UNIVERSITY OF MORATUWA

Faculty of Engineering

EN4594 - Autonomous Systems

Mini Project 2

Road-objects tracking using Lidar and Radar fusion

A.D.U. Dilhara

200128D

Department of Electronic and Telecommunication Engineering

Contents

Road-objects tracking using Lidar and Radar fusion

1. Introduction

Improving safety, lowering road accidents, boosting energy efficiency, enhancing comfort, and enriching driving experience are the most important driving forces behind equipping present-day cars with Advanced Driving Assistance Systems (ADAS). A critical component of the various ADAS features that are also highly required in autonomous cars is the recognition and accurate assessment of the surroundings. This component depends on data observed from sensors mounted on the ego car. If there is an object close by, it is of interest to know where that object is, what the object's velocity is, and if the object can be described by a plain geometric shape. Lidar and radar are ones of the sought-after sensors for exploiting in ADAS and autonomous-car features. A lidar always returns many concentrated detection points (point-cloud) that describe each detected object. Likewise, a radar often returns multiple detections per target but not as dense as a lidar. This means that it is necessary to group detections originating from the same target, ie to cluster the detections, to obtain information about the surroundings.

Recent lidars have a large range (up to 200 m) and a wide field of view, and can thus track objects even at big distances (necessary at high speeds) as well as in curves (i.e. very accurate in position measurement). Their main drawback is that they completely lack dynamic information about the detected objects (velocity measurement). Radar sensors, on the other hand, have a relatively narrow field of view and reduced angular resolution (i.e. less accurate in position measurement), but use the Doppler effect to directly provide velocity information. The fusion of the data from both sensors can thus benefit from the combination of their merits. Accordingly, sensor fusion of lidar and radar that com bines the strengths of both sensor types is a logical step. In this report we will discuss on fusing radar and lidar data to achieve more accurate pose data for moving objects around the ego car, proving that the EKF-based method has better performance.

2. Motion model

The state of the moving object is determined by the five variables grouped into the state vector x shown in equation (1), where P_x, and P_y are the object position in the x and y-axis respectively as shown in Figure. 1, v is the magnitude of object velocity derived from its x and y components v_x and v_y respectively. ψ is the yaw angle (object orientation) and $\dot{\psi}$ is the rate of change of the object-yaw angle.

$$x = \begin{bmatrix} P_x \\ P_y \\ v \\ \varphi \\ \dot{\varphi} \end{bmatrix}, \quad v = \sqrt{{v_x}^2 + {v_y}^2}, \;\; \varphi = tan^{-1}\left(\frac{v_y}{v_x}\right) \tag{1}$$

$P_x\,, P_y - Object\ position$

$v - Object\ velocity$

$\varphi - yaw\ angle\ (object\ orientation)$

$\dot{\varphi} - rate\ of\ change\ of\ object's\ yaw\ angle$

Using object motion model trajectory in Figure 1, we can develop following set of equations (2)-(9). We assume that object velocity and object yaw angle rate are constant within one time step with some noise variance which cause to slight changes of their values.

According to figure 1 we can say,

$$P_{x_{k+1}} - P_{x_k} = rsin\varphi_{k+1} - rsin\varphi_k \qquad (2)$$

$$P_{y_{k+1}} - P_{y_k} = rcos\varphi_k - rcos\varphi_{k+1} \qquad (3)$$

$$\varphi_{k+1} = \varphi_k + \dot{\varphi}_k \Delta t\ \ and\ \ r = \frac{v_k}{\dot{\varphi}_k} \qquad (4)$$

Using equations 2,3 and 4, derive following

$$P_{x_{k+1}} = P_{x_k} + \frac{v_k}{\dot{\varphi}_k}[\sin(\varphi_k + \dot{\varphi}_k \Delta t) - \sin\varphi_k] \quad (5)$$

$$P_{y_{k+1}} = P_{y_k} + \frac{v_k}{\dot{\varphi}_k}[-\cos(\varphi_k + \dot{\varphi}_k \Delta t) + \cos\varphi_k] \quad (6)$$

$$v_{k+1} = v_k \qquad (7)$$

$$\varphi_{k+1} = \varphi_k + \dot{\varphi}_k \Delta t \qquad (8)$$

$$\dot{\varphi}_{k+1} = \dot{\varphi}_k \qquad (9)$$

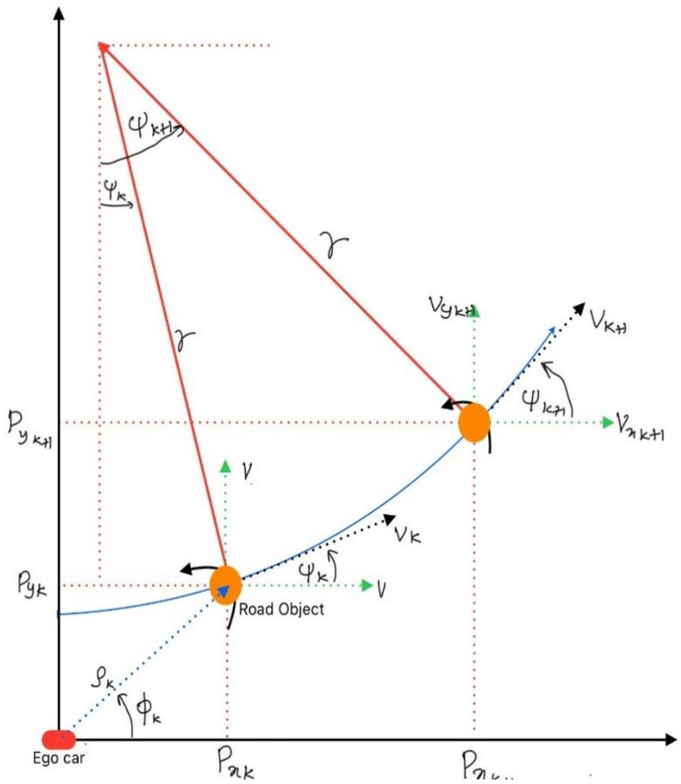


Figure 1- Motion model of an object

The nonlinear $x_{k+1} = g(x_k)$ difference equation that describes the motion model of the object is derived based on the state vector x.

$$x_{k+1} = x_k + \begin{bmatrix} \frac{v_k}{\dot{\varphi}_k}[\sin(\varphi_k + \dot{\varphi}_k \Delta t) - \sin\varphi_k] \\ \frac{v_k}{\dot{\varphi}_k}[-\cos(\varphi_k + \dot{\varphi}_k \Delta t) + \cos\varphi_k] \\ 0 \\ \dot{\varphi}_k \Delta t \\ 0 \end{bmatrix} + \varepsilon_k \qquad (10)$$

$$\varepsilon_k = \begin{bmatrix} \frac{1}{2}(\Delta t)^2 \cos(\varphi_k)\, \tau_{a,k} \\ \frac{1}{2}(\Delta t)^2 \sin(\varphi_k)\, \tau_{a,k} \\ (\Delta t)\tau_{a,k} \\ \frac{1}{2}(\Delta t)^2 \tau_{\ddot{\varphi},k} \\ (\Delta t)\tau_{\ddot{\varphi},k} \end{bmatrix}, \qquad (11)$$

Where

$\Delta t = t_{k+1} - t_k$

$\tau_{a,k} \approx N(0, {\sigma_a}^2)$

$\tau_{\ddot{\psi},k} \approx N\left(0, {\sigma_{\ddot{\psi}}}^2\right)$

$\ddot{\varphi} - yaw\ accelaration, \qquad a - longitadinal\ accelaration$

$\tau_{a,k} -\ longitudinal\ acceleration\ noise\ at\ sample\ \ k$

$\tau_{\ddot{\varphi},k} -\ yaw\ acceleration\ noise\ at\ sample\ k$

If $\dot{\varphi}$ is zero, to avoid dividing by zero (10), the following approximation is used to calculate the prediction of P_x and P_y.

$$P_{x_{k+1}} = P_{x_k} + v_k \cos(\varphi_k)\, \Delta t \tag{12}$$

$$P_{y_{k+1}} = P_{y_k} + v_k \sin(\varphi_k)\, \Delta t \tag{13}$$

Then we have find the Jacobian of the process and process covariance matrix which are required for EKF implementation.

Let

$$g(x_k) = x_k + \begin{bmatrix} \frac{v_k}{\dot{\varphi}_k}[\sin(\varphi_k + \dot{\varphi}_k \Delta t) - \sin \varphi_k] \\ \frac{v_k}{\dot{\varphi}_k}[-\cos(\varphi_k + \dot{\varphi}_k \Delta t) + \cos \varphi_k] \\ 0 \\ \dot{\varphi}_k \Delta t \\ 0 \end{bmatrix} \tag{14}$$

Jacobian of process

$$G_{k+1} = \frac{\partial g}{\partial x_k}\bigg|_{\mu_k} = \begin{bmatrix} \frac{\partial g_1}{\partial P_{x_k}} & \frac{\partial g_1}{\partial P_{y_k}} & \frac{\partial g_1}{\partial V_k} & \frac{\partial g_1}{\partial \varphi_k} & \frac{\partial g_1}{\partial \dot{\varphi}_k} \\ \frac{\partial g_2}{\partial P_{x_k}} & \frac{\partial g_2}{\partial P_{y_k}} & \frac{\partial g_2}{\partial V_k} & \frac{\partial g_2}{\partial \varphi_k} & \frac{\partial g_2}{\partial \dot{\varphi}_k} \\ \frac{\partial g_3}{\partial P_{x_k}} & \frac{\partial g_3}{\partial P_{y_k}} & \frac{\partial g_3}{\partial V_k} & \frac{\partial g_3}{\partial \varphi_k} & \frac{\partial g_3}{\partial \dot{\varphi}_k} \\ \frac{\partial g_4}{\partial P_{x_k}} & \frac{\partial g_4}{\partial P_{y_k}} & \frac{\partial g_4}{\partial V_k} & \frac{\partial g_4}{\partial \varphi_k} & \frac{\partial g_4}{\partial \dot{\varphi}_k} \\ \frac{\partial g_5}{\partial P_{x_k}} & \frac{\partial g_5}{\partial P_{y_k}} & \frac{\partial g_5}{\partial V_k} & \frac{\partial g_5}{\partial \varphi_k} & \frac{\partial g_5}{\partial \dot{\varphi}_k} \end{bmatrix} \tag{15}$$

$$G_{k+1} = \begin{bmatrix} 1 & 0 & \frac{1}{\dot{\varphi}_k} S(\varphi_k, \dot{\varphi}_k) & -\frac{v_k}{\dot{\varphi}_k} C(\varphi_k, \dot{\varphi}_k) & F(\varphi_k, \dot{\varphi}_k) \\ 0 & 1 & \frac{1}{\dot{\varphi}_k} C(\varphi_k, \dot{\varphi}_k) & \frac{v_k}{\dot{\varphi}_k} S(\varphi_k, \dot{\varphi}_k) & G(\varphi_k, \dot{\varphi}_k) \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{16}$$

where $S(\varphi_k, \dot{\varphi}_k) = -\sin(\varphi_k) + \sin(\varphi_k + \dot{\varphi}_k \Delta t)\,, C(\varphi_k, \dot{\varphi}_k) = \cos(\varphi_k) - \cos(\varphi_k + \dot{\varphi}_k \Delta t)$

$F(\varphi_k, \dot{\varphi}_k) = \frac{v_k \Delta t}{\dot{\varphi}_k} \cos(\varphi_k + \dot{\varphi}_k \Delta t) - \frac{v_k}{\dot{\varphi}_k^{\,2}} S(\varphi_k, \dot{\varphi}_k), G(\varphi_k, \dot{\varphi}_k) = \frac{v_k \Delta t}{\dot{\varphi}_k} \sin(\varphi_k + \dot{\varphi}_k \Delta t) - \frac{v_k}{\dot{\varphi}_k^{\,2}} C(\varphi_k, \dot{\varphi}_k)$

Then , the associated process noise covariance matrix(R) is given by,

$$R = \begin{bmatrix} \frac{(\Delta t)^4}{4}\sigma_{a_x}^2 & 0 & \frac{(\Delta t)^3}{2}\sigma_{a_x}^2 & 0 & 0 \\ 0 & \frac{(\Delta t)^4}{4}\sigma_{a_y}^2 & \frac{(\Delta t)^3}{2}\sigma_{a_y}^2 & 0 & 0 \\ \frac{(\Delta t)^3}{2}\sigma_{a_x}^2 & \frac{(\Delta t)^3}{2}\sigma_{a_y}^2 & (\Delta t)^2\sigma_a^2 & 0 & 0 \\ 0 & 0 & 0 & (\Delta t)^2\sigma_{\dot{\varphi}}^2 & 0 \\ 0 & 0 & 0 & 0 & (\Delta t)^2\sigma_{\ddot{\varphi}}^2 \end{bmatrix} \tag{17}$$

3. Measurment model

The received sensor raw data (either lidar or radar) is getting processed before being supplied to the EKF. The processing is performed using clustering and association algorithms. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is an unsupervised clustering algorithm that groups together datapoints if the density of the points is high enough. But in this report processing is not focused and we gave processed sensor data for EKF estimation in our simulation. The processed Lidar measurement vector includes the moving object centroid position $P_x\ and\ P_y$ in cartesian coordinates, while the radar measurement vector includes the moving object centroid position ρ, φ and radian velocity $\dot{\rho}$ in polar coordinates.

$$z_{l_{k+1}} = \begin{pmatrix} P_{x_{k+1}} \\ P_{y_{k+1}} \end{pmatrix} = h_l\ in\ cartesian\ coordinates \tag{18}$$

$$z_{r_{k+1}} = \begin{pmatrix} \rho_{k+1} \\ \varphi_{k+1} \\ \dot{\rho}_{k+1} \end{pmatrix}\ in\ polar\ coordinates \tag{19}$$

Let’s convert polar coordinates into cartesian coordinates.

$$z_{r_{k+1}} = \begin{pmatrix} \rho_{k+1} \\ \varphi_{k+1} \\ \dot{\rho}_{k+1} \end{pmatrix} = \begin{pmatrix} \sqrt{\left(P_{x_{k+1}}\right)^2 + \left(P_{y_{k+1}}\right)^2} \\ arctan\frac{P_{y_{k+1}}}{P_{x_{k+1}}} \\ \frac{P_{x_{k+1}} v_{x_{k+1}} + P_{y_{k+1}} v_{y_{k+1}}}{\sqrt{\left(P_{x_{k+1}}\right)^2 + \left(P_{y_{k+1}}\right)^2}} \end{pmatrix} = h_r \tag{20}$$

$$v_{x_{k+1}} = v_{k+1}\cos(\varphi_{k+1})\,,\ v_{y_{k+1}} = v_{k+1}\sin(\varphi_{k+1}) \tag{21}$$

Using h_l and h_r from equations (18)(20), now we can find Jocabian and measurment nosie covariance matrices for each sensor.

Lidar Jacobian and measurement noise covariance matrices are,

$$\left.\frac{\partial h_r}{\partial x_{k+1}}\right|_{\bar{\mu}_{k+1}} = H_{l_{k+1}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{22}$$

$$Q_l = \begin{bmatrix} \sigma_{P_x}^2 & 0 \\ 0 & \sigma_{P_y}^2 \end{bmatrix} \tag{23}$$

Where σ_{P_x} and σ_{P_y} are the noise standard deviations for the object x and y positions respectively.

Radar Jacobian and measurement noise covariance matrices are,

$$\left.\frac{\partial h_l}{\partial x_{k+1}}\right|_{\bar{\mu}_{k+1}} = H_{r_{k+1}} = \frac{1}{|P_{k+1}|^2}\begin{bmatrix} P_{x_{k+1}}|P_{k+1}| & P_{y_{k+1}}|P_{k+1}| & 0 & 0 & 0 \\ -P_{y_{k+1}} & P_{x_{k+1}} & 0 & 0 & 0 \\ \frac{P_{y_{k+1}} v_{k+1} P_1}{|P_{k+1}|} & -\frac{P_{x_{k+1}} v_{k+1} P_1}{|P_{k+1}|} & P_2|P_{k+1}| & v_{k+1}P_1|P_{k+1}| & 0 \end{bmatrix} \tag{24}$$

where:

$$|P_{k+1}| = \sqrt{\left(P_{x_{k+1}}\right)^2 + \left(P_{y_{k+1}}\right)^2}, P_1 = P_{y_{k+1}}\cos(\varphi_{k+1}) - P_{x_{k+1}}\sin(\varphi_{k+1}),$$

$$P_2 = P_{x_{k+1}}\cos(\varphi_{k+1}) + P_{y_{k+1}}\sin(\varphi_{k+1})$$

$$Q_r = \begin{bmatrix} \sigma_\rho^2 & 0 & 0 \\ 0 & \sigma_\varphi^2 & 0 \\ 0 & 0 & \sigma_{\dot{\rho}}^2 \end{bmatrix} \tag{25}$$

where σ_ρ is the noise standard deviation of the object radial distance, σ_φ is the noise standard deviation of the object heading(bearing), $\sigma_{\dot{\rho}}$ is the noise standard deviation of the object yaw rate.

4. Implementation of EKF

Figure 2 presents the lidar and radar data fusion technique employing the EKF. According to presentation, each sensor has its own prediction update scheme, however, both sensors share the same state prediction scheme. The belief about the object's position and velocity is updated asynchronously each time the measurement is received regardless of the source sensor. Both distinct update schemes are getting updated by any received measurement data (either from lidar or from radar). The state vector(x) is getting updated after receiving a lidar measurement vector ($z_{l_{k+1}}$) in (18) by inserting ($G_{k+1}, R, H_{l_{k+1}}$ and Q_l) equations in (15,1 7, 22, 23). Likewise, the state vector (x) is getting updated after receiving a radar measurement vector ($z_{r_{k+1}}$) in (19) by inserting ($G_{k+1}, R, H_{r_{k+1}}$ and Q_r) equations in (15, 17, 24, 25). Accordingly, the state vector(x) is the product of the fusion of then lidar and radar measurement data.

In the following sections, several important matters that have a crucial effect on the Kalman filters design process will be highlighted and discussed.

4.1 Noise parameters setting

The object motion model described by (11-14) includes several noise parameters that need to be carefully set. To set the two process noise parameters as an example: the longitudinal acceleration noise standard deviation σ_a and the yaw acceleration noise $\sigma_{\ddot{\psi}}$, one has to approximate the expected top acceleration road objects can exhibit both longitudinal and angular as an initial guess, then fine-tune these values through trial-and-error iterations. Table 1 presents the fine-tuned parameters for EKF.

Table 1 - EKF noise parameters

Parameter	Value
σ_a m/s^2	2.0
σ_{a_x} m/s^2	3.0
σ_{a_y} m/s^2	1.5
$\sigma_{\ddot{\psi}}$ rad/s^2	0.9
$\sigma_{\dot{\varphi}}$ rad/s^2	0.2
σ_{P_x} $(lidar)$ m	0.1
σ_{P_y} $(lidar)$ m	0.1
σ_ρ $(radar)$ m	0.05
σ_φ $(radar)$ rad	0.1
$\sigma_{\dot{\rho}}$ $(radar)$ m/s	0.05

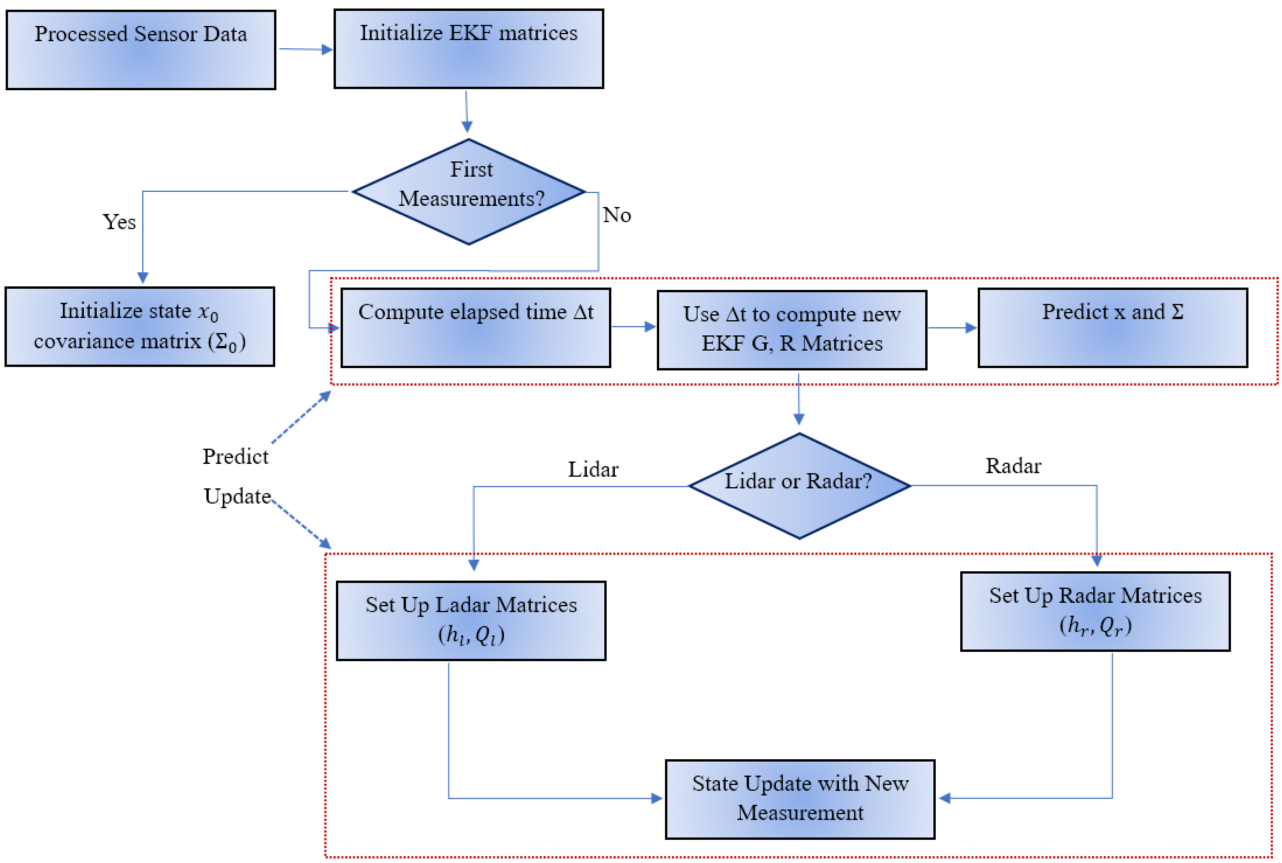


Figure 2 - Lidar and Radar data fusion using EKF

4.2 Initialization of Extended Kalman filter

The proper initialization of the Kalman filter is very crucial to its subsequent performance. The main initialized variables are the estimate state vector (x_0) and its estimate covariance matrix (Σ_0).

$$\Sigma_0 = \begin{bmatrix} \sigma_{P_{x0}}^2 & 0 & 0 & 0 & 0 \\ 0 & \sigma_{P_{y0}}^2 & 0 & 0 & 0 \\ 0 & 0 & \sigma_{v_0}^2 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{\varphi_0}^2 & 0 \\ 0 & 0 & 0 & 0 & \sigma_{\dot{\varphi}_0}^2 \end{bmatrix} \tag{26}$$

Table 2-Initialization of EKF

Parameter	Value
P_x m	1^{st} raw x reading
P_y m	1^{st} raw y reading
v m/s	0.0
φ rad	0.0
$\dot{\varphi}$ rad/s	0.0
$\sigma_{P_{x0}}$ m	1.0
$\sigma_{P_{y0}}$ m	1.0
σ_{v_0} m/s	$\sqrt{10000}$
σ_{φ_0} rad	$\sqrt{10000}$
$\sigma_{\dot{\varphi}_0}$ rad/s	$\sqrt{10000}$

The first two terms of the state vector x_0 given by (1) are P_x and P_y which are simply initialized using the first received raw sensor measurement. For the other three terms of the state vector, intuition augmented with some trial-and-error is used to initialize these variables as listed in Table 2.

The state covariance matrix is initialized as a diagonal matrix that contains the covariance of each variable estimate (Eq. (26)). The initialization logic works as follows: little or almost no correlation among the state variables (independent variables) is assumed, therefore, the off-diagonal terms (covariances between variables) are initialized to zeros. Each diagonal term represents the variance of each state element estimate as shown in (26). The variance of each element is initialized depends on the a priori information about this element. Since the first two elements of the state vector (P_x and P_y) are initialized, using the first raw reading of the sensors, then both $\sigma^2_{P_{x0}}$ *and* $\sigma^2_{P_{y0}}$ are set to small values. However, little priori information is known about the other three terms ($v,\ \varphi,\ \dot{\varphi}$), therefore, they have been initialized to large values as listed in Table 2. Note that radar velocity measurement ρ cannot directly be used to initialize the state vector velocity (object velocity v) as they are not the same.

4.3 Performance measures of Kalman filters

To check the performance of the EKF, in terms of how far the estimated results from the true results (ground truth). There are many evaluation metrics, but most common one is the root mean squared error

$$RMSE = \sqrt{\frac{1}{N}\sum_{k=1}^{k=N}(x_k^{est} - x_k^{true})^2} \tag{27}$$

The metric is calculated over a moving window of measurements of length N. Here, x_k^{est} is the estimated state vector of the EKF given in (1), and x_k^{true} is the true state vector supplied by the simulator during the EKF design phase

5. Testing and evaluation results

In this section, we will discuss first testing and evaluation results and finally code explanation used for testing.

5.1 Testing results of the filter

We did python implementation to test the EKF performance where we generate ground truth state vectors in each time step. Then using measurement noise parameters, we generate noisy measurement for lidar and radar in each time step stored in 2D array. After initializing the state (x_0) and state covariance (Σ_0), EKF predict and update using noisy process parameters and noisy measurements. Figure 3, 4, 5 and 6 present ground truth and estimated position of the object track, velocity estimation performance on given track, yaw-angle estimation performance on the track, yaw-rate estimation performance on the track.

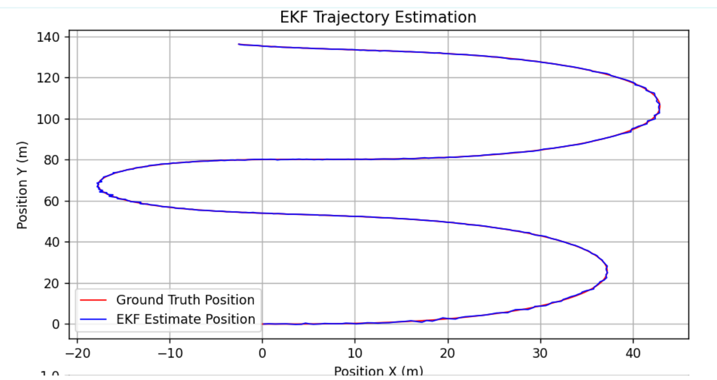


Figure 4-Ground truth and estimated position of the object track

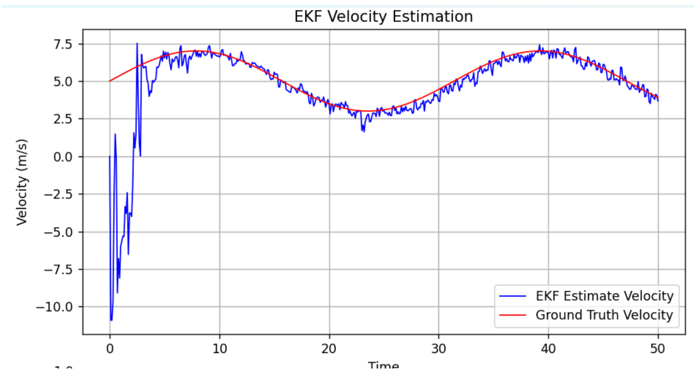


Figure 5-Velocity estimation performance on given track

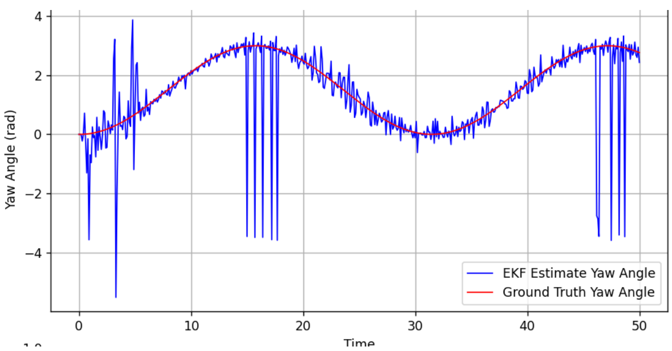


Figure 3- Yaw-angle estimation performance on the track

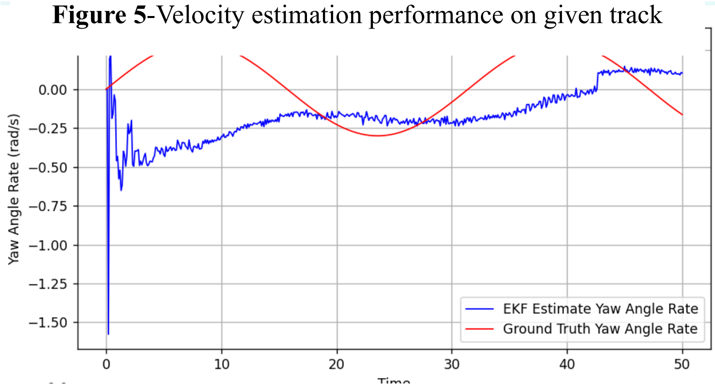


Figure 6-Yaw-rate estimation performance on the track

To check the performance of EKF for different hyper parameter values (noise parameters) we use RMSE in eq. (27). RMSE values of each state variable after tuning as follows:

Table 3- Performance of EKF

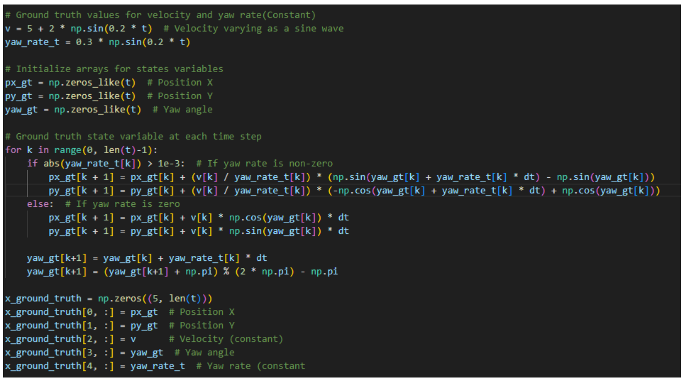
State variable	RMSE values
$\boldsymbol{P_x}$	0.1162
$\boldsymbol{P_y}$	0.0986
$\boldsymbol{v}$	0.5068
$\boldsymbol{\varphi}$	0.9084
$\boldsymbol{\dot{\varphi}}$	0.5206

5.2 Testing code explanation

In this sub section, python code going to be discussed staring from ground truth value generation to filter implementation. As libraries we used NumPy and Matplotlib for matrix calculation and visualization purpose respectively.

5.2.1 Ground truth values generation

First, we calculate the motion of the car over time using a simple model as in figure 7. It starts by defining how the velocity (v) and yaw rate ($\dot{\varphi}$) change over time as sine waves. Then, it uses a loop to update the vehicle's position (P_x *and* P_y) and orientation (φ) step by step. If the yaw rate is non-zero, a curved motion is calculated; otherwise, a straight-line motion is used. Finally, all the data, including position, velocity, yaw angle, and yaw rate, are stored in *x_ground_truth* for analysis. This provides a realistic simulation of the car's path.

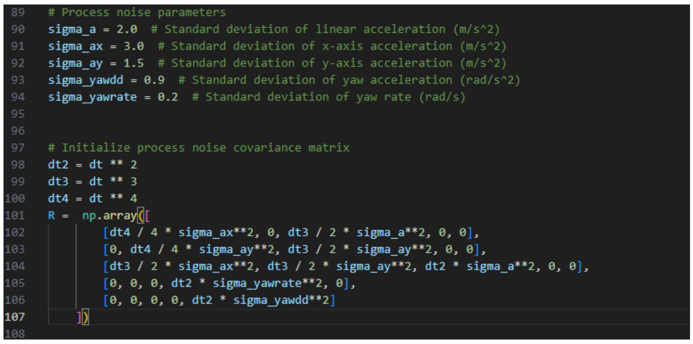


```python
# Ground truth values for velocity and yaw rate(Constant)
v = 5 + 2 * np.sin(0.2 * t)  # Velocity varying as a sine wave
yaw_rate_t = 0.3 * np.sin(0.2 * t)

# Initialize arrays for states variables
px_gt = np.zeros_like(t)  # Position X
py_gt = np.zeros_like(t)  # Position Y
yaw_gt = np.zeros_like(t)  # Yaw angle

# Ground truth state variable at each time step
for k in range(0, len(t)-1):
    if abs(yaw_rate_t[k]) > 1e-3:  # If yaw rate is non-zero
        px_gt[k + 1] = px_gt[k] + (v[k] / yaw_rate_t[k]) * (np.sin(yaw_gt[k] + yaw_rate_t[k] * dt) - np.sin(yaw_gt[k]))
        py_gt[k + 1] = py_gt[k] + (v[k] / yaw_rate_t[k]) * (-np.cos(yaw_gt[k] + yaw_rate_t[k] * dt) + np.cos(yaw_gt[k]))
    else:  # If yaw rate is zero
        px_gt[k + 1] = px_gt[k] + v[k] * np.cos(yaw_gt[k]) * dt
        py_gt[k + 1] = py_gt[k] + v[k] * np.sin(yaw_gt[k]) * dt

    yaw_gt[k+1] = yaw_gt[k] + yaw_rate_t[k] * dt
    yaw_gt[k+1] = (yaw_gt[k+1] + np.pi) % (2 * np.pi) - np.pi

x_ground_truth = np.zeros((5, len(t)))
x_ground_truth[0, :] = px_gt  # Position X
x_ground_truth[1, :] = py_gt  # Position Y
x_ground_truth[2, :] = v      # Velocity (constant)
x_ground_truth[3, :] = yaw_gt  # Yaw angle
x_ground_truth[4, :] = yaw_rate_t  # Yaw rate (constant
```

Figure 7-Ground truth values generation

5.2.2 Initialize process noise parameters and process noise covariance

As figure 8, the code sets up the process noise covariance matrix R using defined uncertainties in acceleration and yaw dynamics. The parameters represent variability in linear and angular motions, while powers of the time step (dt) scale their influence over time. The matrix is critical for systems like Kalman filters to account for noise during state estimation.

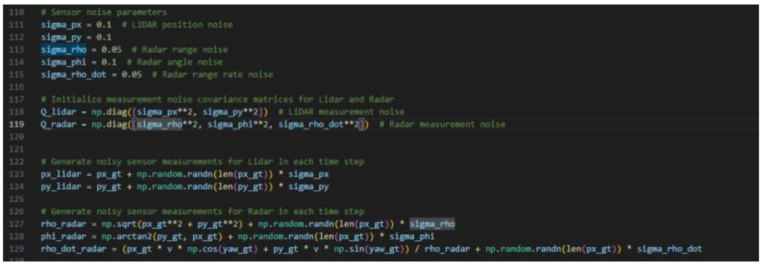


```python
# Process noise parameters
sigma_a = 2.0  # Standard deviation of linear acceleration (m/s^2)
sigma_ax = 3.0  # Standard deviation of x-axis acceleration (m/s^2)
sigma_ay = 1.5  # Standard deviation of y-axis acceleration (m/s^2)
sigma_yawdd = 0.9  # Standard deviation of yaw acceleration (rad/s^2)
sigma_yawrate = 0.2  # Standard deviation of yaw rate (rad/s)


# Initialize process noise covariance matrix
dt2 = dt ** 2
dt3 = dt ** 3
dt4 = dt ** 4
R =  np.array([
        [dt4 / 4 * sigma_ax**2, 0, dt3 / 2 * sigma_a**2, 0, 0],
        [0, dt4 / 4 * sigma_ay**2, dt3 / 2 * sigma_ay**2, 0, 0],
        [dt3 / 2 * sigma_ax**2, dt3 / 2 * sigma_ay**2, dt2 * sigma_a**2, 0, 0],
        [0, 0, 0, dt2 * sigma_yawrate**2, 0],
        [0, 0, 0, 0, dt2 * sigma_yawdd**2]
    ])
```

Figure 8-Initialize process noise parameters

5.2.3 Initialization of sensor noise parameters

As figure 9, this models sensor noise and generates noisy measurements for LiDAR and Radar. Noise parameters (sigma_px, sigma_py, etc.) define the variability in measurements. Covariance matrices (Q_lidar and Q_radar) represent these uncertainties. For LiDAR, noisy px and py positions are generated by adding random noise to the true positions. For Radar, noisy measurements of range (rho_radar), angle (phi_radar), and range rate (rho_dot_radar) are computed by adding noise to their respective true values.



```python
# Sensor noise parameters
sigma_px = 0.1  # LiDAR position noise
sigma_py = 0.1
sigma_rho = 0.05  # Radar range noise
sigma_phi = 0.1  # Radar angle noise
sigma_rho_dot = 0.05  # Radar range rate noise

# Initialize measurement noise covariance matrices for Lidar and Radar
Q_lidar = np.diag([sigma_px**2, sigma_py**2])  # LiDAR measurement noise
Q_radar = np.diag([sigma_rho**2, sigma_phi**2, sigma_rho_dot**2])  # Radar measurement noise


# Generate noisy sensor measurements for Lidar in each time step
px_lidar = px_gt + np.random.randn(len(px_gt)) * sigma_px
py_lidar = py_gt + np.random.randn(len(py_gt)) * sigma_py

# Generate noisy sensor measurements for Radar in each time step
rho_radar = np.sqrt(px_gt**2 + py_gt**2) + np.random.randn(len(px_gt)) * sigma_rho
phi_radar = np.arctan2(py_gt, px_gt) + np.random.randn(len(px_gt)) * sigma_phi
rho_dot_radar = (px_gt * v * np.cos(yaw_gt) + py_gt * v * np.sin(yaw_gt)) / rho_radar + np.random.randn(len(px_gt)) * sigma_rho_dot
```

Figure 9-Initialization of sensor noise parameters

5.2.4 EKF algorithm implementation

In this part of the code, we implement the Extended Kalman Filter (EKF) to estimate the state of the car. It starts with an initial state and covariance matrix representing the system's initial guess and uncertainty as in the figure 10. In each time step, the filter performs two key operations:

```python
# Initial state and covariance
x_t = np.array([px_lidar[0], py_lidar[0], 0, 0, 0])  # Initial state /x_0
Sigma_t= np.eye(5)  # Initial covariance / Sigma_0
Sigma_t[0][0] = 1.0
Sigma_t[1][1] = 1.0
Sigma_t[2][2] = np.sqrt(1000)
Sigma_t[3][3] = np.sqrt(1000)
Sigma_t[4][4] = np.sqrt(1000)


# Storage for EKF results
x_estimates = np.zeros((5, len(t)))
x_estimates[:, 0] = x_t
```

Figure 10-Initial state and covariance matrix

Prediction Step: The system's next state is predicted using the motion model. If the yaw rate is non-zero, the motion is curved; otherwise, it is straight. The covariance matrix is also updated using the Jacobian of the motion model and process noise. (Figure 11)

```python
for k in range(0, len(t)-1):
    x_t = x_estimates[:,k]
    Px_t = x_t[0]
    Py_t = x_t[1]
    v_t = x_t[2]
    yaw_t = x_t[3]
    yaw_rate_t = x_t[4]

    ##############Predition step
    if abs(yaw_rate_t) > 1e-3:
        Px_t_plus_1_pred = Px_t + (v_t / yaw_rate_t) * (np.sin(yaw_t + yaw_rate_t * dt) - np.sin(yaw_t))
        Py_t_plus_1_pred = Py_t + (v_t / yaw_rate_t) * (-np.cos(yaw_t + yaw_rate_t * dt) + np.cos(yaw_t))
    else:
        Px_t_plus_1_pred = Px_t + v_t * np.cos(yaw_t) * dt
        Py_t_plus_1_pred = Py_t + v_t * np.sin(yaw_t) * dt
    v_t_plus_1_pred = v_t
    yaw_t_plus_1_pred = yaw_t + yaw_rate_t * dt
    yaw_t_plus_1_pred = (yaw_t_plus_1_pred + np.pi) % (2 * np.pi) - np.pi
    yaw_rate_t_plus_1_pred = yaw_rate_t
    x_t_plus_1_pred = np.array([Px_t_plus_1_pred, Py_t_plus_1_pred, v_t_plus_1_pred, yaw_t_plus_1_pred, yaw_rate_t_plus_1_pred])

    # Jacobian of the motion model at  x_t
    G_t_plus_1 = calculate_jacobian(x_t, dt )

    # Predict covariance
    Sigma_t_plus_1_pred = G_t_plus_1 @ Sigma_t @ G_t_plus_1.T + R
```

Figure 11-Prediction Step

Update Step: Sensor measurements (alternating between Radar and LiDAR) are used to refine the predicted state. The measurement model's Jacobian and noise covariance are used to calculate the Kalman gain, which adjusts the prediction based on the observed data. (Figure 12)

The filter alternates between Radar and LiDAR updates to account for both sensor types. Results are stored at each step for further analysis. This EKF efficiently combines noisy sensor data to improve state estimation accuracy.

```python
    ############# Update step
    if k % 2 == 0:  # Alternate between radar and LiDAR updates
        rho_t_plus_1_pred = np.sqrt(Px_t_plus_1_pred**2 + Py_t_plus_1_pred**2)
        phi_t_plus_1_pred = np.arctan2(Py_t_plus_1_pred, Px_t_plus_1_pred)
        rho_dot_t_plus_1_pred = (Px_t_plus_1_pred * v_t_plus_1_pred * np.cos(yaw_t_plus_1_pred) +
                                 Py_t_plus_1_pred * v_t_plus_1_pred * np.sin(yaw_t_plus_1_pred)) / rho_t_plus_1_pred
        z_t_plus_1_pred = np.array([rho_t_plus_1_pred, phi_t_plus_1_pred, rho_dot_t_plus_1_pred])

        #Jacobian for radar measurment at x_t_plus_1_pred
        H_t_plus_1 =   radar_jacobian(x_t_plus_1_pred)
        z_t_plus_1 = np.array([rho_radar[k+1], phi_radar[k+1], rho_dot_radar[k+1]])
        Q = Q_radar
    else:
        z_t_plus_1_pred = np.array([Px_t_plus_1_pred, Py_t_plus_1_pred])
        #Jacobian for lidar measurment at x_t_plus_1_pred
        H_t_plus_1 = np.array([[1, 0, 0, 0, 0],
                               [0, 1, 0, 0, 0]])
        z_t_plus_1 = np.array([px_lidar[k+1], py_lidar[k+1]])
        Q = Q_lidar

    # Kalman gain
    S_t_plus_1 = H_t_plus_1 @ Sigma_t_plus_1_pred @ H_t_plus_1.T + Q
    K_t_plus_1 = Sigma_t_plus_1_pred @ H_t_plus_1.T @ np.linalg.inv(S_t_plus_1)

    # Update state and covariance
    x_t_plus_1 = x_t_plus_1_pred + K_t_plus_1 @ (z_t_plus_1 -z_t_plus_1_pred)
    Sigma_t_plus_1 = (np.eye(len(K_t_plus_1)) - K_t_plus_1 @ H_t_plus_1) @ Sigma_t_plus_1_pred
    # Store results
    x_estimates[:, k+1] = x_t_plus_1
```

Figure 12-Update Step

5.2.5 Visualization and measuring the performance of the filter

Purpose of this code (figure 13) the performance of the Extended Kalman Filter (EKF) through visualization and error metrics. It generates four plots comparing the ground truth and EKF estimates for position trajectory, velocity, yaw angle, and yaw rate. Also, the code calculates the Root Mean Square Error (RMSE) for all state variables $(P_x, P_y, v, \varphi, \dot{\varphi})$, providing a quantitative measure of estimation accuracy, with results printed for interpretation.

```python
axes[1, 0].plot(t, x_estimates[3, :], label="EKF Estimate Yaw Angle", color="b", linewidth="1")
axes[1, 0].plot(t, x_ground_truth[3, :], label="Ground Truth Yaw Angle", color="r", linewidth="1")
axes[1, 0].set_xlabel("Time")
axes[1, 0].set_ylabel("Yaw Angle (rad)")
axes[1, 0].set_title("EKF Yaw Angle Estimation")
axes[1, 0].grid()
axes[1, 0].legend()

# # Plot 4: EKF Yaw Angle Rate Estimation
axes[0, 0].plot(t, x_estimates[4, :], label="EKF Estimate Yaw Angle Rate", color="b", linewidth="1")
axes[0, 0].plot(t, x_ground_truth[4, :], label="Ground Truth Yaw Angle Rate", color="r", linewidth="1")
axes[0, 0].set_xlabel("Time")
axes[0, 0].set_ylabel("Yaw Angle Rate (rad/s)")
axes[0, 0].set_title("EKF Yaw Angle Rate Estimation")
axes[0, 0].grid()
axes[0, 0].legend()

# Adjust layout for better spacing
plt.tight_layout()
plt.show()


# Compute RMSE
rmse = np.sqrt(np.mean((x_estimates - x_ground_truth)**2, axis=1))
assert rmse.shape[0] == x_estimates.shape[0], "Mismatch in RMSE dimensions and state variables."
state_variables = ['px', 'py', 'v', 'yaw', 'yaw rate']

print("Root Mean Square Error (RMSE):")
for i, state in enumerate(state_variables):
    print(f"{state}: {rmse[i]:.4f}")
```

Figure 13- Visualization and measuring the performance

5.3 Evaluation of the results

According to simulation results shows in testing section (figures 3, 4, 5, 6), our implemented EKF estimate the states variables $P_x, P_y, v\ and\ \varphi$ almost nearly to ground truth value. There is very small deviation of yaw rate estimation due to noise parameter values. We run the simulation for lidar and radar separately without fusion, there we could see only radar has large deviations in results while only lidar has less deviation compare to only radar simulation (Figure 14, 15). As evaluation matrix, we use RMSE vector and following table present how far introduce fusion algorithm effective and efficient.

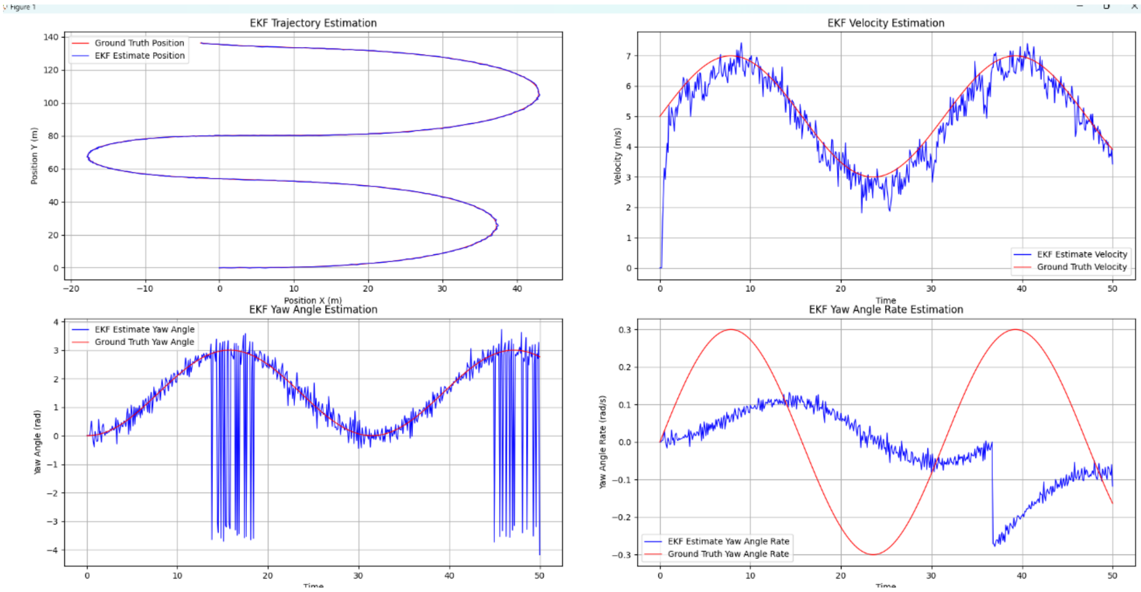


Figure 14- Simulation using only lidar

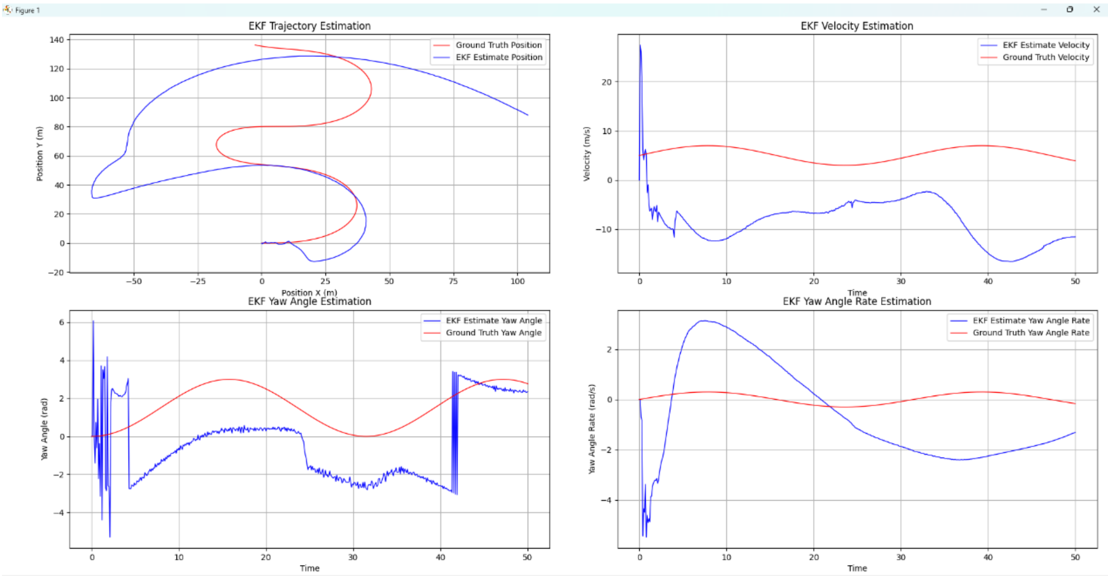


Figure 15-Simulation using only radar

Table 4-Fusion evaluation of the EKF

	Lidar + Radar (Fusion)	Only Lidar	Only radar
RMSE – P_x	0.1162	1.2208	34.6479
RMSE – P_y	0.0986	0.8021	15.9605
RMSE – v	0.5068	1.6508	4.9106
RMSE – φ	0.9084	1.4075	3.4135
RMSE – $\dot{\varphi}$	0.5206	0.2380	7.1965

6. Conclusion

The usefulness of combining lidar and radar sensor data with an Extended Kalman Filter (EKF) for object tracking in Advanced Driving Assistance Systems (ADAS) is effectively illustrated in this research. When compared to using either sensor alone, the EKF performs noticeably better at estimating state variables like position, velocity, yaw angle, and yaw rate by utilizing the complementary strengths of both sensors—radar's precise velocity measurement through the Doppler effect and lidar's high positional accuracy.

The fusion algorithm's durability and dependability in dynamic contexts are highlighted by simulation results, which confirm that it consistently produces lower RMSE values for all state variables. Notably, lidar outperformed radar alone but still behind the fusion technique, while radar alone showed significant discrepancies in location and velocity estimation. The integrated approach is a critical development for improving situational awareness in autonomous and assisted driving systems since it works very well, particularly in reducing noise-induced uncertainty. For even higher accuracy and dependability, future research could investigate additional noise parameter optimization and the combination of several sensor kinds.

7. Reference

Farag, Wael. "Road-objects tracking for autonomous driving using lidar and radar fusion." Journal of Electrical Engineering 71.3 (2020): 138-149.

R. O. Chavez-Garcia and O. Aycard, "Multiple Sensor Fusion and Classification for Moving Object Detection and Tracking", IEEE Transactions on Intelligent Transportation Systems.

M. C. Best and K. Bogdanski, "Extending the Kalman filter for structured identification of linear and nonlinear systems", Int. J. Modelling, Identification, and Control.