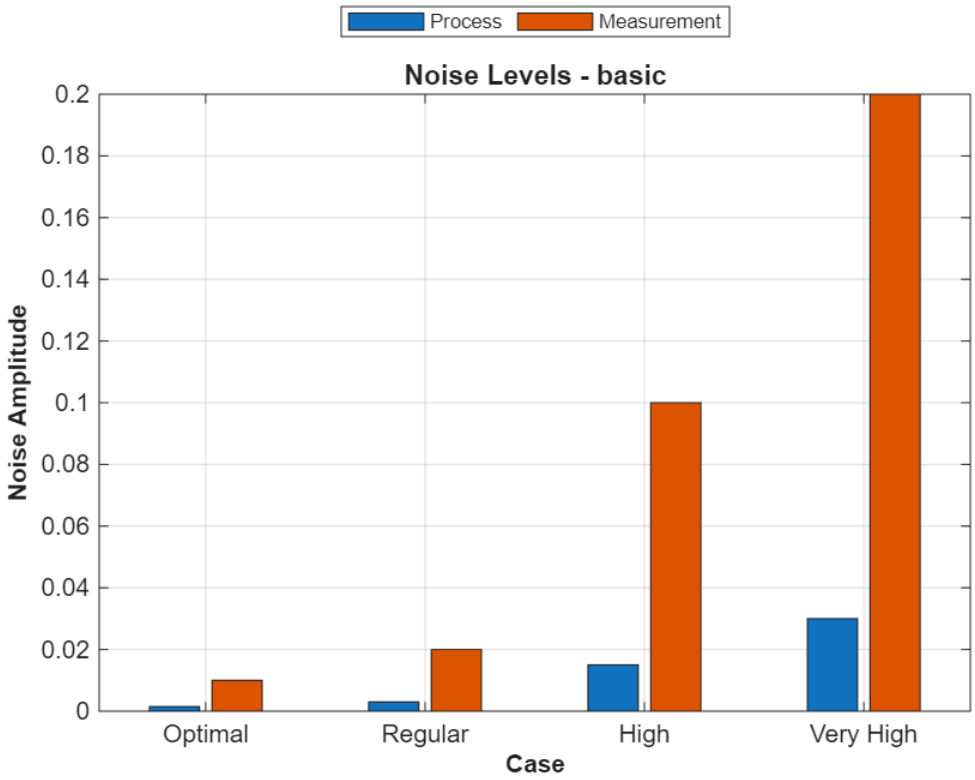

Figure 35. Process and measurement variance for robustness analysis. These values don't vary between the scenarios.

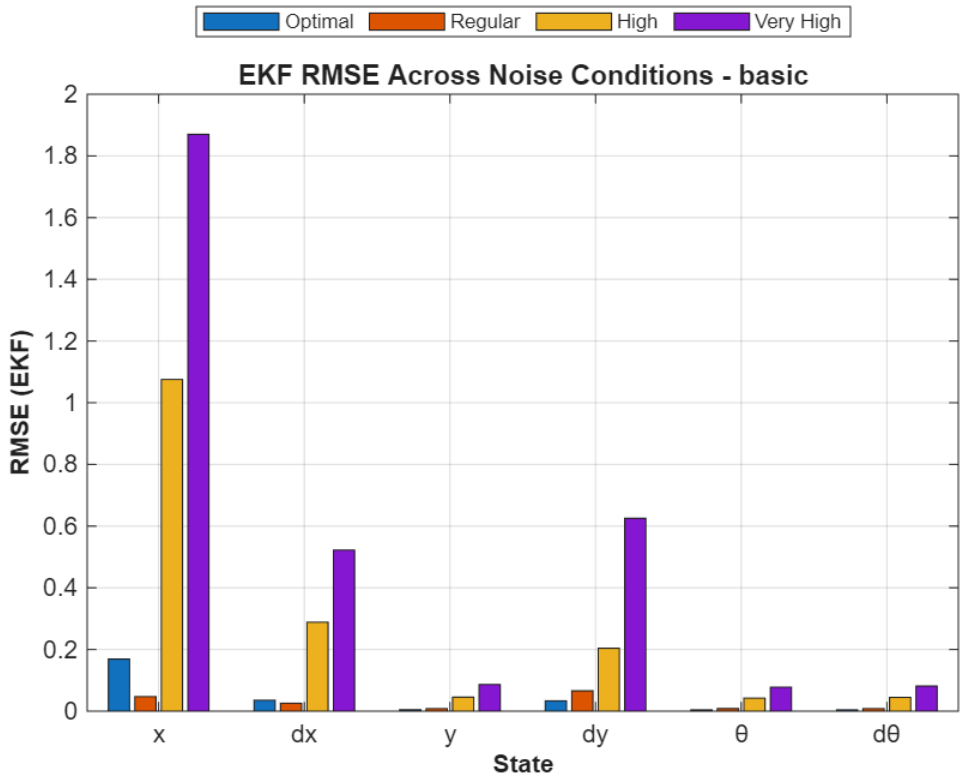

Figure 36. EKF RMSE across four noise levels for each state (basic scenario). For the most part, RMSE increases along with variance, however x and $\dot{x}$ are slightly different due to their progressing Euler integration error. Same trend can be expected in the other cases.

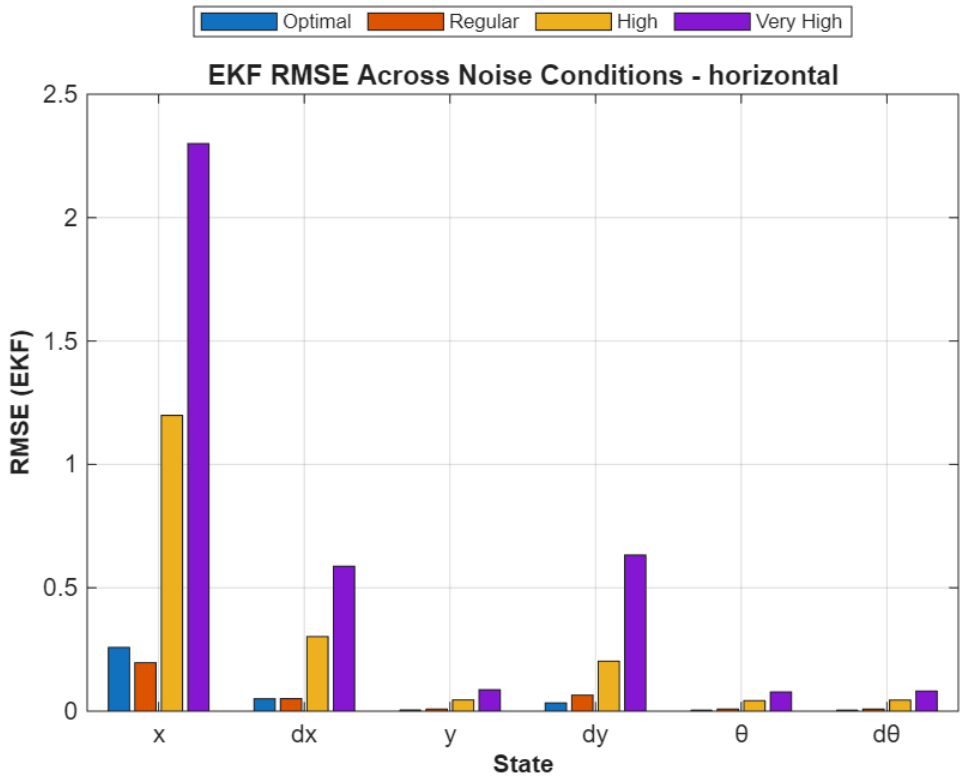

Figure 37. EKF RMSE across four noise levels for each state (horizontal recovery scenario). As expected, x is still different from the rest, for the same reasons as described in Figure 36 description. Maximum RMSE values have grown compared to the basic cases because of the increased movement complexity.

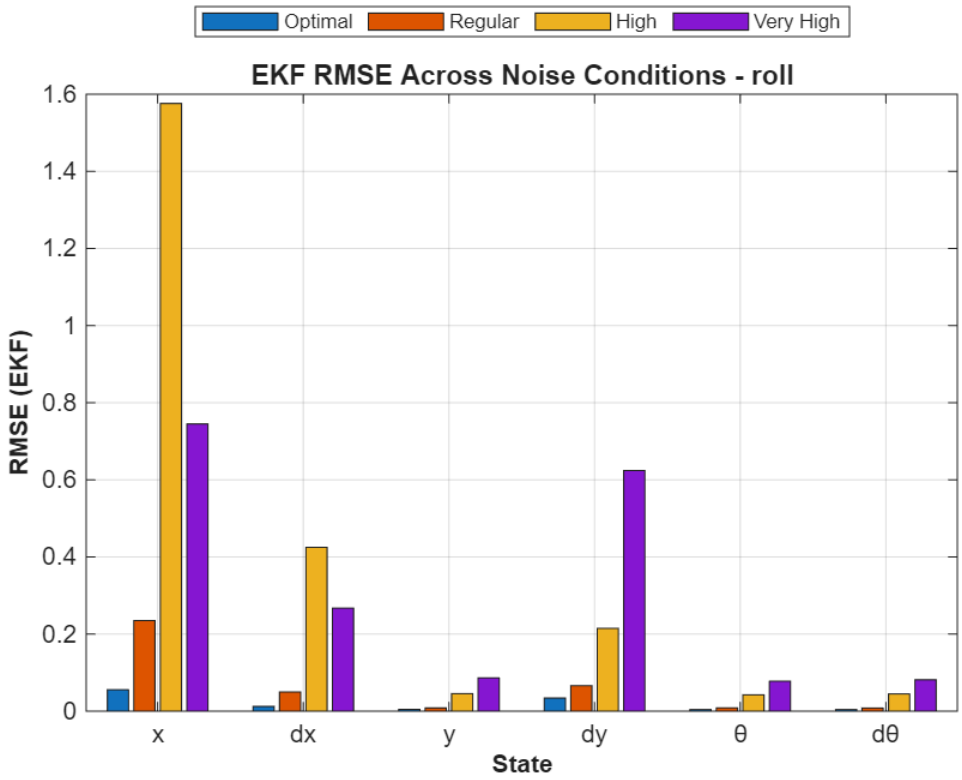

Figure 38. EKF RMSE across four noise levels for each state (roll scenario). Once again, x and its derivative display different RMSE progression when compared to the rest. This time, high variance leads to the highest RMSE values, even overtaking the very high variance.

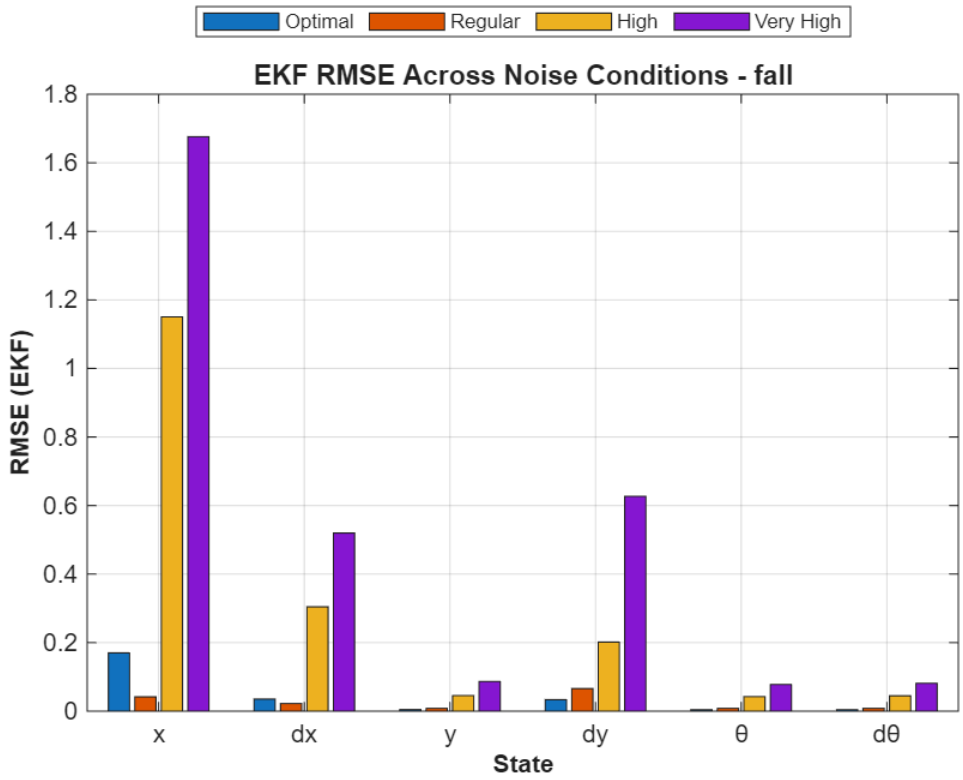

Figure 39. EKF RMSE across four noise levels for each state (fall recovery scenario). For a simple motion, same RMSE trend is seen as for the basic case (fig. 36), with similar RMSE values as well.

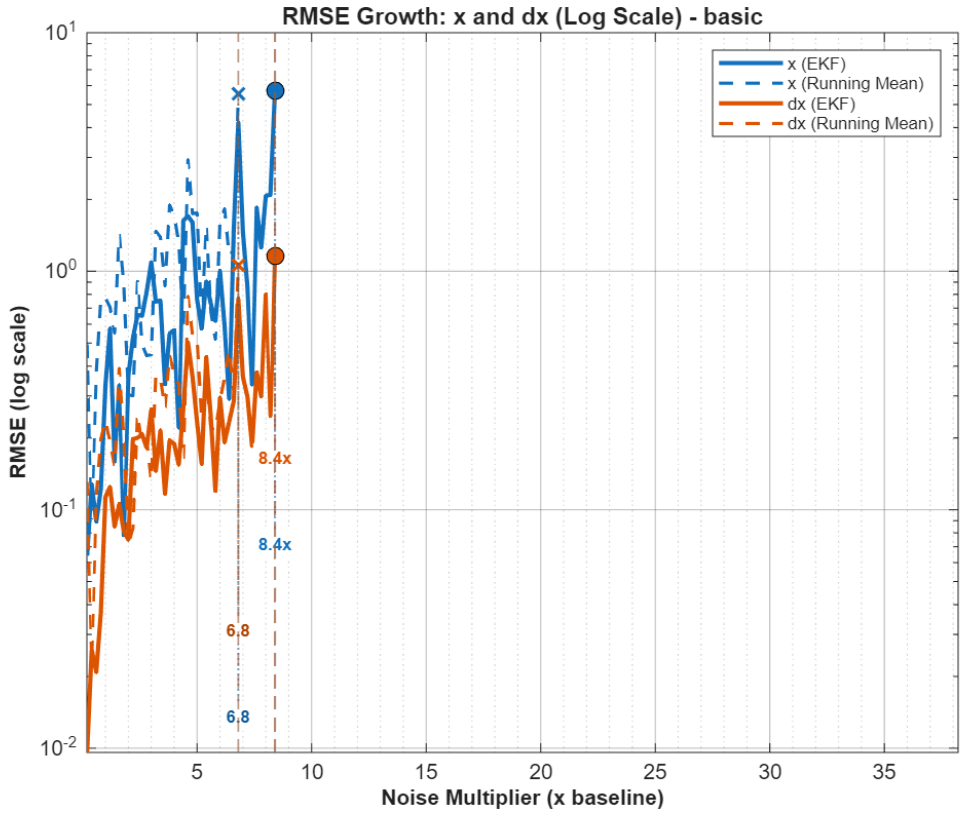

Figure 40. RMSE growth for x and $\dot{x}$ under increasing noise (log scale, basic scenario). As discussed previously, x and $\dot{x}$ are system parameters that have the biggest errors. In the simplest case, running mean is not so far from the EKF in terms of robustness.

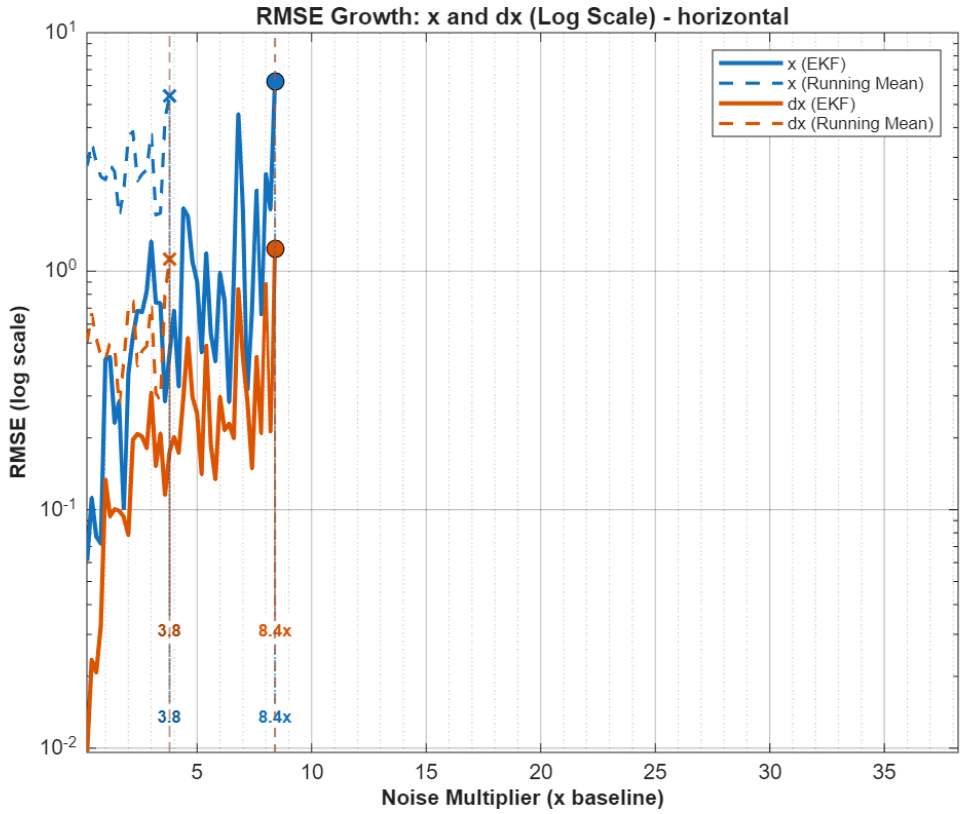

Figure 41. RMSE growth for x and $\dot{x}$ under increasing noise (log scale, horizontal recovery scenario). Complicated movement cases are harder for running mean to keep up with. Divergence of running mean happens significantly earlier compared to EKF.

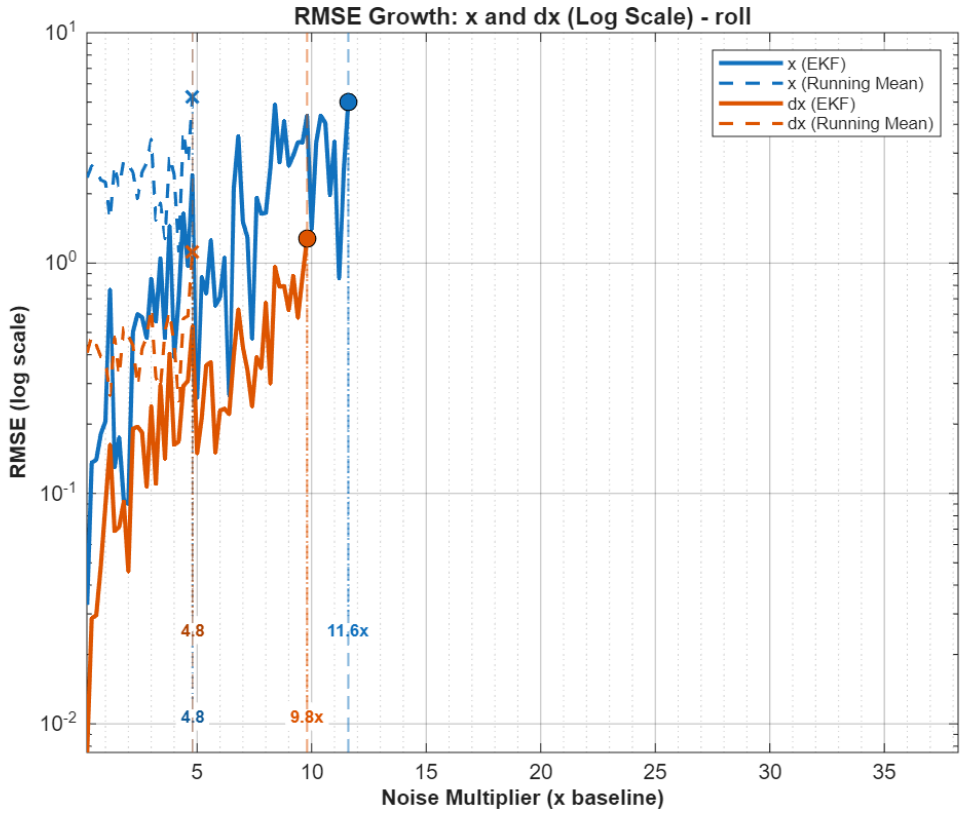

Figure 42. RMSE growth for x and $\dot{x}$ under increasing noise (log scale, roll scenario). Another control-heavy case, another early running mean divergence. Interesting difference is early divergence of $\dot{x}$.

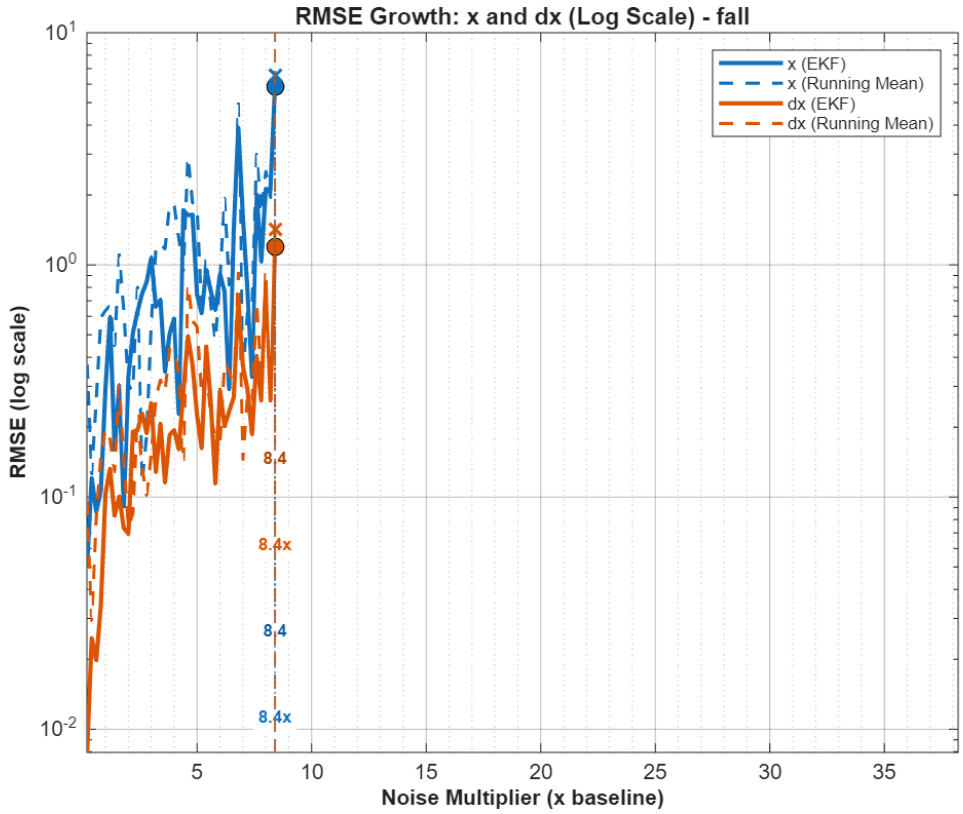

Figure 43. RMSE growth for x and $\dot{x}$ under increasing noise (log scale, fall recovery scenario). As stated before, in simple simulations without rotation running mean is actually comparable in efficiency with the EKF.

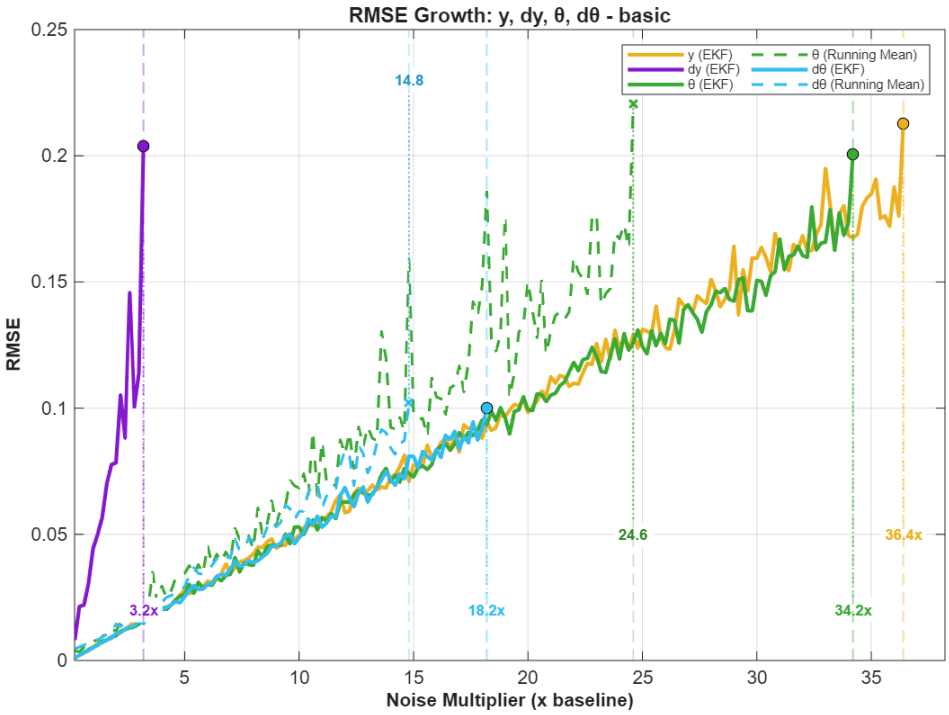

Figure 44. RMSE growth for y, $\dot{y}$, θ, and $\dot{\theta}$ under increasing noise (linear scale, basic scenario). Unlike the significant error growth of x and $\dot{x}$, other system parameters RMSE growth can be displayed linearly. Measured parameters of the system have significantly higher variance multiplier required for divergence.

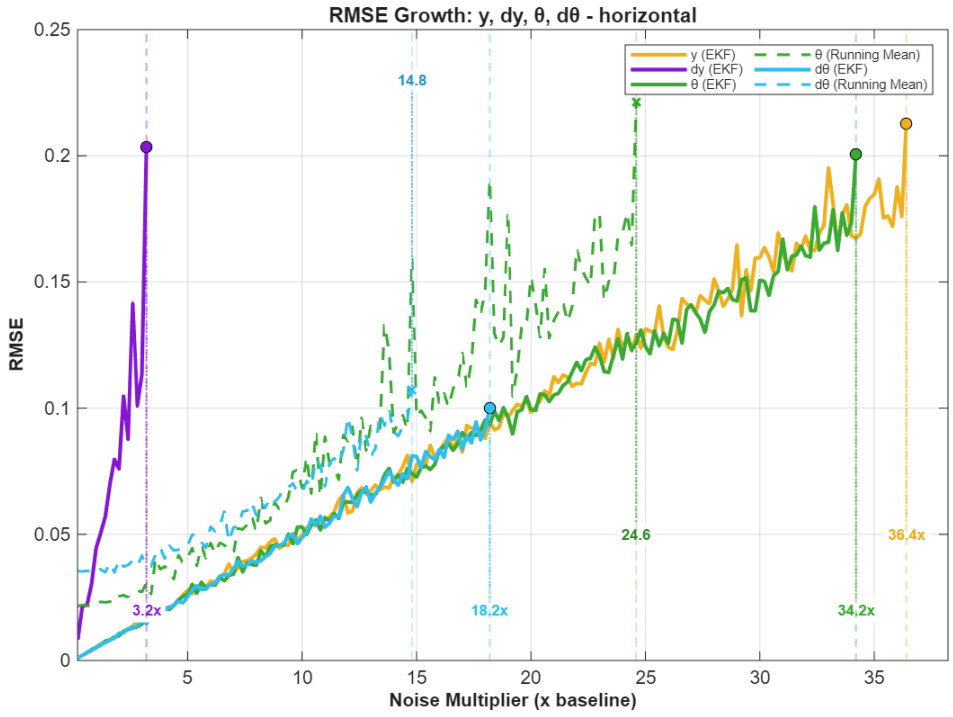

Figure 45. RMSE growth for y, $\dot{y}$, θ, and $\dot{\theta}$ under increasing noise (linear scale, horizontal recovery scenario). The only visible difference present is some slight running mean divergence multiplier (which shows how stable EKF is with different flight cases).

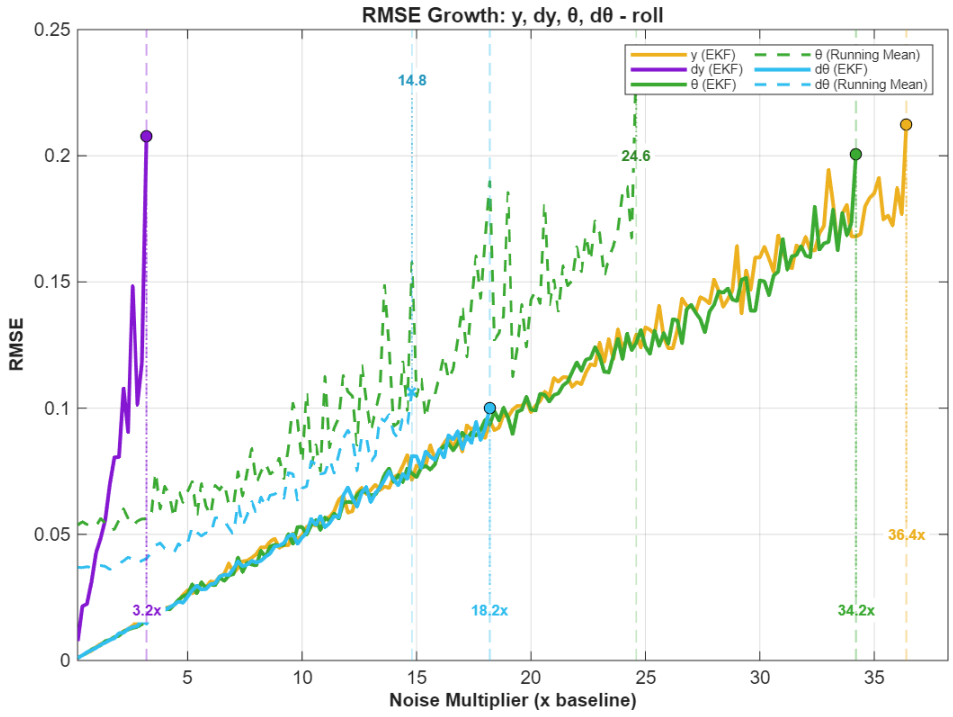

Figure 46. RMSE growth for y, $\dot{y}$, θ, and $\dot{\theta}$ under increasing noise (linear scale, roll scenario). Another case, another running mean multiplier change, EKF stable as in the two previous examples (figs. 44, 45

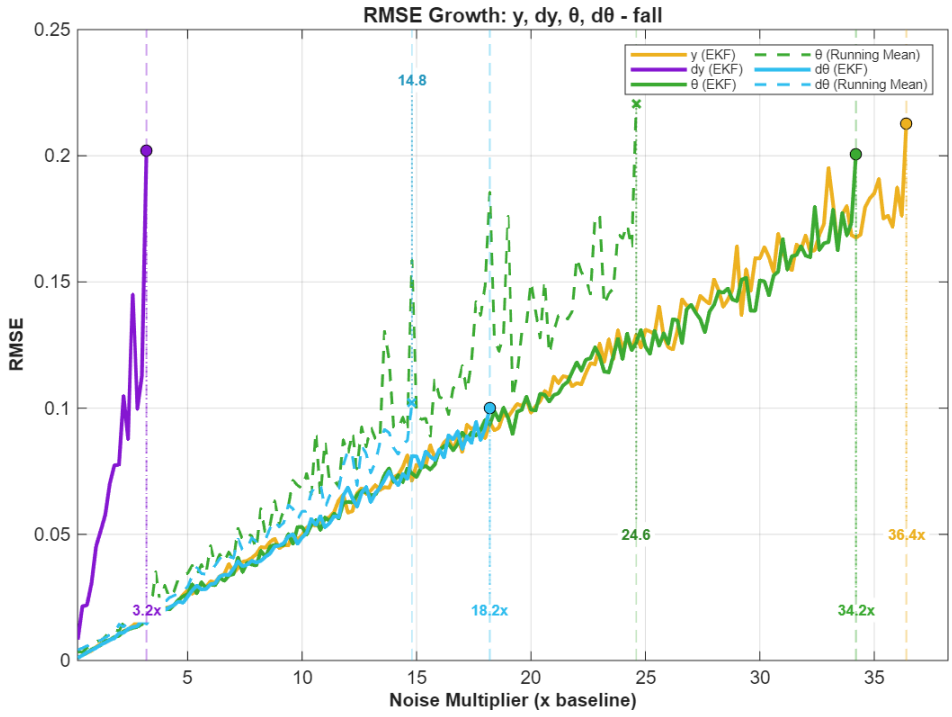

Figure 47. RMSE growth for y, $\dot{y}$, θ, and $\dot{\theta}$ under increasing noise (linear scale, fall recovery scenario). The last case definitely proves the point of EKF divergence multiplier stability for different flight conditions of the quadrotor system.

Table 1. RMSE Values for State Variables

Variable	**Kalman RMSE**	**Running Mean RMSE**	**Unfiltered RMSE**
x	0.0417	0.6632	-
$\dot{x}$	0.0222	0.1866	-
y	0.0084	0.5453	0.0098
$\dot{y}$	0.0659	0.1902	-
θ	0.0084	0.0100	0.0101
$\dot{\theta}$	0.0083	0.0095	0.0104

The EKF successfully mitigates sensor noise while maintaining close tracking of the true trajectory. It outperforms Running Mean filter. However, it is important to acknowledge the fact that the filter is perfectly tuned in all the tested scenarios.

Seed 47 was used for any random number generation in the report. All code can be accessed at https://github.com/Arseni10Lk/Planar-Quadrotor

Team member Contribution

I will add it later, just wanted to see if there is anything to correct before the final submission

Bibliography

Welch, Greg, and Gary Bishop. *An Introduction to the Kalman Filter*. Technical Report TR 95-041. Updated March 11, 2002. University of North Carolina at Chapel Hill, 2002.

Planar_Quadrotor.m

```
clc;
clear;
close all;

rng(47);

%% DEFINE VARIABLES
m = 0.5;         % mass [kg]
r = 0.15;        % distance from center to rotors [m]
I = 0.005;       % moment of inertia [kg*m^2]
g = 9.81;        % gravity [m/s^2]
dt = 0.01;       % time step [s]
theta = 0;       % angle [rads]
u1 = 5;       % force [N]
u2 = 5;       % force [N]

% Structure definition
rotor_data.m = m;
rotor_data.r = r;
rotor_data.I = I;
rotor_data.g = g;
rotor_data.dt = dt;
rotor_data.theta = theta;
rotor_data.u1 = u1;
rotor_data.u2 = u2;

```

```
t_max = 10;      % simulation duration [s]

% A matrix - System dynamics
A = [0  1  0  0   0   0;
0  0  0  0   -cos(theta)*(u1+u2)/m   0;
0  0  0  1   0   0;
0  0  0  0   -sin(theta)*(u1+u2)/m   0;
0  0  0  0   0   1;
0  0  0  0   0   0];

% It desscribes the states, so let's say it is rotor data
rotor_data.A = A;

% C matrix - Measurements
C = [0  0  1  0  0  0;
0  0  0  0  1  0;
0  0  0  0  0  1];

rotor_data.C = C; % It says what we measure, so let's say it is rotor data
% (like what the sensors are)

%% STEP 4: DEFINE TIME VECTOR
time = 0:dt:t_max;

%% STEP 5: DEFINE CONTROL INPUTS
% Create CU1 and CU2 (basic case)
CU1 = u1 * 0.6 * (1 + 0.001*cos(2*time));
```

```
CU2 = u2 * 0.6 * (1 + 0.001*sin(2*time));

% Create CU3 and CU4 (horizontal flight recovery)
CU3 = zeros(size(time));
CU4 = zeros(size(time));

for t = 0:109 % roll up
CU3(t + 1) = u1 * 0.81 * (1 + 0.001*cos(2 * 0.01 * t));
CU4(t + 1) = u2 * 0.8 * (1 + 0.001*sin(2 * 0.01 * t));
end
for t = 110:219  % stop rotation
CU3(t + 1) = u1 * 0.8 * (1 + 0.001*cos(2 * 0.01 * t));
CU4(t + 1) = u2* 0.81 * (1 + 0.001*sin(2 * 0.01 * t));
end
for t = 220:length(time) % fly up
CU3(t + 1) = u1 * 0.8 * (1 + 0.001*cos(2 * 0.01 * t));
CU4(t + 1) = u2 * 0.8 * (1 + 0.001*sin(2 * 0.01 * t));
end

% Create CU5 and CU6 (360 roll)
CU5 = zeros(size(time));
CU6 = zeros(size(time));

for t = 0:199 % iniiate roll
CU5(t + 1) = u1 * 0.808 * (1 + 0.001*cos(2 * 0.01 * t));
CU6(t + 1) = u2 * 0.8 * (1 + 0.001*sin(2 * 0.01 * t));
end
```

```
for t = 200:299 % let it roll
CU5(t + 1) = u1 * 0 * (1 + 0.001*cos(2 * 0.01 * t));
CU6(t + 1) = u2 * 0 * (1 + 0.001*sin(2 * 0.01 * t));
end
for t = 300:500 % stop rotation
CU5(t + 1) = u1 * 0 * (1 + 0.001*cos(2 * 0.01 * t));
CU6(t + 1) = u2 * 0.008 * (1 + 0.001*sin(2 * 0.01 * t));
end
for t = 501:length(time) % recover vertically
CU5(t + 1) = u1 * (1 + 0.001*cos(2 * 0.01 * t));
CU6(t + 1) = u2 * (1 + 0.001*sin(2 * 0.01 * t));
end

% Create CU7 and CU8 (straight fall recovery)
CU7 = u1 * 0.54 * (1 + 0.001*cos(2*time));
CU8 = u2 * 0.54 * (1 + 0.001*sin(2*time));

% Combine for sim
control_input.basic = [CU1; CU2]';
control_input.horizontal = [CU3; CU4]';
control_input.roll = [CU5; CU6]';
control_input.fall = [CU7; CU8]';


%% STEP 6: DEFINE NOISE AMPLITUDE & robustness testing

initial_state.basic = [0;0;1;0;0;0]; % basic case
```

```
initial_state.horizontal = [0;3;10;0;-pi/2;0]; % horizontal flight recovery
initial_state.roll = [0;0;50;5;0;0]; % roll
initial_state.fall = [0;-1;15;-3;-(pi/2-atan(3/1));0]; % straight fall recovery

noise_data.state_noise_amp = 0.003;
noise_data.output_noise_amp = 0.01;

%% STEP 7: SIMULATION

% Define the test case name here (e.g., 'fall', 'basic', 'roll', 'horizontal')
current_case = 'roll';

[states, output, error] = simulation_quadrotor(rotor_data, control_input.(current_case),
    noise_data, time, initial_state.(current_case));

% Run Robustness
[rmse_mat, noise_mat, div_data, rmse_running] = robustness(rotor_data, control_input.(
    current_case), time, initial_state.(current_case), noise_data);

% New functions with saving logic:
% Pass 'current_case' to save files with that specific name
% plot_robustness_separate_windows(rmse_mat, noise_mat, div_data, current_case);

plot_quadrotor_separate_windows(time, states, output, C, error, current_case);
```

simulation_quadrotor.m

```
function [state, output, errors] = simulation_quadrotor(rotor_data, control_input,
    noise_data, time, x0)

% Three functions combined into one

% getting all the quadrotor characteristics
m = rotor_data.m;
r = rotor_data.r;
I = rotor_data.I;
g = rotor_data.g;
dt = rotor_data.dt;
C = rotor_data.C;
A = rotor_data.A;

if nargin == 4
x0 = zeros(1,6);
end

% Initialize output and states based on input parameters
output.clean = zeros(length(time), size(C, 1));
output.real = zeros(length(time), size(C, 1));
output.filtered = zeros(length(time), size(C, 1));
output.running = zeros(length(time), size(C, 1));%RUNNING

state.clean = zeros(length(time), size(A, 1));
state.real = zeros(length(time), size(A, 1));
state.estimate = zeros(length(time), size(A, 1));
```

```
state.running = zeros(length(time), size(A, 1));%RUNNING

% Simulate the system dynamics over the specified time

% Set initial state
state.clean(1, :) = x0;
state.real(1, :) = x0;
state.estimate(1, :) = x0;
state.running(1, :) = x0;%RUNNING

output.clean(1, :) = C*state.clean(1, :)';
output.real(1, :) = output.clean(1, :);
output.filtered(1, :) = output.clean(1, :);
output.running(1, :) = output.clean(1, :);%RUNNING

% Filter data

P = eye(6); % Initial Uncertainty
% Q: Process Noise Covariance (Trust in Physics)
% We use a small value to allow the model to drive the smoothness
Q = eye(6) * (0.003^2);
% R: Measurement Noise Covariance (Trust in Sensors)
% We set this higher than actual noise to filter out the jitters
R = eye(3) * (0.01^2);

% Running
window_size = 10; % You can adjust this window size
```

```
54
55 for t = 2:length(time)
56
57 %%% Perfect simulation
58
59 % update theta
60 theta_clean = state.clean(t - 1, 5);
61
62 % Update states based on linear dynamics, this works for everything
63 % except dx and dy
64 delta_x_clean(1) = state.clean(t - 1, 2)*dt;
65 delta_x_clean(3) = state.clean(t - 1, 4)*dt;
66 delta_x_clean(5) = state.clean(t - 1, 6)*dt;
67 delta_x_clean(6) = (r / I * control_input(t, 1) - r / I * control_input(t, 2)) * dt;
68
69 % Now, non-linear part
70 delta_x_clean(2) = (-sin(theta_clean)*(control_input(t, 1)+control_input(t, 2))/m) * dt;
71 delta_x_clean(4) = (cos(theta_clean)*(control_input(t, 1)+control_input(t, 2))/m - g) * dt;
72
73
74 % Lastly, updating states
75 state.clean(t, :) = state.clean(t-1, :) + delta_x_clean(:)';
76 output.clean(t, :) = (C * state.clean(t, :)')';
77
78 %%% Real-world simulation
79
80 % update theta
```

```
theta_real = state.real(t - 1, 5);

% Update states based on linear dynamics, this works for everything
% except dx and dy
delta_x_real(1) = state.real(t - 1, 2)*dt;
delta_x_real(3) = state.real(t - 1, 4)*dt;
delta_x_real(5) = state.real(t - 1, 6)*dt;
delta_x_real(6) = (r / I * control_input(t, 1) - r / I * control_input(t, 2)) * dt;

% Now, non-linear part
delta_x_real(2) = (-sin(theta_real)*(control_input(t, 1)+control_input(t, 2))/m) * dt;
delta_x_real(4) = (cos(theta_real)*(control_input(t, 1)+control_input(t, 2))/m - g) * dt;

% getting all the noise characteristics
state_noise = noise_data.state_noise_amp * randn(1, size(A, 1));
output_noise = noise_data.output_noise_amp * randn(1, size(C, 1));

state.real(t, :) = state.real(t - 1, :) + delta_x_real(:)' + state_noise;
output.real(t, :) = (C*state.real(t, :)')' + output_noise;

%%% Filter
% Filter uses perfect physics and sensor measurements, but we do not pass
% real state to it.

% PREDICTION STEP

% update theta
```

```
108 theta_estimate = state.estimate(t - 1, 5);
109
110 % Update states based on linear dynamics, this works for everything
111 % except dx and dy
112 delta_x_estimate(1) = state.estimate(t - 1, 2)*dt;
113 delta_x_estimate(3) = state.estimate(t - 1, 4)*dt;
114 delta_x_estimate(5) = state.estimate(t - 1, 6)*dt;
115 delta_x_estimate(6) = (r / I * control_input(t, 1) - r / I * control_input(t, 2)) * dt;
116
117 % Now, non-linear part
118 delta_x_estimate(2) = (-sin(theta_estimate)*(control_input(t, 1)+control_input(t, 2))/m) *
        dt;
119 delta_x_estimate(4) = (cos(theta_estimate)*(control_input(t, 1)+control_input(t, 2))/m - g)
        * dt;
120
121 state.estimate(t, :) = state.estimate(t - 1, :) + delta_x_estimate(:)';
122
123 % Predict covariance
124
125 F = eye(6) + dt * [0 1 0 0 0 0;
126 0 0 0 0 -cos(theta_estimate)*(control_input(t, 1)+control_input(t, 2))/m 0;
127 0 0 0 1 0 0;
128 0 0 0 0 -sin(theta_estimate)*(control_input(t, 1)+control_input(t, 2))/m 0;
129 0 0 0 0 0 1;
130 0 0 0 0 0 0];
131
132 P_prediction = F * P * F' + Q;
```

```
133
134 % CORRECTION STEP
135
136 % 1. Calculate Measurement Residual
137 measurement_residual = output.real(t, :)' - C * state.estimate(t, :)';
138 measurement_residual(2) = mod(measurement_residual(2) + pi, 2*pi) - pi;
139
140 % 2. Calculate Kalman Gain
141 S = C * P_prediction * C' + R;
142 K = P_prediction * C' / S;
143
144 % 3. Update State Estimate
145 state.estimate(t, :) = state.estimate(t, :) + (K * measurement_residual)';
146 output.filtered(t, :) = C * state.estimate(t, :)';
147
148 % 4. Update Covariance
149 P = (eye(6) - K * C) * P_prediction;
150
151
152 % RUNNING
153 % Determine the start of the window
154 window_start = max(1, t - window_size);
155
156 % Average the noisy measurements ('real' outputs) over the window
157 output.running(t, :) = mean(output.real(window_start:t, :), 1);
158
159 % update theta
```

```
160 theta_running = output.running(t, 2);
161
162 state.running(t, 3) = output.running(t, 1); % Overwrite y (State 3)
163 state.running(t, 5) = output.running(t, 2); % Overwrite theta (State 5)
164 state.running(t, 6) = output.running(t, 3); % Overwrite theta_dot (State 6)
165
166 % For the states, we apply the same averaging to the noisy running states
167 % Update states based on linear dynamics, this works for everything
168 % except dx and dy
169 delta_x_running(1) = state.running(t-1, 2) * dt;
170
171 % Now, non-linear part
172 delta_x_running(2) = (-sin(theta_running)*(control_input(t, 1)+control_input(t, 2))/m) * dt;
173 delta_x_running(4) = (cos(theta_running)*(control_input(t, 1)+control_input(t, 2))/m - g) *
        dt;
174
175 state.running(t, [1 2 4]) = state.running(t - 1, [1 2 4]) + delta_x_running([1 2 4]);
176
177 end
178
179
180 %RUNNING
181 errors.output_clean_VS_running_total = output.running - output.clean;
182 errors.states_real_VS_running = state.real - state.running;
183
184 errors.output_clean_VS_real_total = output.real - output.clean;
185 errors.output_clean_VS_filtered_total = output.filtered - output.clean;
```

```
errors.output_real_VS_filtered_total = output.filtered - output.real;

errors.states_clean_VS_real_total = state.real - state.clean;
errors.states_real_VS_estimate = state.real - state.estimate;
errors.state_real_VS_output_real = output.real - state.real(:, [3, 5, 6]);
errors.state_real_VS_output_filtered = output.filtered - state.real(:, [3, 5, 6]);

% Calculate the Mean Squared Error for each column (state variable)
num_states = size(errors.states_real_VS_estimate, 2);
rmse_values = zeros(1, num_states);
rmse_running = zeros(1, num_states);


for i = 1:num_states
% RMSE = sqrt(mean(error^2))
rmse_values(i) = sqrt(mean(errors.states_real_VS_estimate(:, i).^2));
%RUNNING
rmse_running(i) = sqrt(mean(errors.states_real_VS_running(:, i).^2));
end

rmse_unfiltered = zeros(1, 3);

for i = 1:3
rmse_unfiltered(i) = sqrt(mean(errors.state_real_VS_output_real(:, i).^2));
end

% Store the RMSE values in the errors structure
```

```

% state variables are typically: [x, dx, y, dy, theta, dtheta]
errors.rmse_states = rmse_values;
%RUNNING
errors.rmse_running = rmse_running;

errors.rmse_unfiltered = rmse_unfiltered;
end

```

robustness.m

```
function [rmse_matrix, noise_matrix, divergence_data, rmse_matrix_running] = robustness(
    rotor_data, control_input, time, initial_state, noise_data_base)
% ROBUSTNESS - Run robustness tests and return data for plotting
% Returns: rmse_matrix (Kalman), noise_matrix, divergence_data, rmse_matrix_running (Running
    )

fprintf('\n=== Running Robustness Analysis ===\n');

% Define 4 noise cases (Cases 1-4)
noise_cases = [
0.0015, 0.01; % Case 1: Optimal
0.003,  0.02; % Case 2: Regular
0.015,  0.1;  % Case 3: High noise
0.03,   0.2   % Case 4: Very high noise
];

```

```
num_cases = size(noise_cases, 1);
rmse_matrix = zeros(num_cases, 6);
rmse_matrix_running = zeros(num_cases, 6); % NEW: Store running filter RMSE
noise_matrix = noise_cases;

% Run simulations for Cases 1-4
for case_num = 1:num_cases
noise_data.state_noise_amp = noise_cases(case_num, 1);
noise_data.output_noise_amp = noise_cases(case_num, 2);

[~, ~, errors] = simulation_quadrotor(rotor_data, control_input, noise_data, time,
    initial_state);

rmse_matrix(case_num, :) = errors.rmse_states;
rmse_matrix_running(case_num, :) = errors.rmse_running; % NEW

fprintf('Case %d RMSE (Kalman): ', case_num);
fprintf('%.4f ', errors.rmse_states);
fprintf('\n');
end

% Run divergence analysis
divergence_data = find_divergence_individual_states(rotor_data, control_input, time,
    initial_state, noise_data_base);

fprintf('\n=== Analysis Complete === \n');
end
```

```
40
41 %% ================= HELPER FUNCTION - TRACKS INDIVIDUAL STATE DIVERGENCE ================
42 function div_data = find_divergence_individual_states(rotor_data, control_input, time,
       initial_state, noise_data_base)
43 base_noise = [noise_data_base.state_noise_amp, noise_data_base.output_noise_amp]; % Base
       noise levels
44 max_multiplier = 50;
45 multiplier = 0.2;
46
47 div_data.multipliers = [];
48 div_data.rmse_values = [];          % Kalman RMSE history
49 div_data.rmse_values_running = [];  % NEW: Running RMSE history
50
51 % State names
52 state_names = {'x', 'dx', 'y', 'dy', 'theta', 'dtheta'};
53 display_names = {'x', 'dx', 'y', 'dy', 'θ', 'dθ'};
54 div_data.state_names = state_names;
55 div_data.display_names = display_names;
56
57 % Initialize divergence tracking for BOTH filters
58 for i = 1:6
59 % Kalman Tracking
60 div_data.(['div_point_' state_names{i}]) = 0;
61 div_data.(['actually_diverged_' state_names{i}]) = false;
62 div_data.(['threshold_' state_names{i}]) = 0;
63
64 % NEW: Running Filter Tracking
```

```
div_data.(['div_point_running_' state_names{i}]) = 0;
div_data.(['actually_diverged_running_' state_names{i}]) = false;
end

% Thresholds (Shared between filters)
thresholds = [5.0, 1.0, 0.2, 0.2, 0.2, 0.1];
for i = 1:6
div_data.(['threshold_' state_names{i}]) = thresholds(i);
end

fprintf('\n--- Starting Dual-Filter Divergence Analysis ---\n');

iteration = 0;
while multiplier <= max_multiplier
iteration = iteration + 1;
noise_data.state_noise_amp = base_noise(1) * multiplier;
noise_data.output_noise_amp = base_noise(2) * multiplier;

[~, ~, errors] = simulation_quadrotor(rotor_data, control_input, noise_data, time,
    initial_state);

div_data.multipliers(end+1) = multiplier;
div_data.rmse_values(end+1, :) = errors.rmse_states;
div_data.rmse_values_running(end+1, :) = errors.rmse_running; % NEW

% Check EACH STATE for divergence (Kalman & Running)
for state_idx = 1:6
```

```
91 threshold = thresholds(state_idx);
92 s_name = state_names{state_idx};
93
94 % 1. Check Kalman
95 if errors.rmse_states(state_idx) > threshold
96 if ~div_data.(['actually_diverged_' s_name])
97 div_data.(['actually_diverged_' s_name]) = true;
98 div_data.(['div_point_' s_name]) = multiplier;
99 fprintf('  [Kalman]  %s diverged at %.1fx\n', display_names{state_idx}, multiplier);
100 end
101 end
102
103 % 2. Check Running (NEW)
104 if errors.rmse_running(state_idx) > threshold
105 if ~div_data.(['actually_diverged_running_' s_name])
106 div_data.(['actually_diverged_running_' s_name]) = true;
107 div_data.(['div_point_running_' s_name]) = multiplier;
108 fprintf('  [Running] %s diverged at %.1fx\n', display_names{state_idx}, multiplier);
109 end
110 end
111 end
112
113 % Progress
114 if mod(iteration, 10) == 0
115 fprintf('  Progress: %.1f/%.1f\n', multiplier, max_multiplier);
116 end
117
```

```
% Early exit: Only if BOTH filters have failed on ALL states
all_diverged = true;
for state_idx = 1:6
s_name = state_names{state_idx};
if ~div_data.(['actually_diverged_' s_name]) || ~div_data.(['actually_diverged_running_'
    s_name])
all_diverged = false;
break;
end
end

if all_diverged
fprintf('  All states in both filters diverged. Stopping.\n');
break;
end

multiplier = multiplier + 0.2;
end

% Cleanup: Set stable states to max_tested
max_tested = max(div_data.multipliers);
div_data.max_multiplier_tested = max_tested;

for state_idx = 1:6
s_name = state_names{state_idx};
if ~div_data.(['actually_diverged_' s_name])
div_data.(['div_point_' s_name]) = max_tested;
```

```
144 end
145 if ~div_data.(['actually_diverged_running_' s_name])
146 div_data.(['div_point_running_' s_name]) = max_tested;
147 end
148 end
149 end
150
```