Performance Comparison of EKF and UKF for Vehicle GPS-Based Velocity and Position Estimation

Mustafa Tursun - 19016909
Eren Topaçoğlu - 20016029
Kamil Hanoğlu - 19015022

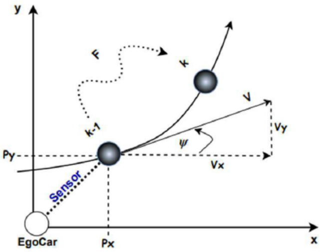
January 18, 2025

Outline

Introduction

- Accurate state estimation is essential for autonomous vehicles.
- Sensor fusion integrates GPS and IMU data.
- Kalman Filters (KF) improve robustness by handling noise and combining data.
- Objective: Compare Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF) for vehicle state estimation.

CTRV Model



- P_x, P_y: Global position coordinates of the object in the Cartesian frame.
- V_x, V_y: Velocity components in the x- and y-directions, respectively.
- ψ: Yaw angle, representing the orientation of the object relative to the x-axis.
- V: Total velocity of the object (resultant of V_x and V_y).
- $k, k-1$: Current and previous time steps in the trajectory of the object.

State Equation:

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

- $\mathbf{x}_k = [V_x, V_y, \psi, x_g, y_g]^T$
- $f(\mathbf{x}_k, \mathbf{u}_k, \Delta t)$ is the nonlinear state transition function:

State Equation

- $\mathbf{u}_k = [a_x, a_y, \omega]^T$: Control inputs.
- $\mathbf{w}_k$: Process noise with covariance $\mathbf{Q}$.

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

Measurement Equation

Measurement Equation:

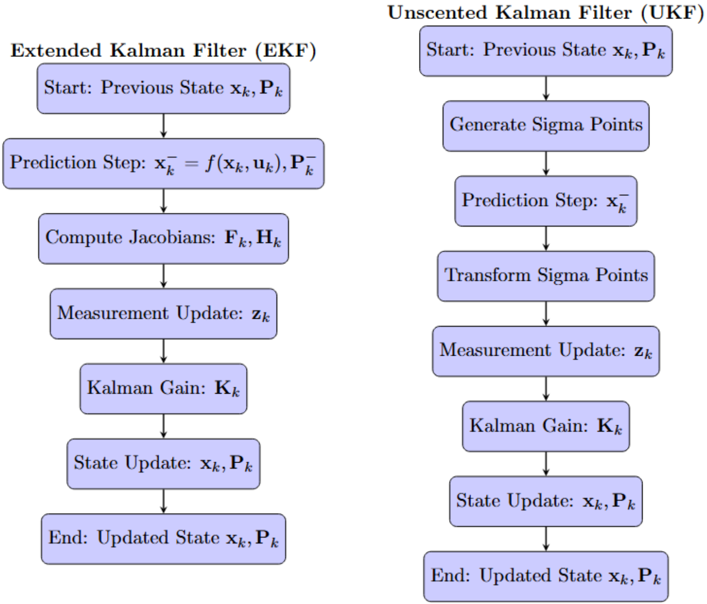
$$\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k$$

- $\mathbf{z}_k = [x_{g,k}, y_{g,k}, V_{e,k}, V_{n,k}]^T$: Measurement vector.
- $h(\mathbf{x}_k)$: Nonlinear measurement function:

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

- $\mathbf{v}_k$: Measurement noise with covariance $\mathbf{R}$.

Flowchart: EKF and UKF Algorithm



Jacobian Matrices for EKF

State Transition Jacobian $\mathbf{F}_k$:

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

Measurement Jacobian $\mathbf{H}_k$:

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

Measurement Update for EKF

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

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

- $\mathbf{y}_k$: measurement innovation or innovation
- $\mathbf{z}_k$: measurement vector
- $h(\mathbf{x}_k^-)$: predicted measurement
- $\mathbf{S}_k$: innovation covariance

Kalman Gain and State Update for EKF

Kalman Gain:

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}_k^T \mathbf{S}_k^{-1}$$

State Update:

- Updated state estimate:

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

- Updated covariance:

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

UKF: Sigma Points and Prediction

Generating Sigma Points:

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

Prediction Step:

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

Transform Sigma Points:

$$\mathbf{x}_{k+1}^- = \sum_{i=0}^{2n} W_i^m \chi_{i,k+1},$$

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

- χ_i: Sigma points.
- W_i^m, W_i^c: Weights for mean and covariance.
- $\mathbf{Q}$: Process noise covariance.

Measurement Update and Kalman Gain for UKF

Measurement Update:

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

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

Kalman Gain:

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

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

- $\mathbf{z}_{k+1}^{-}$: Predicted measurement.
- $\mathbf{S}_{k+1}$: Innovation covariance.
- $\mathbf{P}_{xz}$: Cross-covariance.
- $\mathbf{K}_{k+1}$: Kalman gain.

State Update for UKF

State Update:

$$
\begin{aligned}
\mathbf{x}_{k+1} &= \mathbf{x}_{k+1}^{-} + \mathbf{K}_{k+1}\left(\mathbf{z}_{k+1} - \mathbf{z}_{k+1}^{-}\right),\\
\mathbf{P}_{k+1} &= \mathbf{P}_{k+1}^{-} - \mathbf{K}_{k+1}\mathbf{S}_{k+1}\mathbf{K}_{k+1}^{T}.
\end{aligned}
$$

- $\mathbf{x}_{k+1}$: Updated state estimate.
- $\mathbf{P}_{k+1}$: Updated covariance matrix.
- $\mathbf{K}_{k+1}$: Kalman gain.
- $\mathbf{S}_{k+1}$: Innovation covariance.
- $\mathbf{z}_{k+1}$: Measurement vector.

Simulation Setup: Process and Measurement Noise

Process Noise Covariance (Q):

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

Measurement Noise Covariance (R):

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

- **Q**: Variances for longitudinal velocity (V_x), lateral velocity (V_y), yaw angle (ψ), and global positions (x_g, y_g).
- **R**: Variances for GPS-based positions (x_g, y_g) and ENU velocity components (V_e, V_n).

Simulation Parameters

Sampling Time (Δt):

$$\Delta t = 0.1\,\text{seconds.}$$

Simulation Duration:

$$T = 150\,\text{seconds.}$$

Initial State and Covariance:

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

Control Inputs:

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

Defining Parameters and Normalize Angle

```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

Defining Dynamic Model

```python

def f(x, u, dt):
    """
    Euler composition of the continuous time model.
    x = [Vx, Vy, psi, Xg, Yg]
    u = [ax, ay, w]  -> IMU'dan gelen ivmeler ve yaw_rate
    """
    Vx, Vy, psi, Xg, Yg = x
    ax, ay, w = u

    Vx_next  = Vx + dt * (Vy*w + ax)
    Vy_next  = Vy + dt * (-Vx*w + ay)
    psi_next = wrap_angle(psi + dt * w)
    Xg_next  = Xg + dt * (Vx*np.cos(psi) - Vy*np.sin(psi))
    Yg_next  = Yg + dt * (Vx*np.sin(psi) + Vy*np.cos(psi))

    return np.array([Vx_next, Vy_next, psi_next, Xg_next, Yg_next])


def h(x):
    """
    Measurement model:
        - GPS position [Xg, Yg]
        - ENU velocity components [Ve, Vn]
    """
    Vx, Vy, psi, Xg, Yg = x
    Ve = Vx*np.cos(psi) - Vy*np.sin(psi)
    Vn = Vx*np.sin(psi) + Vy*np.cos(psi)
    return np.array([Xg, Yg, Ve, Vn])
```

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

Defining Jacobian Function-1

```
def jacobian_F(x, u, dt):
    """
    Calculates f(x,u)'s Jacobian.
    x = [Vx, Vy, psi, Xg, Yg], u = [ax, ay, w]
    """
    Vx, Vy, psi, Xg, Yg = x
    ax, ay, w = u

    dfdx = np.zeros((5,5))

    #  d(Vx_next)/dVx = 1
    #  d(Vx_next)/dVy = dt*w
    dfdx[0,0] = 1.0
    dfdx[0,1] = dt * w

    #  d(Vy_next)/dVx = -dt*w
    #  d(Vy_next)/dVy = 1
    dfdx[1,0] = -dt * w
    dfdx[1,1] = 1.0

    #  d(psi_next)/dpsi = 1
    dfdx[2,2] = 1.0

    #  d(Xg_next)/dVx = dt*cos(psi)
    #  d(Xg_next)/dVy = -dt*sin(psi)
    #  d(Xg_next)/dpsi= dt*( -Vx sin(psi) - Vy cos(psi) )
    dfdx[3,0] = dt * np.cos(psi)
    dfdx[3,1] = -dt * np.sin(psi)
    dfdx[3,2] = dt * (-Vx*np.sin(psi) - Vy*np.cos(psi))

    #  d(Yg_next)/dVx = dt*sin(psi)
    #  d(Yg_next)/dVy = dt*cos(psi)
    #  d(Yg_next)/dpsi= dt*( Vx cos(psi) - Vy sin(psi) )
    dfdx[4,0] = dt * np.sin(psi)
    dfdx[4,1] = dt * np.cos(psi)
    dfdx[4,2] = dt * (Vx*np.cos(psi) - Vy*np.sin(psi))
```

Defining Jacobian Function / EKF Predict and Update

```python
    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

UKF Sigma Points Calculation Function

```python
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

```

UKF Weights and Predict Functions

```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
```

UKF Update

```python
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
```

UKF Update

```python
        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

Main Code and Executions

```python
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):
```

Calculation Contd.

```
        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

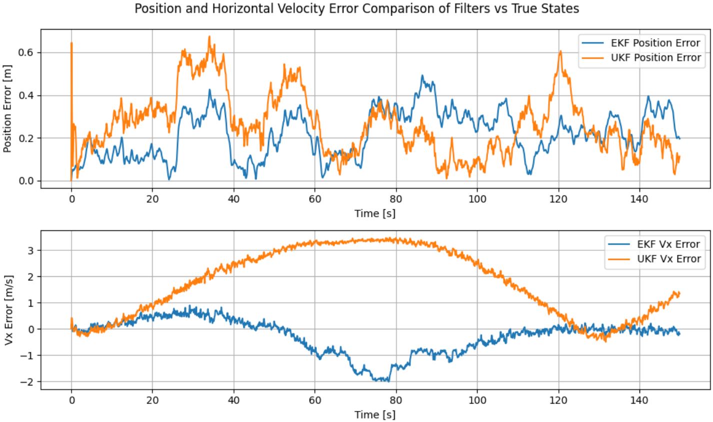
Position Error and Vx Error Execution

```
# 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]
```

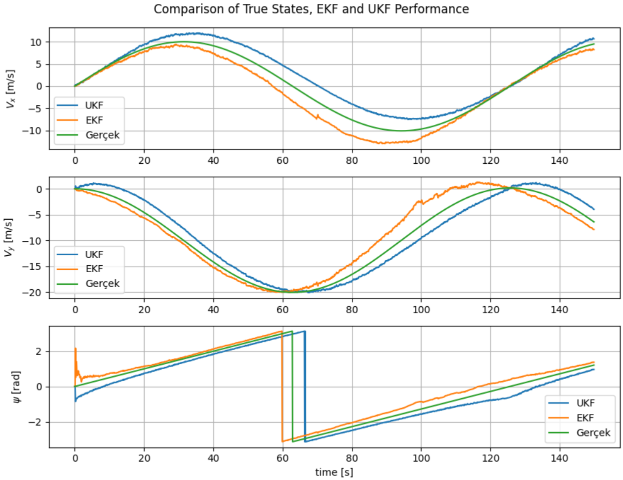
Listing 6: Error Calculation for Performance

Simulation Results: Position Error



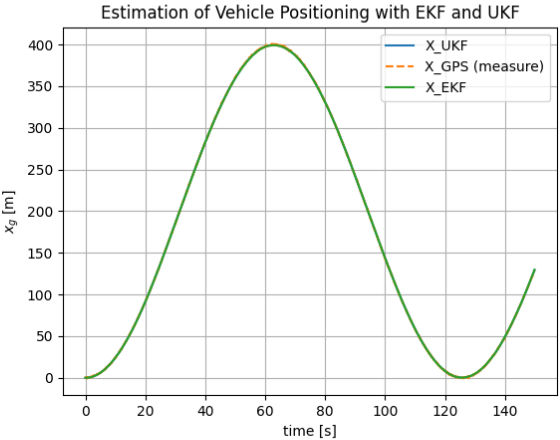
- UKF outperforms EKF in position estimation.
- EKF exhibits larger oscillations.

Simulation Results: Velocity Estimation



- Both filters perform well for velocity estimation.
- Minor differences in response to dynamic changes.

Simulation Results: Positioning



- GPS measurements align closely with UKF results.
- EKF shows slight deviations.

Error Output

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

Conclusions

- EKF and UKF are effective for vehicle state estimation.
- UKF handles nonlinearities better but is computationally intensive.
- EKF is simpler but sensitive to initialization and linearization errors.

THANK YOU FOR LISTENING

Mustafa Tursun - 19016909

Eren Topaçoğlu - 20016029

Kamil Hanoğlu - 19015022