Performance Comparison of Two Filters for Vehicle GPS-Based Velocity and Position Estimation

Mustafa Tursun 19016909
Eren Topaçoğlu 20016029
Kamil Hanoğlu 19015022

January 18, 2025

Abstract

In modern technological advancements, accurate state estimation plays a pivotal role in ensuring the reliability and safety of autonomous systems, particularly in applications like autonomous driving. The ability of a vehicle to determine its position, velocity, and orientation (yaw angle, ψ) is fundamental to achieving precise control, efficient path planning, and obstacle avoidance. To accomplish this, sensor fusion techniques are widely employed, integrating measurements from Global Positioning System (GPS) sensors and inertial measurement units (IMUs) to derive a comprehensive understanding of the vehicle's dynamics. GPS provides reliable global position and velocity data, albeit at a lower update rate and subject to measurement noise. However, the presence of noise and intermittent loss of signals in challenging environments, such as urban canyons or tunnels, poses limitations to standalone GPS systems. On the other hand, IMUs, capable of measuring accelerations and angular rates, complement GPS by offering high-frequency data essential for capturing rapid state transitions. The fusion of these sensors through advanced algorithms like Kalman Filters allows for robust state estimation by leveraging the strengths of both systems while mitigating their individual weaknesses. This project focuses on designing a nonlinear Kalman Filter to estimate the vehicle's position, velocity, and yaw angle (ψ) using a mathematical model of the vehicle and sensor measurements. The vehicle's dynamics are modeled as a discrete-time nonlinear system, where the control inputs include longitudinal and lateral accelerations and yaw rate. The measurement model incorporates GPS-based global positions and ENU (East-North-Up) velocity components, providing a comprehensive representation of the system's state. By implementing an Extended Kalman Filter (EKF) and an Unscented Kalman Filter (UKF), this study highlights the effectiveness of nonlinear filtering techniques in handling the inherent nonlinearities of the vehicle's motion and measurement models. Such a system is not only vital for the accurate localization of autonomous vehicles but also contributes to enhancing their ability to navigate complex environments and maintain stability under varying conditions. This research, therefore, serves as a critical step in advancing state estimation methodologies, aligning with the broader objectives of safe and efficient autonomous driving technologies.

Keywords: Kalman Filter, Dynamic Model, Measurement Model, Nonlinear Kalman Filtering, EKF and UKF

1 Introduction

Dynamic modeling plays a central role in state estimation and control tasks for autonomous systems, including self-driving vehicles. These models provide the foundation for designing algorithms capable of predicting and controlling system behavior. Vehicle State Estimation (VSE), leveraging Kalman Filters (KF), has been widely applied in this context to ensure robust localization and state prediction under dynamic and noisy conditions.

Ragab, Khamis and Napoleon [1] emphasized the utility of Extended Kalman Filters (EKF) and Unscented Kalman Filters (UKF) in multi-sensor fusion frameworks, particularly for tracking dynamic objects in autonomous driving environments. Their study demonstrated that UKF surpasses EKF in non-linear scenarios due to its ability to handle non-linearity without requiring explicit linearization.

In another study, Lin, Yoon and Kim (2020) [2] developed a Particle-Aided Unscented Kalman Filter (PAUKF) approach to improve localization accuracy in self-driving cars. By combining particle filters (PF) with UKF, their method integrated non-Gaussian noise models and high-definition maps, achieving precise localization even in urban environments where GPS signals are unreliable.

Similarly, Bersani, Mentasti, Vignati, Arrigoni and Cheli [3] explored adaptive EKF and UKF methods for vehicle state estimation, focusing on real-time implementation. Their work highlighted the importance of dynamic models, such as the single-track kinematic model, in ensuring reliable state estimation while balancing computational efficiency.

These studies underline the growing significance of dynamic models in Kalman filter applications. While traditional VSE approaches rely on simplified linear models, modern techniques like UKF and PAUKF accommodate complex, non-linear dynamics, making them suitable for advanced systems such as autonomous vehicles.

Key Contributions in Literature

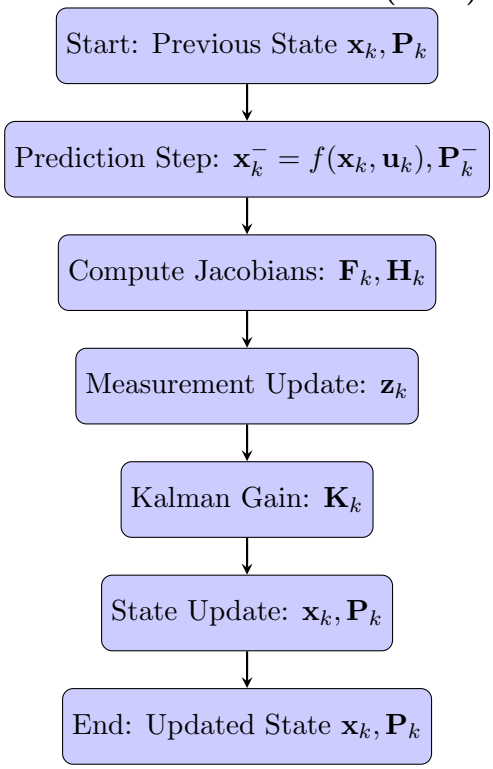
- Ragab, Khamis, and Napoleon [1] demonstrated the superior performance of UKF over EKF in handling non-linearities in dynamic object tracking within autonomous driving contexts.
- Lin, Yoon, and Kim (2020) [2] proposed the Particle-Aided Unscented Kalman Filter (PAUKF), integrating PF and UKF to address challenges in urban localization with non-Gaussian noise.
- Bersani, Mentasti, and Arrigoni [3] implemented real-time adaptive EKF and UKF for state estimation, emphasizing computational efficiency and the use of dynamic models for enhanced accuracy.

2 System Model

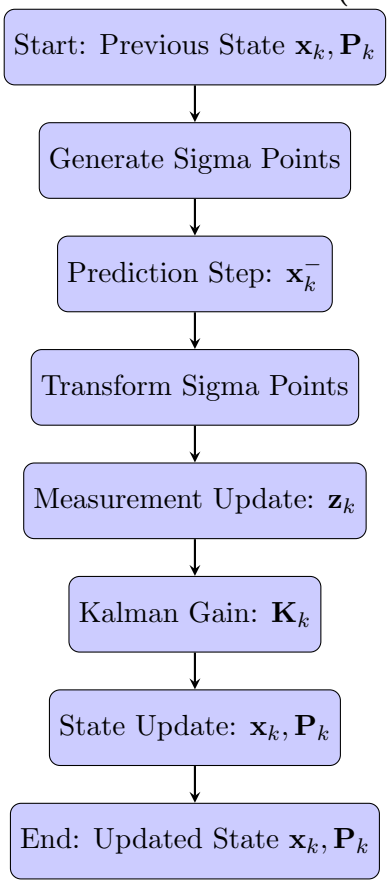
The Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF) operate on a non-linear system designed specifically for vehicle state estimation. The process involves predicting the vehicle's position, velocity, and yaw angle (ψ) based on control inputs and refining these predictions with sensor measurements. The flowcharts below illustrate the specific steps for the EKF and UKF algorithms:

2.1 EKF and UKF Flowchart Algorithm

Extended Kalman Filter (EKF)



Unscented Kalman Filter (UKF)



These flowcharts provide a clear representation of the steps involved in both the EKF and UKF algorithms, tailored for the nonlinear vehicle state estimation problem in this project.

State Equation

$$\mathbf{x}_{k+1} = \begin{bmatrix} V_{x,k+1} \\ V_{y,k+1} \\ \psi_{k+1} \\ x_{g,k+1} \\ y_{g,k+1} \end{bmatrix} = f(\mathbf{x}_k, \mathbf{u}_k, \Delta t) + \mathbf{w}_k, \tag{1}$$

where:

- $\mathbf{x}_k$ is the state vector at time step k, defined as $\mathbf{x}_k = [V_{x,k}, V_{y,k}, \psi_k, x_{g,k}, y_{g,k}]^T$,
- $f(\mathbf{x}_k, \mathbf{u}_k, \Delta t)$ is the nonlinear state transition function given by:

$$\begin{aligned} V_{x,k+1} &= V_{x,k} + \Delta t(V_{y,k}\omega_k + a_{x,k}), \\ V_{y,k+1} &= V_{y,k} + \Delta t(-V_{x,k}\omega_k + a_{y,k}), \\ \psi_{k+1} &= \psi_k + \Delta t\omega_k, \\ x_{g,k+1} &= x_{g,k} + \Delta t(V_{x,k}\cos(\psi_k) - V_{y,k}\sin(\psi_k)), \\ y_{g,k+1} &= y_{g,k} + \Delta t(V_{x,k}\sin(\psi_k) + V_{y,k}\cos(\psi_k)), \end{aligned} \tag{2}$$

- $\mathbf{u}_k = [a_{x,k}, a_{y,k}, \omega_k]^T$ is the control input vector (longitudinal acceleration, lateral acceleration, yaw rate),
- $\mathbf{w}_k$ is the process noise, assumed to be Gaussian with covariance $\mathbf{Q}$.

Measurement Equation

$$\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k, \tag{3}$$

where:

- $\mathbf{z}_k$ is the measurement vector, defined as $\mathbf{z}_k = [x_{g,k}, y_{g,k}, V_{e,k}, V_{n,k}]^T$,
- $h(\mathbf{x}_k)$ is the nonlinear measurement function given by:

$$h(\mathbf{x}_k) = \begin{bmatrix} x_{g,k} \\ y_{g,k} \\ V_{x,k}\cos(\psi_k) - V_{y,k}\sin(\psi_k) \\ V_{x,k}\sin(\psi_k) + V_{y,k}\cos(\psi_k) \end{bmatrix}, \tag{4}$$

- $\mathbf{v}_k$ is the measurement noise, assumed to be Gaussian with covariance $\mathbf{R}$.

2.2 Computation of Extended Kalman Filter

2.2.1 Jacobian Computation for EKF

The Jacobian matrices $\mathbf{F}_k$ and $\mathbf{H}_k$ are required for linearizing the system dynamics and measurement model, respectively.

State Transition Jacobian ($\mathbf{F}_k$) The Jacobian of the nonlinear process model $f(\mathbf{x}, \mathbf{u})$ with respect to the state vector $\mathbf{x}$ is given by:

$$\mathbf{F}_k = \begin{bmatrix} 1 & 0 & -\Delta t\, V_y \sin(\psi_k) - \Delta t\, V_x \cos(\psi_k) & \Delta t\, \cos(\psi_k) & -\Delta t\, \sin(\psi_k) \\ 0 & 1 & \Delta t\, V_x \cos(\psi_k) - \Delta t\, V_y \sin(\psi_k) & \Delta t\, \sin(\psi_k) & \Delta t\, \cos(\psi_k) \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5}$$

Measurement Jacobian ($\mathbf{H}_k$) The Jacobian of the measurement model $h(\mathbf{x})$ with respect to the state vector $\mathbf{x}$ is:

$$\mathbf{H}_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \cos(\psi_k) & \sin(\psi_k) & -V_x \sin(\psi_k) - V_y \cos(\psi_k) & 0 & 0 \\ -\sin(\psi_k) & \cos(\psi_k) & V_x \cos(\psi_k) - V_y \sin(\psi_k) & 0 & 0 \end{bmatrix}. \tag{6}$$

2.2.2 Measurement Update for EKF

The measurement residual (or innovation) $\mathbf{y}_k$ is calculated as:

$$\mathbf{y}_k = \mathbf{z}_k - h(\mathbf{x}_k^-), \tag{7}$$

where $\mathbf{z}_k$ is the measurement vector and $h(\mathbf{x}_k^-)$ is the predicted measurement.

The innovation covariance $\mathbf{S}_k$ is computed as:

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_k^- \mathbf{H}_k^\top + \mathbf{R}. \tag{8}$$

2.2.3 Kalman Gain for EKF

The Kalman gain $\mathbf{K}_k$ is calculated as:

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}_k^\top \mathbf{S}_k^{-1}. \tag{9}$$

2.2.4 State Update for EKF

The state estimate $\mathbf{x}_k$ and covariance $\mathbf{P}_k$ are updated as:

$$\mathbf{x}_k = \mathbf{x}_k^- + \mathbf{K}_k \mathbf{y}_k, \tag{10}$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)\mathbf{P}_k^-. \tag{11}$$

Explanation:

- $\mathbf{x}_k^-$: Predicted state.
- $\mathbf{P}_k^-$: Predicted covariance.
- $\mathbf{y}_k$: Measurement residual.
- $\mathbf{S}_k$: Innovation covariance.
- $\mathbf{K}_k$: Kalman gain.
- $\mathbf{H}_k$: Measurement Jacobian matrix.
- $\mathbf{R}$: Measurement noise covariance matrix.
- $\mathbf{I}$: Identity matrix.

2.3 Computation of Unscented Kalman Filter

2.3.1 Generating Sigma Points

The sigma points are calculated as:

$$\chi_0 = \mathbf{x}_k, \quad \chi_i = \mathbf{x}_k + \sqrt{(n+\lambda)\mathbf{P}_k}, \quad \chi_{i+n} = \mathbf{x}_k - \sqrt{(n+\lambda)\mathbf{P}_k}, \quad i = 1, \ldots, n \tag{12}$$

where λ is a scaling parameter and n is the dimension of the state vector.

2.3.2 Prediction Step

The sigma points are propagated through the process model:

$$\chi_{i,k+1} = f(\chi_{i,k}, \mathbf{u}_k, \Delta t), \quad i = 0, \ldots, 2n \tag{13}$$

2.3.3 Transform Sigma Points

The transformed sigma points are used to calculate the predicted state and covariance:

$$\mathbf{x}_{k+1}^- = \sum_{i=0}^{2n} W_i^m \chi_{i,k+1}, \quad \mathbf{P}_{k+1}^- = \sum_{i=0}^{2n} W_i^c (\chi_{i,k+1} - \mathbf{x}_{k+1}^-)(\chi_{i,k+1} - \mathbf{x}_{k+1}^-)^T + \mathbf{Q} \tag{14}$$

2.3.4 Measurement Update

The predicted measurements are calculated as:

$$\mathbf{z}_{k+1}^- = \sum_{i=0}^{2n} W_i^m h(\chi_{i,k+1}) \tag{15}$$

The innovation covariance is:

$$\mathbf{S}_{k+1} = \sum_{i=0}^{2n} W_i^c (h(\chi_{i,k+1}) - \mathbf{z}_{k+1}^-)(h(\chi_{i,k+1}) - \mathbf{z}_{k+1}^-)^T + \mathbf{R} \tag{16}$$

2.3.5 Kalman Gain

The cross-covariance is calculated as:

$$\mathbf{P}_{xz} = \sum_{i=0}^{2n} W_i^c (\chi_{i,k+1} - \mathbf{x}_{k+1}^-)(h(\chi_{i,k+1}) - \mathbf{z}_{k+1}^-)^T \tag{17}$$

The Kalman gain is:

$$\mathbf{K}_{k+1} = \mathbf{P}_{xz}\mathbf{S}_{k+1}^{-1} \tag{18}$$

2.3.6 State Update

The state and covariance are updated as:

$$\mathbf{x}_{k+1} = \mathbf{x}_{k+1}^- + \mathbf{K}_{k+1}(\mathbf{z}_{k+1} - \mathbf{z}_{k+1}^-) \tag{19}$$

$$\mathbf{P}_{k+1} = \mathbf{P}_{k+1}^- - \mathbf{K}_{k+1}\mathbf{S}_{k+1}\mathbf{K}_{k+1}^T \tag{20}$$

3 Numerical Results

3.1 Simulation Setup

The numerical simulation is conducted using shared parameters for both EKF and UKF filters. These parameters include process noise covariance, measurement noise covariance, and the system's sampling time.

Process Noise Covariance Q The process noise covariance matrix is defined based on the continuous-time dynamics and converted to discrete time. It is expressed as:

$$\mathbf{Q} = \begin{bmatrix} 0.01 & 0 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0 & 0 \\ 0 & 0 & 0.0001 & 0 & 0 \\ 0 & 0 & 0 & 0.0001 & 0 \\ 0 & 0 & 0 & 0 & 0.0001 \end{bmatrix}, \tag{21}$$

where each diagonal element represents the variance of the process noise for the corresponding state variables:

- V_x: Longitudinal velocity, with variance $\mathrm{Var}(V_x) = 0.01$,
- V_y: Lateral velocity, with variance $\mathrm{Var}(V_y) = 0.01$,
- ψ: Yaw angle, with variance $\mathrm{Var}(\psi) = 0.0001$,
- x_g: Global x position, with variance $\mathrm{Var}(x_g) = 0.0001$,
- y_g: Global y position, with variance $\mathrm{Var}(y_g) = 0.0001$.

Measurement Noise Covariance R The measurement noise covariance matrix is defined based on the measurement model. It is expressed as:

$$\mathbf{R} = \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0.04 & 0 \\ 0 & 0 & 0 & 0.04 \end{bmatrix}, \tag{22}$$

where each diagonal element represents the variance of the measurement noise for the corresponding observed variables:

- x_g: Global x position (GPS), with variance $\text{Var}(x_g) = 0.25$,
- y_g: Global y position (GPS), with variance $\text{Var}(y_g) = 0.25$,
- V_e: Eastward velocity component, with variance $\text{Var}(V_e) = 0.04$,
- V_n: Northward velocity component, with variance $\text{Var}(V_n) = 0.04$.

Sampling Time Δt The sampling time for the system is defined as:

$$\Delta t = 0.1 \text{ seconds.} \tag{23}$$

Simulation Duration The simulation runs for a total duration of:

$$T = 150 \text{ seconds.} \tag{24}$$

Initial State and Covariance The initial state vector and covariance are defined as:

$$\mathbf{x}_0 = [0, 0, 0, 0, 0]^T, \quad \mathbf{P}_0 = 10 \cdot \mathbf{I}_5, \tag{25}$$

where $\mathbf{I}_5$ is a 5×5 identity matrix.

Control Inputs The control inputs are defined as:

- Longitudinal acceleration: $a_x = 0.5\,\text{m/s}^2$,
- Lateral acceleration: $a_y = 0.0\,\text{m/s}^2$,
- Yaw rate: $\omega = 0.05\,\text{rad/s}$.

3.2 Simulation with Python

After determining the simulation setup and initial values, the simulation was performed with Python.

```python
import numpy as np
import matplotlib.pyplot as plt

# Car's Parameters
# m       = 668.0
# wheel_b = 1.765
# track   = 1.450

# Sampling Time and Simulation Duration
dt       = 0.1          # [s]
sim_time= 150.0         # [s]

# Covariance Matrices
# State Noise Covariance Q
Q = np.diag([0.1, 0.1, 0.01, 0.01, 0.01])**2

# Measurement Noise Covariance R
# [ Xg, Yg, Ve, Vn ]
R = np.diag([0.5, 0.5, 0.2, 0.2])**2

def wrap_angle(angle):
    return (angle + np.pi) % (2.0 * np.pi) - np.pi
```

Listing 1: System Parameters and Auxiliary Functions

The necessary libraries introduced in the above Python part (Listing 1), the necessary definitions for the simulation were made. The noise matrix Q (line 15) and the measurement noise R (line 19) were added for the state equation. The added noise covariance matrices were mentioned earlier in equations (21) and (22). The wrapangle() function we used on line 21 is used to limit the rotation angle. It keeps the rotation angle between π and $-\pi$. The necessary radian conversion is also done.

```
1
2  def f(x, u, dt):
3      """
4      Euler composition of the continuous time model.
5      x = [Vx, Vy, psi, Xg, Yg]
6      u = [ax, ay, w]   -> IMU'dan gelen ivmeler ve yaw_rate
7      """
8      Vx, Vy, psi, Xg, Yg = x
9      ax, ay, w = u
10
11     Vx_next  = Vx + dt * (Vy*w + ax)
12     Vy_next  = Vy + dt * (-Vx*w + ay)
13     psi_next = wrap_angle(psi + dt * w)
14     Xg_next  = Xg + dt * (Vx*np.cos(psi) - Vy*np.sin(psi))
15     Yg_next  = Yg + dt * (Vx*np.sin(psi) + Vy*np.cos(psi))
16
17     return np.array([Vx_next, Vy_next, psi_next, Xg_next, Yg_next])
18
19
20 def h(x):
21     """
22     Measurement model:
23         - GPS position [Xg, Yg]
24         - ENU velocity components [Ve, Vn]
25     """
26     Vx, Vy, psi, Xg, Yg = x
27     Ve = Vx*np.cos(psi) - Vy*np.sin(psi)
28     Vn = Vx*np.sin(psi) + Vy*np.cos(psi)
29     return np.array([Xg, Yg, Ve, Vn])
```

Listing 2: Kinematic Model (State Transition) and Measurement Model

This function (in Line 2 to 17) models vehicle dynamics by discretizing a continuous-time system using the Euler discretization method. The vehicle state vector $x = [V_x, V_y, \psi, X_g, Y_g]$, representing longitudinal velocity (V_x), lateral velocity (V_y), yaw angle (ψ), and global positions X_g and Y_g, is updated using control inputs $u = [a_x, a_y, \omega]$. The longitudinal and lateral velocities are updated as

$$V_{x,\text{next}} = V_x + \Delta t(V_y \cdot \omega + a_x)$$

and

$$V_{y,\text{next}} = V_y + \Delta t(-V_x \cdot \omega + a_y),$$

respectively. The yaw angle is normalized using the formula

$$\psi_{\text{next}} = \text{wrap\_angle}(\psi + \Delta t \cdot \omega).$$

Global positions are updated based on velocities and the yaw angle as

$$X_{g,\text{next}} = X_g + \Delta t(V_x \cos(\psi) - V_y \sin(\psi))$$

and

$$Y_{g,\text{next}} = Y_g + \Delta t(V_x \sin(\psi) + V_y \cos(\psi)).$$

The updated values are returned as the next state vector:

$$x_{\text{next}} = [V_{x,\text{next}}, V_{y,\text{next}}, \psi_{\text{next}}, X_{g,\text{next}}, Y_{g,\text{next}}].$$

This function is used in the prediction step of Kalman filters to estimate the vehicle's next state based on the control inputs. In Line 20-29 function defines the measurement model $h(x)$, which maps the state vector $x = [V_x, V_y, \psi, X_g, Y_g]$ to the predicted measurement values. The inputs include x, the vehicle state vector, where V_x and V_y are the longitudinal and lateral velocities, ψ is the yaw angle, and X_g, Y_g are the global x and y positions. The outputs are the measurement vector $[X_g, Y_g, V_e, V_n]$, which includes X_g and Y_g as the GPS-measured global positions and V_e, V_n as the ENU velocity components. The eastward velocity V_e is computed as $V_e = V_x \cos(\psi) - V_y \sin(\psi)$, and the northward velocity V_n is computed as $V_n = V_x \sin(\psi) + V_y \cos(\psi)$. The function returns $h(x) = [X_g, Y_g, V_e, V_n]$, enabling comparison between predicted and actual sensor measurements during the update step of Kalman filters.

```
1  def jacobian_F(x, u, dt):
2      """
3      Calculates f(x,u)'s Jacobian.
4      x = [Vx, Vy, psi, Xg, Yg], u = [ax, ay, w]
5      """
6      Vx, Vy, psi, Xg, Yg = x
7      ax, ay, w = u
8
9      dfdx = np.zeros((5,5))
10
11     #  d(Vx_next)/dVx = 1
12     #  d(Vx_next)/dVy = dt*w
13     dfdx[0,0] = 1.0
14     dfdx[0,1] = dt * w
15
16     #  d(Vy_next)/dVx = -dt*w
17     #  d(Vy_next)/dVy = 1
18     dfdx[1,0] = -dt * w
19     dfdx[1,1] = 1.0
20
21     #  d(psi_next)/dpsi = 1
22     dfdx[2,2] = 1.0
23
24     #  d(Xg_next)/dVx = dt*cos(psi)
25     #  d(Xg_next)/dVy = -dt*sin(psi)
26     #  d(Xg_next)/dpsi= dt*( -Vx sin(psi) - Vy cos(psi) )
27     dfdx[3,0] = dt * np.cos(psi)
28     dfdx[3,1] = -dt * np.sin(psi)
29     dfdx[3,2] = dt * (-Vx*np.sin(psi) - Vy*np.cos(psi))
30
31     #  d(Yg_next)/dVx = dt*sin(psi)
32     #  d(Yg_next)/dVy = dt*cos(psi)
33     #  d(Yg_next)/dpsi= dt*( Vx cos(psi) - Vy sin(psi) )
34     dfdx[4,0] = dt * np.sin(psi)
35     dfdx[4,1] = dt * np.cos(psi)
36     dfdx[4,2] = dt * (Vx*np.cos(psi) - Vy*np.sin(psi))
37
38     return dfdx
39
40 def jacobian_H(x):
41     """
42     Calculates h(x)'s Jacobian.
43       h(x) = [Xg, Yg, Vx cos(psi)-Vy sin(psi), Vx sin(psi)+Vy cos(psi)]
44     """
45     Vx, Vy, psi, Xg, Yg = x
```

```
    dhdx = np.zeros((4,5))

    # dh/dXg = [1, 0, 0, 0, 0] -> 1,0,0,0,0
    dhdx[0,3] = 1.0   # dXg/dXg
    dhdx[1,4] = 1.0   # dYg/dYg

    # Ve = Vx cos(psi) - Vy sin(psi)
    dhdx[2,0] = np.cos(psi)
    dhdx[2,1] = -np.sin(psi)
    dhdx[2,2] = -Vx*np.sin(psi) - Vy*np.cos(psi)

    # Vn = Vx sin(psi) + Vy cos(psi)
    dhdx[3,0] = np.sin(psi)
    dhdx[3,1] = np.cos(psi)
    dhdx[3,2] =  Vx*np.cos(psi) - Vy*np.sin(psi)

    return dhdx

def ekf_predict(x_est, P_est, u):
    x_pred = f(x_est, u, dt)
    F_k = jacobian_F(x_est, u, dt)
    P_pred = F_k @ P_est @ F_k.T + Q
    return x_pred, P_pred

def ekf_update(x_pred, P_pred, z_meas):
    H_k = jacobian_H(x_pred)
    z_pred = h(x_pred)

    S = H_k @ P_pred @ H_k.T + R
    K = P_pred @ H_k.T @ np.linalg.inv(S)

    y = z_meas - z_pred
    x_upd = x_pred + K @ y
    x_upd[2] = wrap_angle(x_upd[2])

    I = np.eye(len(x_pred))
    P_upd = (I - K @ H_k) @ P_pred
    return x_upd, P_upd
```

Listing 3: EKF Auxiliary Functions

The first 38 lines creates Process Matrix F's Jacobian matrix (5). This done by introducing the state vector x and input vector u. After that, on line 40, the Jacobian matrix of H (6), the measurement function. Next, it's implemented the prediction step of the Extended Kalman Filter (EKF), which predicts the next step by using process matrix F. Later, it has been implemented the update step of EKF, uses the predicted state (11) and predicted covariance.

```
def ukf_sigma_points(x, P, alpha=1e-3, beta=2, kappa=0):
    """
    Calculates sigma points. Returns 2n+1 points.
    """
    n = len(x)
    lam = alpha**2*(n+kappa) - n
    # Cholesky
    U = np.linalg.cholesky((n+lam)*P)
    sigmas = np.zeros((2*n+1, n))
    sigmas[0] = x
    for i in range(n):
        sigmas[i+1]     = x + U[i]
        sigmas[n+i+1]   = x - U[i]
    return sigmas

```

```python
def ukf_weights(n, alpha=1e-3, beta=2, kappa=0):
    lam = alpha**2*(n+kappa) - n
    Wm = np.zeros(2*n+1)  # mean weights
    Wc = np.zeros(2*n+1)  # covariance weights

    Wm[0] = lam/(n+lam)
    Wc[0] = lam/(n+lam) + (1 - alpha**2 + beta)

    for i in range(1, 2*n+1):
        Wm[i] = 1.0/(2*(n+lam))
        Wc[i] = 1.0/(2*(n+lam))
    return Wm, Wc

def ukf_predict(x_est, P_est, u, alpha=1e-3, beta=2, kappa=0):
    n = len(x_est)
    sigmas = ukf_sigma_points(x_est, P_est, alpha, beta, kappa)
    Wm, Wc = ukf_weights(n, alpha, beta, kappa)

    sigmas_pred = np.zeros_like(sigmas)
    for i in range(sigmas.shape[0]):
        sigmas_pred[i] = f(sigmas[i], u, dt)

    # Mean Calculation
    x_pred = np.zeros(n)
    for i in range(sigmas_pred.shape[0]):
        x_pred += Wm[i]*sigmas_pred[i]
    x_pred[2] = wrap_angle(x_pred[2])

    # Covariance Calculation
    P_pred = np.zeros((n,n))
    for i in range(sigmas_pred.shape[0]):
        diff = sigmas_pred[i] - x_pred
        diff[2] = wrap_angle(diff[2])
        P_pred += Wc[i]*np.outer(diff, diff)
    P_pred += Q

    return x_pred, P_pred, sigmas_pred, Wm, Wc

def ukf_update(x_pred, P_pred, sigmas_pred, z_meas, Wm, Wc):
    n = len(x_pred)
    m = len(z_meas)

    # Apply Measurement Model on Sigma Points
    Zsig = np.zeros((sigmas_pred.shape[0], m))
    for i in range(sigmas_pred.shape[0]):
        Zsig[i] = h(sigmas_pred[i])

    # Mean of Z
    z_pred = np.zeros(m)
    for i in range(Zsig.shape[0]):
        z_pred += Wm[i]*Zsig[i]

    # Mean of Innovation
    S = np.zeros((m,m))
    for i in range(Zsig.shape[0]):
        diff_z = Zsig[i] - z_pred
        S += Wc[i]*np.outer(diff_z, diff_z)
    S += R

    # Cross-Covariance
    Pxz = np.zeros((n,m))
    for i in range(sigmas_pred.shape[0]):
        diff_x = sigmas_pred[i] - x_pred
```

```
        diff_x[2] = wrap_angle(diff_x[2])
        diff_z = Zsig[i] - z_pred
        Pxz += Wc[i]*np.outer(diff_x, diff_z)

    # Gain
    K = Pxz @ np.linalg.inv(S)

    # Update UKF
    y = z_meas - z_pred
    x_upd = x_pred + K @ y
    x_upd[2] = wrap_angle(x_upd[2])
    P_upd = P_pred - K @ S @ K.T

    return x_upd, P_upd
```

Listing 4: UKF Auxiliary Functions

The first 14 lines create random sigma points which will be used in the Unscented Kalman Filter (UKF) (12). Then, for the next 14 lines, the weights of the UKF were calculated by mean and covariance weights. Later, UKF's predict and update functions were implemented, which use predicted values and some constants a = 1e-3, b = 2, k = 0. Then, at the end, UKF's update function runs through and cross-covariances are calculated. (19,20)

```
def main():
    # Time
    t = np.arange(0, sim_time, dt)

    # ax constant 0.5, ay=0, w=0.05 constant
    ax_cmd = 0.5 * np.ones_like(t)
    ay_cmd = 0.0 * np.ones_like(t)
    w_cmd  = 0.05* np.ones_like(t)

    # Real States
    X_true = np.zeros((len(t), 5))
    x0_true = np.array([0.0, 0.0, 0.0, 0.0, 0.0])  # [Vx, Vy, psi, Xg, Yg]
    X_true[0] = x0_true

    for k in range(len(t)-1):
        u_k = np.array([ax_cmd[k], ay_cmd[k], w_cmd[k]])
        X_true[k+1] = f(X_true[k], u_k, dt)

    # Measurements (GPS -> [Xg, Yg, Ve, Vn])
    Z_meas = np.zeros((len(t), 4))
    for k in range(len(t)):
        z_clean = h(X_true[k])
        noise = np.random.multivariate_normal(np.zeros(4), R)
        Z_meas[k] = z_clean + noise

    # EKF UKF Initialization
    xEKF = np.array([0.0, 0.0, 0.0, 0.0, 0.0])
    PEKF = np.eye(5) * 10.0

    xUKF = xEKF.copy()
    PUKF = np.eye(5) * 10.0

    X_est_EKF = np.zeros((len(t), 5))
    X_est_UKF = np.zeros((len(t), 5))
    X_est_EKF[0] = xEKF
    X_est_UKF[0] = xUKF

    # Filter Loop
    for k in range(len(t)-1):
```

```
        u_k = np.array([ax_cmd[k], ay_cmd[k], w_cmd[k]])

        # --- EKF ---
        x_pred_ekf, P_pred_ekf = ekf_predict(xEKF, PEKF, u_k)
        x_upd_ekf, P_upd_ekf   = ekf_update(x_pred_ekf, P_pred_ekf, Z_meas[k
    +1])
        xEKF, PEKF = x_upd_ekf, P_upd_ekf
        X_est_EKF[k+1] = xEKF

        # --- UKF ---
        x_pred_ukf, P_pred_ukf, sigmas_pred, Wm, Wc = ukf_predict(xUKF, PUKF,
    u_k)
        x_upd_ukf, P_upd_ukf = ukf_update(x_pred_ukf, P_pred_ukf,
                                          sigmas_pred, Z_meas[k+1], Wm, Wc)
        xUKF, PUKF = x_upd_ukf, P_upd_ukf
        X_est_UKF[k+1] = xUKF
```

Listing 5: Plotting the Main Simulation and Outputs

This main function is where the operations are being done. In line 11, a true state vector is introduced, and the initial values of the true state vector to 0. Then, it's been simulated the vehicle's dynamics based on the process matrix F. Later, measurements were simulated by `Z_meas`, using eastward and northward velocities. After that, EKF's and UKF's initial state guesses and covariance matrices with high uncertainty. In the end, EKF's and UKF's function got used and calculated the outputs of those filters.

```
    # Position Error
    ekf_pos_error = np.sqrt((X_est_EKF[:,3] - X_true[:,3])**2
                            + (X_est_EKF[:,4] - X_true[:,4])**2)
    ukf_pos_error = np.sqrt((X_est_UKF[:,3] - X_true[:,3])**2
                            + (X_est_UKF[:,4] - X_true[:,4])**2)

    # Vx Error
    ekf_vx_error = X_est_EKF[:,0] - X_true[:,0]
    ukf_vx_error = X_est_UKF[:,0] - X_true[:,0]
```

Listing 6: Error Calculation for Performance

This part calculates the error in position and velocities.

```
plt.figure(figsize=(10,6))
    plt.subplot(2,1,1)
    plt.plot(t, ekf_pos_error, label='EKF Position Error')
    plt.plot(t, ukf_pos_error, label='UKF Position Error')
    plt.xlabel('Time [s]')
    plt.suptitle('Position and Horizontal Velocity Error Comparison of Filters
    vs True States')
    plt.ylabel('Position Error [m]')
    plt.legend()
    plt.grid()

    plt.subplot(2,1,2)
    plt.plot(t, ekf_vx_error, label='EKF Vx Error')
    plt.plot(t, ukf_vx_error, label='UKF Vx Error')
    plt.xlabel('Time [s]')
    plt.ylabel('Vx Error [m/s]')
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()
 plt.figure(figsize=(9,7))
```

```
21
22      # Vx
23      plt.subplot(3,1,1)
24      plt.plot(t, X_est_UKF[:,0], label='UKF')
25      plt.plot(t, X_est_EKF[:,0], label='EKF')
26      plt.plot(t, X_true[:,0],    label='Real')
27      plt.ylabel(r'$V_x$ [m/s]')
28      plt.legend()
29      plt.grid(True)
30
31      # Vy
32      plt.subplot(3,1,2)
33      plt.plot(t, X_est_UKF[:,1], label='UKF')
34      plt.plot(t, X_est_EKF[:,1], label='EKF')
35      plt.plot(t, X_true[:,1],    label='Real')
36      plt.ylabel(r'$V_y$ [m/s]')
37      plt.legend()
38      plt.grid(True)
39
40      # Psi
41      plt.subplot(3,1,3)
42      plt.plot(t, X_est_UKF[:,2], label='UKF')
43      plt.plot(t, X_est_EKF[:,2], label='EKF')
44      plt.plot(t, X_true[:,2],    label='Real')
45      plt.ylabel(r'$\psi$ [rad]')
46      plt.xlabel('time [s]')
47      plt.legend()
48      plt.grid(True)
49      plt.suptitle('Fig. 3. Vehicle state estimation - validation')
50      plt.tight_layout()
51      plt.show()
52  plt.figure()
53      plt.plot(t, X_est_UKF[:,3], label='X_UKF')
54      plt.plot(t, Z_meas[:,0],    label='X_GPS (measure)', linestyle='--')
55      plt.plot(t, X_est_EKF[:,3], label='X_EKF')
56      plt.ylabel(r'$x_g$ [m]')
57      plt.xlabel('time [s]')
58      plt.legend()
59      plt.title('Estimation of Vehicle Positioning with EKF and UKF')
60      plt.grid(True)
61      plt.show()
62      print("EKF Average Position Error =", np.mean(ekf_pos_error))
63      print("UKF Average Location Error =", np.mean(ukf_pos_error))
64      print("EKF Average Vx Error    =", np.mean(np.abs(ekf_vx_error)))
65      print("UKF Average Vx Error    =", np.mean(np.abs(ukf_vx_error)))
66
67  if __name__ == "__main__":
68      main()
```

Listing 7: Plotting Results

In line 1 Error is plotted, after the first line, on lines 23-32-41 Velocities and Yaw Rates are estimated and compared. At the end, on line 53 estimated position and plotted the results.

```
EKF Average Position Error = 0.7088804433692812
UKF Average Position Error = 0.4043353048579761
EKF Average Vx Error    = 1.9628221948033842
UKF Average Vx Error    = 2.0736729580088316
```

Figure 1: Error Values

Printed values in Figure 1 can seen in above, the average position error (x_g, y_g) and velocity error (V_x) for both the Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF). The operations performed in the code (Listing 6: Line 1-9) can be explained as follows:

The position error is computed as the Euclidean distance between the estimated (x_g, y_g) and true positions at each time step. For EKF, this is calculated as:

$$\text{ekf\_pos\_error} = \sqrt{(X_{\text{est,EKF},x} - X_{\text{true},x})^2 + (X_{\text{est,EKF},y} - X_{\text{true},y})^2}.$$

A similar computation is performed for UKF. The average position error is then obtained by taking the mean of all computed position errors:

$$\text{Average Position Error} = \text{mean(position\_errors)}.$$

The velocity error (V_x) is calculated as the absolute difference between the estimated and true longitudinal velocities at each time step. For EKF, this is expressed as:

$$\text{ekf\_vx\_error} = |V_{x,\text{est,EKF}} - V_{x,\text{true}}|.$$

Similarly, UKF's velocity error is computed in the same way. The mean of all velocity errors is then used to determine the average velocity error:

$$\text{Average Velocity Error} = \text{mean(velocity\_errors)}.$$

The printed results reveal the following insights: - The EKF average position error is 0.7088, while the UKF average position error is 0.4043. These results indicate that UKF outperforms EKF in position estimation, as it is better at handling non-linearities in the system dynamics and measurements. - The EKF average V_x error is 1.9628, whereas the UKF average V_x error is 2.0737. In this case, EKF provides slightly better performance for velocity estimation. This suggests that V_x may be more sensitive to the linearization assumptions in EKF.

Overall, these results demonstrate that UKF excels in position estimation due to its ability to handle non-linear dynamics, whereas EKF shows a slight advantage in velocity estimation. This highlights the importance of selecting the appropriate filter based on the specific requirements and characteristics of the system.

3.3 Graphical Results

Figure 2 shows the performance and comparison between EKF and UKF. As can be seen, position-wise, EKF's response oscillates compared to UKF's response. It can be seen both of them are stable, but UKF performs slightly better than EKF. On the other hand, velocity-wise, EKF performs slightly better than UKF. Since both of them worked with stochastic data for this paper, re-simulations may alter from these.

In Figure 3 show the comparison between UKF, EKF and the real value. In the top subplot, it can be seen it is a comparison of Longitudinal Axis Velocity, and can be observed that there isn't much difference between EKF and UKF filter's performances. In the middle plot, which is the Lateral Axis Velocity comparison, in this graph, it can be seen that EKF's response seems slightly better than UKF's. In the last plot, is a comparison of yaw rate (ψ). Even though a discontinuity happened in real value, both of the filters seem to adapt it as well. Since both of them worked with stochastic data for this paper, re-simulations may alter from these.

Figure 4 illustrates the estimation of vehicle positioning along the x_g axis using the EKF, UKF, and GPS measurements. The UKF (blue line) closely aligns with the GPS measurements (dashed orange line) throughout the trajectory, demonstrating its accuracy in handling non-linear dynamics. Similarly, the EKF (green line) performs comparably well in this scenario,

maintaining a close fit to the true trajectory inferred from the GPS data. However, minor deviations can be observed in both filters during the peak and steep transitions of the trajectory, which are attributed to the model assumptions and noise handling limitations. Overall, the graph highlights that both the UKF and EKF are effective in estimating the vehicle's position, with UKF offering slightly improved accuracy, particularly in scenarios with nonlinear dynamics or rapid transitions.

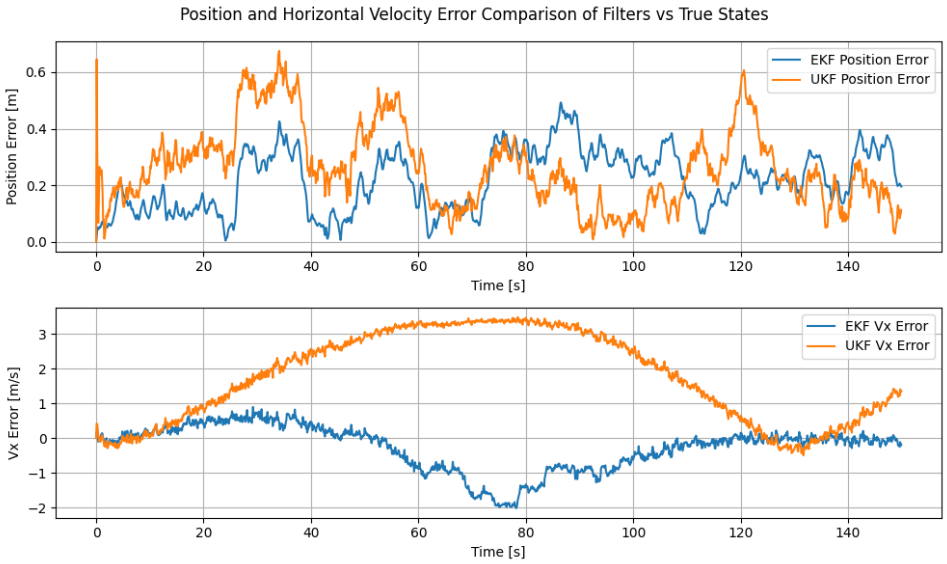


Figure 2: Position and Horizontal Velocity Error Comparison of Filters vs True States

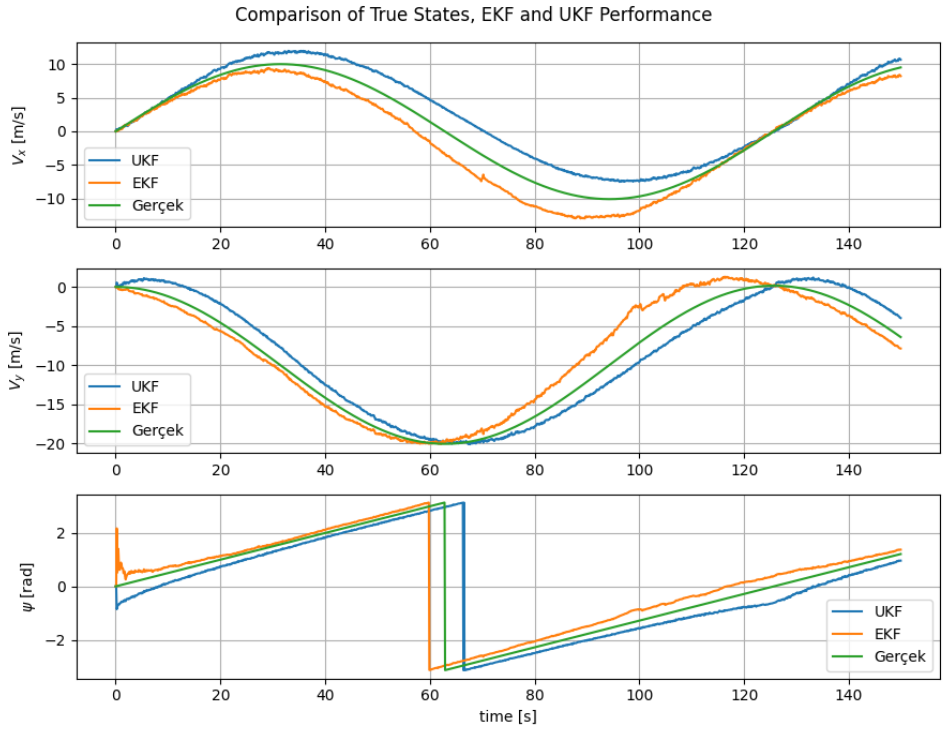


Figure 3: Comparison of True States, EKF and UKF Performance

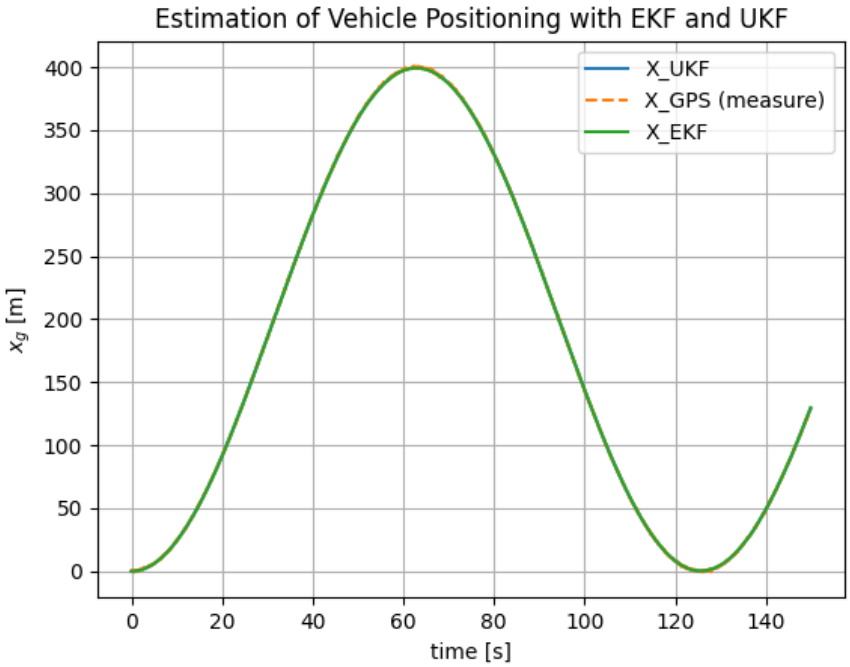


Figure 4: Estimation of Vehicle Positioning with EKF and UKF. The graph shows the trajectory estimated by EKF (green), UKF (blue), and GPS measurements (dashed orange).

4 Limitations

Extended Kalman Filter (EKF) Specific Limitations

EKF relies on linearizing the non-linear dynamics around the current estimate. For highly non-linear systems (e.g., sudden vehicle turns or changes in dynamics), the linear approximation can introduce errors. Linearization may fail if the initial estimate is far from the true state. EKF performance is highly sensitive to the tuning of process and measurement noise covariance matrices. Poor tuning can lead to divergence. Linearization errors can accumulate, causing the filter to diverge from the true state, especially during rapid accelerations or decelerations.

Unscented Kalman Filter (UKF) Specific Limitations

UKF generates sigma points and propagates them through the non-linear system, making it computationally more expensive than EKF, especially for high-dimensional systems. UKF introduces additional parameters (e.g., scaling factors for sigma points), which add complexity to the tuning process. For systems that are close to linear, UKF may not provide significant advantages over EKF and could add unnecessary computational overhead.

Specific Limitations for Vehicle GPS Applications

Both EKF and UKF often assume a simplified dynamic model (e.g., constant velocity or acceleration). These assumptions may not hold during rapid maneuvers or off-road conditions. Ignoring wheel slip or external forces (e.g., wind or slope effects) can result in inaccurate estimates. GPS typically provides updates at a lower frequency compared to vehicle dynamics. Both EKF and UKF may struggle to interpolate accurate positions and velocities between GPS updates. During GPS outages, the filters rely entirely on the process model, which can lead to significant drift in position and velocity estimates.

Additionally, it is important to note that the real GPS values were not generated in this study. Instead, GPS data were derived from other academic papers, which provided pre-processed or simulated values. While this approach ensures consistency and reproducibility, it may not fully capture the nuances of real-world GPS measurements, such as signal noise, multipath effects, or hardware-specific errors.

5 Conclusions

This paper discusses the limitations that EKF and UKF pose within GPS-based vehicle velocity and position estimation. Although both the EKF and UKF find their wide applications in navigation systems due to their great effectiveness in dealing with nonlinear systems, there are some challenges still exist.

The linearization point used in the EKF and the sensitivity of it to the initial condition and noise covariance tuning make the EKF prone to divergence, especially in highly dynamic or nonlinear scenarios. UKF offers a more robust framework for the handling of nonlinearities but involves higher computational complexity and additional tuning challenges. In both, the scope is reduced by some limiting assumptions or dependencies: assumption of Gaussian noise, simple dynamic model-based filters, GPS signal conditions regarding multipath and outage effects.

These limitations are amplified in realistic applications, which include urban areas, off-road driving, and high-speed maneuvers. Given this, a future enhancement to this work will involve the application of adaptive filtering approaches, robust rejection of outliers, and enhanced vehicle dynamic models. Again, these kinds of filter might be combined-for example, with other alternative localization approaches like LIDAR or even vision-based SLAM-operating in GPS-denied environments.

In conclusion, EKF and UKF remain the quintessential tools for navigating a vehicle, but their limitations need to be overcome to meet the demands posed by increasingly complex and dynamic environments. This research work provides the building block for a future study that might bridge the gap between theoretical performance and real-world application.

References

[1] H. Ragab, S. Khamis, and S. A. Napoleon, "Advanced Object Tracking in Self-Driving Cars: EKF and UKF Performance Evaluation," in *Proc. of NILES2024: 6th Novel Intelligent and Leading Emerging Sciences Conference*, 2024.

[2] M. Lin, J. Yoon, and B. Kim, "Self-Driving Car Location Estimation Based on a Particle-Aided Unscented Kalman Filter," *Sensors*, vol. 20, no. 9, pp. 2544, Apr. 2020.

[3] M. Bersani, M. Vignati, S. Mentasti, S. Arrigoni, and F. Cheli, "Vehicle State Estimation Based on Kalman Filters," in *Proc. IEEE EETA*, 2019.