SDP II - Final Report

Senior Design Project II - ELEN 498

Vehicle Position & Speed Measurement Using Radar and Camera Modules

Group 8

Shayma Mohammed Alteneiji | 100063072

Engy Emad Farouk | 100061659

Maryam Ahmed Alhmoudi | 100062844

Hind Ibrahim Albastaki | 100060327

Project Advisors:

Dr. Omar Alzaabi

Dr. Ahmed Altunaiji

Dr. Mohamed Alkhatib (UAEU)

Dr Aleksej Makarov, Vlatacom Tech

Date: 23/11/2025

Abstract

This project develops an asynchronous radar–camera data fusion system for multi-target tracking using an Extended Kalman Filter (EKF), with the additional objective of improving tracking robustness under foggy conditions through fog-level classification using a random forest model. Object-level fusion is employed, where radar and camera sensors independently perform object detection and their measurements are integrated by the EKF.

Due to the multi-rate nature of the sensors, an asynchronous fusion architecture with explicit handling of out-of-sequence measurements is implemented. Measurement-to-track association is performed using the Global Nearest Neighbour (GNN) algorithm with Mahalanobis gating. Ground-truth target states, extracted from ArUco markers, are used to quantitatively evaluate tracking accuracy.

While camera-based measurements could not be reliably incorporated due to calibration-related projection errors, the radar-only EKF tracker achieved promising performance, with root mean square error (RMSE) values as low as 0.5 m in tested scenarios. The GNN data association performed reliably in simple scenes but exhibited limitations in more complex scenes. Future work includes correcting the camera projection pipeline to enable full fusion and evaluate the impact of the ML model on improving system robustness.

Table of Content

Acknowledgment

We would like to express our sincere appreciation to all those who supported and guided us throughout our Senior Design Project.

We extend our deepest gratitude to Dr. Omar Alzaabi for his guidance and support across all aspects of the project. We also thank Dr. Ahmed Altunaiji for his support and encouragement.

Our sincere appreciation goes to Dr. Mohamed Alkhatib, whose expertise and thoughtful guidance were especially impactful this semester. His insights shaped many of our approaches.

We are also grateful to our external advisor from Vlatacom Tech, Dr. Aleksej Makarov, for introducing us to this project topic and for providing valuable technical directions.

Finally, we gratefully acknowledge the support of the Electrical Engineering Department at Khalifa University, whose resources and facilities enabled us to carry out this work.

Introduction

In this section, we introduce the project's goals and objectives. We also provide the technical background of the project, and finally, we outline the structure of the report.

Goals & Objectives & Overview of the Technical Area

This project aims to develop a camera–radar data fusion system with applications in Advanced Driver Assistance Systems (ADAS) [1,2]. Fusion is needed because each sensor compensates for the weaknesses of the other, as illustrated in Figure 1. The camera reliably detects objects under normal lighting but degrades in low-light or adverse weather conditions. Radar, on the other hand, maintains consistent performance in fog, rain, and low-light environments and can detect objects at longer ranges [1,2].

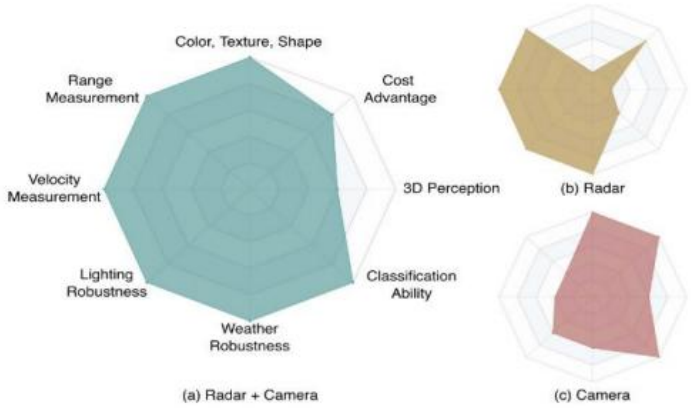


Figure 1. Comparing Radar and Camera Characteristics. (a) Radar – Camera fusion Characteristics. (b) Radar Characteristics. (c) Camera Characteristics [1].

A Raspberry Pi is used as the processing unit due to its affordability and accessibility. Although many recent fusion and object-detection approaches rely on deep neural networks [1,3–7], these methods were deemed impractical for this project due to the Raspberry Pi's limited computational capability and the available time. For this reason, the project focuses on three classical but computationally efficient algorithms: the Extended Kalman Filter (EKF) [1,2,7], Bayesian method [1,6], and the Hungarian Algorithm [1,5]. These algorithms perform object-level fusion, where radar and camera detections are independently generated and then cross-validated to produce a unified output, as shown in Figure 2.

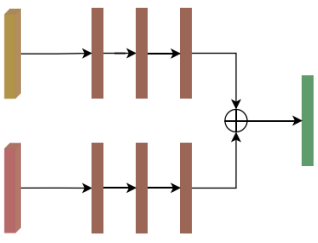


Figure 2. Taxonomy of radar – camera fusion at the object level [1].

Building on this foundation, the project also explores improving system performance under foggy conditions. A classifier machine leaning (ML) model is integrated to estimate fog levels, which are then used to evaluate the confidence on the received camera data; hence improving target tracking accuracy. Fusion performance is evaluated using the Root Mean Square Error (RMSE) between the ground-truth states and the tracker-estimated states for radar-only, camera-only, and fused outputs.

The required deliverables as follows:

1. Interfacing between the TI IWR6843 (Radar module) and the Raspberry Pi.
2. Optimized YOLOv11 using the Coral Edge TPU.
3. Camera intrinsic calibration and radar–camera extrinsic calibration.
4. Comparison matrix of fusion algorithms, to decide the optimal approach.
5. Theoretical foundation on the selected fusion algorithm.
6. Offline MATLAB code for radar–camera data fusion and a data association method for track formation.
7. Design the setup and code for obtaining ground-truth states to benchmark fusion results.
8. Improve system robustness under foggy conditions by integrating a ML classifier model that estimates fog level used to adjust the camera noise covariance matrix accordingly.
9. Evaluate and document system performance, including RMSE of radar-only, camera-only, and fused tracks under clear and foggy conditions.

Overview of the report

The report begins with an overview of the system design and technical background. It then presents the theoretical foundations of the fusion algorithms and describes the individual project components. This is followed by the methodology and implementation details, including calibration procedures, object detection, data association, and machine learning model for fog estimation. Finally, the results, analysis, and potential improvements are discussed.

System Overview

This section presents a high-level overview of the system. Design choices, including the selection of the EKF as the fusion algorithm, the unified point of view (PoV), and the fusion architecture, are summarized here and justified in detail in subsequent sections. Figure 3 illustrates the camera–radar data fusion system block diagram, integrating blocks from the EKF fusion algorithm and incorporates the adopted design decisions.

The radar module employs DSP and accelerator units for real-time processing, outputting high-level processed data. This data is then parsed, that is, converted from an unstructured format into a structured form suitable for analysis. The parsed output includes object coordinates and speed measurements. On the Raspberry Pi, through multithreading, object detection on the camera module output is performed simultaneously, generating bounding boxes, centroids, and confidence scores for each detected object.

By default, the radar module outputs measurements in bird's-eye view (BEV) [1,2], providing a top-down perspective of the scene. The camera module, in contrast, captures data in the front view (FV). To align both perspectives and represent all detected targets within a unified frame, coordinate transformation is applied to the *camera* data using the intrinsic and extrinsic parameters obtained from offline intrinsic calibration and radar–camera extrinsic calibration. Therefore, unifying the PoV to that of the radar's (BEV).

Next, temporal alignment is performed to synchronize sensor measurements by their arrival times. The pipeline is triggered upon the arrival of a sensor measurement. If both sensors capture data corresponding to the same frame within a small delay window, the Extended Kalman Filter (EKF) update is performed using the combined measurement from both sensors. Otherwise, the EKF update is executed using a single-sensor measurement (camera-only or radar-only).

Once the update flag is determined (combined, camera-only, or radar-only), the EKF prediction step is executed, followed by data association using the Global Nearest Neighbour (GNN) method for measurement-to-track matching. Subsequently, track management logic layer, including initialization and deletion are carried out.

The next stage employs an integrated Random Forest model, which extracts features from the camera frames to classify the fog level. Based on the fog level, the camera measurement covariance matrix is scaled accordingly, effectively reducing the influence of the camera measurement in the EKF state estimation step under foggy conditions. This process repeats for each new incoming measurement. Finally, tracks that accumulate more than some threshold value of assigned measurements are confirmed and displayed. In the following section the system requirements are summarized.

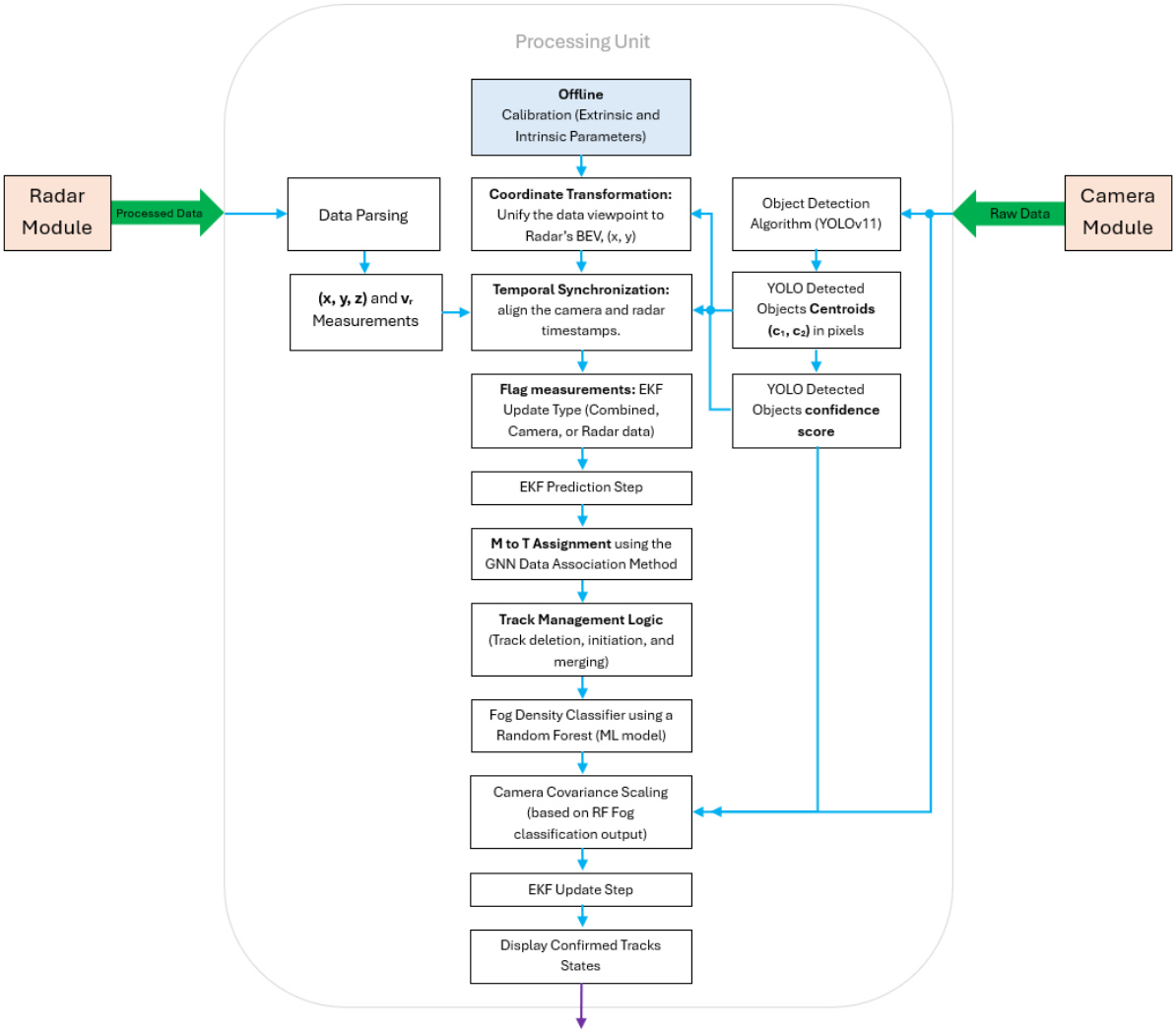


Figure 3. High-level overview of the data fusion system.

Requirements

The system requirements are summarized as follows:

- Object detection and speed measurement using the radar module.
- Object detection and classification or categorization using the camera module.
- Radar data parsing to obtain the high-level data, including 3D positions, speed, and number of detected objects.
- Offline camera intrinsic calibration and radar–camera extrinsic calibration.
- Coordinate transformation, which is the process of unifying the sensors' coordinate frames using intrinsic and extrinsic parameters.
- Implementing data fusion to using the EKF to produce the tracks of the tracked targets using the rich data of the camera and radar sensor modules.

System Specification

The system specifications are discussed in this section, The details of the data fusion system, including the components and methods, are listed below.

- The radar module employed is the TI IWR6843 radar module. It uses DSP and accelerator units for real-time processing and outputs high-level processed data (3D positions and speeds of detected objects). The radar has a horizontal field of view (FoV) of 120°.
- The camera module used is an HQ IR-Cut Camera with a horizontal field of view (FoV) of 65°.
- The camera object detection algorithms considered include the YOLO algorithm, FOMO algorithm, and Background Subtraction Algorithm (BSA). Later analysis in the next sections favours the use of YOLO.
- Coordinate transformation is performed using the obtained intrinsic and extrinsic parameters. The transformation unifies the sensor data into the bird's-eye view (BEV).
- The data generation rates of the two sensors differ, requiring careful processing of the sensors timestamps and design of the offline data processing pipeline.
- Several data fusion algorithms were evaluated, including the Hungarian algorithm, Bayesian methods, and the Extended Kalman Filter (EKF). The selected method is the EKF algorithm; the rationale for this selection is detailed in the Different Design Options and Assessment section.

Theoretical Foundations: The Engineering Model

In this chapter, we present the theoretical foundations underlying the various blocks of the project. We begin with the radar measurement model, the camera projection model, and the mathematical formulation of intrinsic and extrinsic calibration. We then introduce the theory of coordinate transformation that unifies the two sensor PoVs.

Next, we explore the system state-space model, followed by an analysis of observability. The Kalman Filter and Extended Kalman Filter (EKF) are then derived, including their assumptions. After that, we present the IEEE standards and regulations considered for this project.

Details of the Theoretical Model

The theoretical details and underlying concepts of the various blocks of the project model are presented in the subsections below.

Radar Model

The radar module employed, the TI IWR6843, uses frequency modulated continuous wave (FM-CW) signals. The diagram of the processing chain implemented within the module is shown in Figure 4. The ADC raw radar data first undergoes the range and Doppler FFTs, which produce a 2D radar tensor (range–Doppler). This is then followed by the Constant False Alarm (CFAR), a non-coherent detection algorithm used to extract signals due to target returns and suppress random noise spikes or false alarms [8]. This step converts radar data from tensor form to point cloud form [8].

Following that, the angle FFT is applied to provide the direction or angle information with respect to the radar for each of the detected objects. At this stage, the radar data has a 3D form (range–Doppler–azimuth). Next, clustering, tracking, and classification algorithms are applied to the radar point cloud data points to produce a single point representative of each target [8].

These blocks represent the stages raw radar data undergoes in the DSP unit to output the 3D positions and speeds of real-world targets, thereby directly providing processed data to the Raspberry Pi via the UART port. The second major component of the system, the camera and its projection model, is discussed next. Before that, we clarify the need for radar–camera spatial alignment, which is addressed in the following subsection.

Calibration, Coordinate Transformation, and the Data Fusion Algorithm

In data fusion algorithms, the predictions or measurements from multiple sensors are combined to produce a more reliable output with richer information. In radar–camera fusion, the algorithm also works to confirms that both sensors detect the same object at the *same position*, to reduce the chance of false detections.

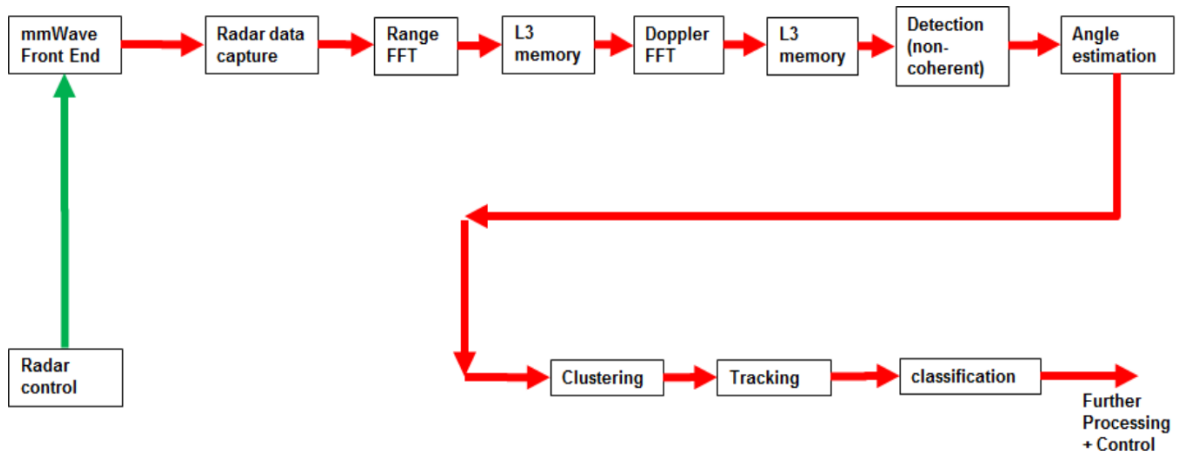


Figure 4. Typical mmWave radar processing chain [8].

But this introduces a key challenge: each sensor has a different point of view and uses a different coordinate frame. The camera uses a front view (FV) and treats its optical centre as the origin, with its axes defined as z-forward, y-down, and x-right. The radar, on the other hand, uses a bird's-eye view (BEV), with the radar at the origin and axes defined as z-up, y-forward, and x-right.

Hence, even if the radar and camera are placed as close as possible to maximize the overlap between their fields of view (FoV), there will always be some translational and rotational offset between the sensors' coordinate frames, and the same targets viewed by each sensor would appear at different positions.

The mechanism required to solve this issue should, in a way, rotate and translate the coordinate frames of one sensor such that the origin points and viewpoint are aligned. Then, cross-validation between the detected objects' positions by both sensors can be reliably performed. This is where calibration comes in. It ensures reliable spatial alignment of the two sensors data.

The intrinsic camera calibration allows us to map 3D points in the camera's frame to their respective 2D pixel coordinates. The extrinsic calibration between the radar and camera enables us to project the radar 3D coordinate points into the camera's frame, or vice versa, using inverse projection.

Hence, once the calibration parameters are applied, the coordinates can be unified to either the camera's FV or the radar's BEV, the data points are spatially aligned, and object cross-validation can be performed. In this report, we obtain these parameters offline. They will be used in real-time to continuously project the generated set of one sensor's data points into the chosen reference frame. In the next section we introduce the projective transformation theory and coordinate frame transformation.

Projective Transformation

In the previous section, we touched on how extrinsic parameters allow us to transform perspectives from one sensor's point of view to another, as illustrated by figure 5. We also discussed how intrinsic parameters enable the mapping from 3D camera coordinates to 2D image plane coordinates, as illustrated by figure 6. These two operations together define what is known as the projective transformation [9, 10].

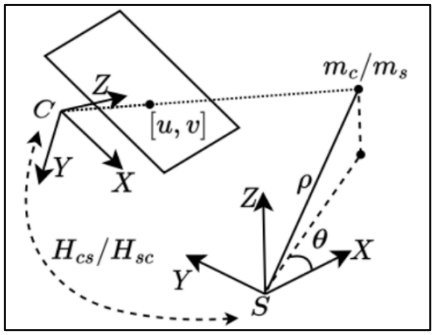


Figure 5. Extrinsic parameters used to transform from 3D camera coordinate frame to 2D image plane [10].

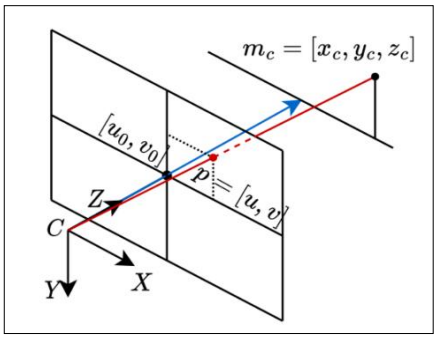


Figure 6. Intrinsic parameters used to transform from 3D camera coordinate frame to 2D image plane [10].

The general expression for this transformation in the case of perspective projection onto the image plane, is given below, where (X, Y, Z) is a real-world coordinate point [9, 11]:

$$k\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = P_{3\times4}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

When the world coordinate points all lie on the same plane ($Z = z_{projection}$), we have a special case of this transformation called homography or plane-to-plane transformation [9, 11], expressed as:

$$k\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = H_{3\times3}\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Matrix multiplication performs a linear coordinate transformation; where the point of origin remains unchanged, while the coordinate system's axes and basis vectors only change. Therefore, by multiplying the coordinates (X, Y, Z) by the appropriate matrix ($P_{3\times4}$), we can transform the view from the perspective of the real-world coordinate (specifically, with respect to some chosen origin point) to that of the camera's perspective in (u, v). The point (u, v) is measured with respect to the optical centre or the centre of the image plane, given by (c_x, c_y).

Matrix multiplication, however, does not allow us to also shift the origin point. This is resolved through the use of homogeneous coordinate system. It allows full perspective transformation, whether it is translation, rotation, or scaling using a single matrix multiplication [9].

In homogeneous coordinate systems, we represent the Euclidean (real-world) coordinate points (whether 2D or 3D) by appending an extra element, typically a 1, extending the Euclidean space R^n into a projective space $P^n = R^{n+1}$ [9]. Moreover, two points that are scalar multiples of each other, are considered equivalent, $(x, y, 1) \equiv (kx, ky, k)$ [9]. Dividing by k allows us to easily go back to the original form, where k is a non-zero scalar. With this property, homogenous coordinates allow us to model "parallel lines meeting at infinity" by defining the special case when k = 0 [9].

It is important to note that projective transformations do not preserve geometric shapes or parallelism. A circle may become deformed into an ellipse, and parallel lines can appear to intersect at a finite point. This occurs because, in projective transformation, all 3D points, even those with a zero as the last coordinate, are treated equally and get mapped to some set of finite points. This results in the loss of parallelism, as there is no explicit case to define a line at infinity in 2D projective space or a plane at infinity in 3D when k = 0 [9].

In our application, we are primarily interested in the camera's intrinsic parameters, rather than the full projective transformation matrix which encodes both the intrinsic and extrinsic parameters. Since the intrinsic parameter matrix K is an upper triangular matrix, and the extrinsic parameter matrix is orthonormal, we can extract both from the full $P_{3\times4}$ matrix using QR factorization [12]. In the next section, we delve deeper into the derivation of the intrinsic matrix K, specifically for the pinhole camera model approximation.

Camera Model

A camera model describes the mapping from 3D world coordinates into 2D pixels coordinates [9]. While multiple models exist, each applies depending on the field view of the used camera. When the field of view is below 95°, the pinhole model is typically sufficient [13]. The full expression of the model is given below. The K matrix includes the intrinsic parameters, while [R | t], describe rigid body transformation or the extrinsic parameters matrix.

$$s\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = K_{3\times4}[R|t]_{4\times4}\begin{bmatrix} X \\ y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}\begin{bmatrix} X \\ y \\ Z \\ 1 \end{bmatrix}$$

To derive the elements of matrix K, we consider the following. The model assumes the camera has no lenses but instead a simple aperture, where rays from 3D points converge and pass through to form an inverted 2D image. The point at which the rays converge is called the centre of projection [14].

The model defines the mapped (u, v) pixel point to be collinear with the centre of projection and the corresponding 3D point (X_c, Y_c, Z_c) from the camera coordinate frame [9, 14]. In computer vision, it is

more convenient to place the image plane in front of the projection centre to avoid dealing with a negative focal length. This pinhole model geometry is shown in Figure 7.

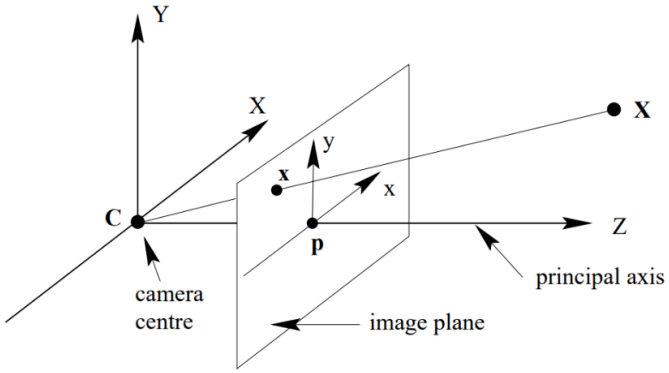


Figure 7. Pinhole model geometry with the image plane before the centre of projection point (C). p is the principal point. [9]

We can derive the coordinates of the (u, v) pixel corresponding to a given (X_c, Y_c, Z_c) point in terms of f using Figure 8. The figure illustrates how we can exploit the properties of similar triangles to obtain the expression. We can express u, and v as shown below. D is a scaling factor used to convert the focal length metric units into pixels [15]:

$$\begin{bmatrix} u_i \\ v_i \end{bmatrix} = \frac{D \times f}{Z_c} \begin{bmatrix} X_{i,c} \\ Y_{i,c} \end{bmatrix}$$

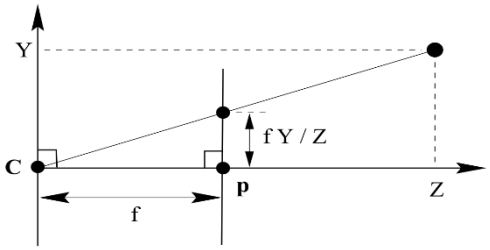


Figure 8. Similar tringles showing how derive the expression for the (u, v) pixel coordinates. [9]

The equation above presumes that the centre of the image plan is at the principal point **p** [9, 15] (Figure 7), in practice however this is not the case. It is actually at the upper left corner of the image plane [15]. Hence, to shift the origin to be the principal point, we shift the u coordinates of all points by c_x and the v points by c_y as shown below.

$$\begin{bmatrix} u_i \\ v_i \end{bmatrix} = \frac{D \times f}{Z_c} \begin{bmatrix} X_{i,c} \\ Y_{i,c} \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \end{bmatrix}$$

It was stated above that homogenous coordinates allow us to perform multiple linear transformations simultaneously using a single matrix multiplication. The following equation illustrates the resulting

matrix K. The reader may attempt to prove the equivalence of the two expressions (note that w_i is the scaler number k).

$$\begin{bmatrix} u_i w_i \\ v_i w_i \\ w_i \end{bmatrix} = \begin{bmatrix} D \times f & 0 & c_x & 0 \\ 0 & D \times f & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} X_{i,c} \\ Y_{i,c} \\ Z_{i,c} \\ 1 \end{bmatrix}$$

In practice, there is a small difference between the focal length in the x and y directions due to lens effects. Hence, instead we define f_x and f_y separately [15]. Also, element (1, 2) of the intrinsic matrix K can be non-zero and is used to model the skew between the camera x and y axes [15]. The final form of the intrinsic matrix K then becomes:

$$K = \begin{bmatrix} D \times f_x & skew & c_x & 0 \\ 0 & D \times f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The resulting model is referred to as the linear pinhole model [15]. The model is relatively simple, but it is not sufficient when high accuracy is required [15]. For that, we take into consideration (1) the radial distortions which are due to the bending of light near the edges more so than at the centre. (2) We also account for the tangential distortions which occur when the camera lens and the image 2D plane are not parallel [14, 15].

Typically, three parameters $(k_1, k_2, and\ k_3)$ are sufficient to model the radial distortion, while two parameters $(p_1,\ and\ p_2)$ model the tangential distortion [14]. The resulting model then becomes as expressed below. In the following subsection, we discuss the mathematical formulation of the extrinsic or rigid body transformation between the radar – and camera, aligning the sensor modules coordinate frame.

$$\begin{bmatrix} u_i w_i \\ v_i w_i \\ w_i \end{bmatrix} = \begin{bmatrix} D \times (f_x + \delta u_i^{(r)} + \delta u_i^{(t)}) & skew & c_x & 0 \\ 0 & D \times (f_y + \delta v_i^{(r)} + \delta v_i^{(t)}) & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} X_{i,c} \\ Y_{i,c} \\ Z_{i,c} \\ 1 \end{bmatrix}$$

Radar- Camera Extrinsic Calibration

This section discusses the rigid body transformation required to align the radar and camera coordinate frames. The radar used in this project is the TI IWR6843, which outputs 3D data with the axes defined as: x – right, y – forward, and z – up. As stated above, the camera typically follows the order as x – right, y – down, and z – forward. We model the transformation from the radar to the camera coordinate frame using the equation given below.

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = R \times \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} + t$$

Together, R and t form 12 parameters, expressed as given below.

$$[R|t] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To estimate these parameters which model the camera pose with respect to the radar, we use the Efficient Perspective-n-Point (EPnP) algorithm. The algorithm requires at least 6 known correspondences between 3D radar data and 2D camera pixel coordinates. We obtain these correspondences by recording the image pixel points corresponding to a known target detected by the radar and repeat the process by changing the target's position in each iteration. The EPnP estimation results are refined using a non-linear estimation method, one of which is the Gauss–Newton optimization algorithm.

The Sensor Fusion Algorithm

In this section, we lay the foundations of the core component of the fusion algorithm: the Kalman filter. The radar and camera each provide complementary information about the scene, but both are affected by measurement noise and hardware limitations. To generate reliable and continuously updated target tracks, with the target's position and velocity, we must consider two elements: the noise in the sensor data and the dynamics of the moving targets.

Estimation methods that rely solely on measurements, such as RLS, are ineffective here because they assume static or slowly varying parameters. In contrast, targets motions are dynamic with time and must be modelled explicitly through a motion profile. However, the motion model alone cannot fully capture real-world target behavior due to the uncertainty in their motion.

The Kalman filter addresses both issues, by:

(1) predicting the state using the motion model, and
(2) correcting that prediction using sensor measurements.

This prediction–update cycle runs at every time step, gradually stabilizing the estimates and producing smooth, reliable tracks. Under the assumption of white Gaussian noise, the Kalman filter is the optimal linear estimator. Since in our system, the radar provides a nonlinear measurement of the target's radial velocity, the Extended Kalman Filter (EKF) is used, which linearizes the measurement equation around the nominal point, prior state estimate (to be defined in the upcoming subsections).

The remainder of this chapter presents the essential theoretical background: the state space model, the discrete-time state-space model, observability considerations, the EKF prediction and update equations, the selected constant-acceleration motion model, and the estimation of the sensor noise covariance matrices.

State Space Model

A state space model is used to represent how the states and output of a system evolve with time, which is dependent on the system's internal modes or dynamics. It is also dependent on initial states and / or a control input. The system state equation and the output equation, as well as the dimensions of each vector and matrix are given below.

$$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u} \qquad \dot{\boldsymbol{x}}, \boldsymbol{x} \ni \mathbb{R}^{n\times 1}, \boldsymbol{A} \ni \mathbb{R}^{n\times n}, \boldsymbol{B} \ni \mathbb{R}^{n\times r}, \boldsymbol{u} \ni \mathbb{R}^{r\times 1}$$

$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} + \boldsymbol{D}\boldsymbol{u} \qquad \boldsymbol{y} \ni \mathbb{R}^{q\times 1}, \boldsymbol{C} \ni \mathbb{R}^{r\times n}, \boldsymbol{D} \ni \mathbb{R}^{q\times r}$$

What are the system states? The system states are the minimum number of variables that if known at t = t_0 together with the input for t > t_0 allows you to determine the system output for all future time t>t_0 [16]; and the state space is an n – dimensional space that represents all possible states the system can only potentially reach.

The solution to the system model, $\boldsymbol{y(t)}$ $\boldsymbol{and}$ $\boldsymbol{x(t)}$, are given below (assuming the feedforward matrix $\boldsymbol{D}$ is zero). These can be obtained by either solving the differential equation or taking Laplace or Fourier transform on both sides and their inverses to return back to the solution's time domain expression.

$$\boldsymbol{x}(t) = \boldsymbol{x}(t_0) \cdot e^{\boldsymbol{A}(t-t_0)} + \int_{t_0}^{t} e^{\boldsymbol{A}(t-\tau)} \cdot \boldsymbol{B} \cdot \boldsymbol{u}(t-\tau)d\tau$$

$$\boldsymbol{y}(t) = \boldsymbol{C} \cdot \boldsymbol{x}(t_0) \cdot e^{\boldsymbol{A}(t-t_0)} + \boldsymbol{C} \cdot \int_{t_0}^{t} e^{\boldsymbol{A}(t-\tau)} \cdot \boldsymbol{B} \cdot \boldsymbol{u}(t-\tau)d\tau$$

The $\boldsymbol{A}$ matrix is the State Transition Matrix. Since we observe the matrix $\boldsymbol{A}$ to be (1) both in the zero – input and the zero – state parts of the system solution, and (2) it is taken as an exponent, the properties of the matrix directly reflect the system's structure and determines its stability, generally describing whether the output of the system converges for bounded inputs.

While most system's dynamic is described by continuous equations, however, they are processed by microprocessor which can operate on only a discrete set of values [16], for that, continuous system state equations are usually quantized, as well as the system measurement equation. In the next section, we show how to derive the state equation model of a discrete time system.

State Space Model of Discrete Time Systems

Assume (1) the sampling period is $\Delta t = t_k - t_{k-1}$ [16], (2) the system is piece -wise constant, meaning the control input, $\boldsymbol{u}(t)$, and the state transition matrix, $\boldsymbol{A}$, and the input matrix, $\boldsymbol{B}$, are constant over the sampling period from t_{k-1} to t_k [16], then, the state at the future time (t_k), can be expressed with respect to its current value, $\boldsymbol{x}(t_{k-1})$, as given below [16].

$$\boldsymbol{x}(t_k) = \boldsymbol{x}(t_{k-1}) \cdot e^{\boldsymbol{A}(t_k - t_{k-1})} + \int_{t_k-1}^{t_k} e^{\boldsymbol{A}(t_k-\tau)} \cdot \boldsymbol{B} \cdot u(t_k - \tau) d\tau$$

To expresses the integral as a function of Δt, we use the substitution variable (α) in place of $t_k - \tau$. The resulting expression is shown below [16].

$$\boldsymbol{x}(t_k) = \boldsymbol{x}(t_{k-1}) \cdot e^{\boldsymbol{A}(\Delta t)} + u(\alpha) \int_0^{\Delta t} e^{\boldsymbol{A}(\alpha)} \cdot \boldsymbol{B}\, d\alpha$$

Comparing this equation to the continuous time state space model equation, we can define the discrete time system state equation as follows:

$$\boldsymbol{x_k} = \boldsymbol{F} \cdot \boldsymbol{x_{k-1}} + \boldsymbol{G} \cdot \boldsymbol{u_{k-1}}$$

Were:

$$\boldsymbol{F} = e^{\boldsymbol{A}(\Delta t)}, \text{ and } \boldsymbol{G} = \int_0^{\Delta t} e^{\boldsymbol{A}(\alpha)} \cdot \boldsymbol{B}\, d\alpha$$

Depending on the system, samples maybe are taken at uniform periods or non – uniform sampling maybe performed, in which case Δt is not constant. $\boldsymbol{F}$ and $\boldsymbol{G}$ are then a function of time and written as $\boldsymbol{F_{k-1}}$ and $\boldsymbol{G_{k-1}}$. $\boldsymbol{x_{k-1}}$ represents the current state value, and $\boldsymbol{u_{k-1}}$ represents the current input applied. The output equation is given below, the discrete time matrix $\boldsymbol{H}$ matrix maps the states to their respective output measurements [17].

$$\boldsymbol{y_k} = \boldsymbol{H} \cdot \boldsymbol{x_k}$$

In the next section, we discuss observability, which discusses the conditions under which we can re-construct our states given only the output measurements.

Observability, Constructability, and the Kalman Filter

The observability of a linear system is a critical concept in a Kalman filter, since the Kalman filter is fundamentally an observer [18]. Hence for a given system, it must be verified that it is observable, to guarantee the constructability of its states from the output measurement captured [19]. For a discrete time system, the observability definition is given as follows:

"*A discrete-time system is observable if for any initial state* $\boldsymbol{x_0}$ *and some final time k the initial state* $\boldsymbol{x_0}$ *can be uniquely determined by knowledge of the input* $\boldsymbol{u_i}$ *and output* $\boldsymbol{x_i}$ *for all* $i \in [0, k]$." [16]

While there are multiple definitions for what makes a system observable [16]. The selection of the method used depends on its computational ease in a given scenario [16]. We present here the widely known definition:

"*The n-state discrete linear time-invariant system*

$$\boldsymbol{x_k} = \boldsymbol{F} \cdot \boldsymbol{x_{k-1}} + \boldsymbol{G} \cdot \boldsymbol{u_{k-1}}$$

$$\boldsymbol{y_k} = \boldsymbol{H} \cdot \boldsymbol{x_k}$$

has the observability matrix Q defined by

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{H} \\ \boldsymbol{HF} \\ \vdots \\ \boldsymbol{HF^{n-1}} \end{bmatrix}$$

The system is observable if and only if p(Q) = n." [16]

The matrix Q has full rank if *either* **(1)** the H matrix has full rank [16, 20], meaning in the Jordon form of matrix $\boldsymbol{H}$ ($\bar{\boldsymbol{H}} = \mathbf{HM}, \bar{\mathbf{F}} = \mathbf{M^{-1}FM}, \bar{\mathbf{G}} = \mathbf{M^{-1}G}$, where the $\mathbf{M}$ matrix is the matrix of linearly independent eigenvectors of the matrix F) its states are coupled to the output through a non – zero row in $\bar{\boldsymbol{H}}$ [16, 20]. The rank of the H matrix (linearly independent rows or columns) determines the number of directions in which we can be observe our states in $\mathbb{R}^{n\times 1}$ [20].

(2) When H is not full rank, the system dynamics defined by F can help allow the states of the system become observable after some n dynamic steps in time [20]. The Linearly independent columns or rows of F help complement those of H such that all states in (n) possible different directions can be observed over time [20].

The estimation algorithms such as LS, weighted LS, or RLS are generalizations of the Kalman filter which assumes dynamic systems. To gain better understanding of the Kalman filter presented in the next section, check Appendix A for additional material discussing the mathematics of these estimation methods and their connections with the KF. In the next section we discuss the Kalman filter equations.

Kalman Filter Equations

The Kalman filter predicts future states by modelling how the states' mean evolves over time [16]. Using the system model, we express future states as a function of the state transition matrix, the mean of the previous state, the input signal (if any), and process noise [16]. For zero-mean noise, which is a requirement for an unbiased KF, the last term disappears, as shown below.

$$E(\boldsymbol{x_k}) = \boldsymbol{F_{k-1}} E(\boldsymbol{x_{k-1}}) + \boldsymbol{G_{k-1}} \boldsymbol{u_{k-1}} + E(\boldsymbol{w_{k-1}})$$

$$\overline{x_k} = \boldsymbol{F_{k-1}}\overline{x_{k-1}} + \boldsymbol{G_{k-1}u_{k-1}}$$

The next property of interest is the covariance of the state estimation error. Substituting the above equation into the covariance expression gives the discrete-time Lyapunov (or Stein) equation [16]:

$$E[(\boldsymbol{x} - \overline{x_k})(\boldsymbol{x} - \overline{x_k})^T] = \boldsymbol{F_{k-1}P_{k-1}F_{k-1}}^{\boldsymbol{T}} + \boldsymbol{Q_{k-1}}$$

We cannot rely solely on measurements, as in RLS, as the tracked states are not constants. Instead, we use the model to predict the expected future state and correct this estimate using the measurements. The Kalman filter defines two estimates for x, the priori and posteriori estimates, each of whose equation are given below [16].

$$\hat{\boldsymbol{x}}_{\boldsymbol{k}}^{-} = E[\boldsymbol{x_k}|\boldsymbol{y_1}, \boldsymbol{y_2}, \cdots, \boldsymbol{y_{k-1}}] \textbf{ (priori)}$$

$$\hat{\boldsymbol{x}}_{\boldsymbol{k}}^{+} = E[\boldsymbol{x_k}|\boldsymbol{y_1}, \boldsymbol{y_2}, \cdots, \boldsymbol{y_k}] \textbf{ (posteriori)}$$

The posteriori estimates at $t = k$ refers to the mean of x after including the measurement at $t = k$. The more measurements available, the better the estimate, as noise averages out. The covariance of the estimation error, P_k^+, always decreases after a measurement, with the rate of decrease depending on measurement noise.

We use the same RLS equations, incorporating priori and posteriori estimates:

Prediction Step*:* The prediction is based on the model, providing the estimate of x at $t = k$ before the measurement y_k. We substitute the posteriori estimate from $k - 1$ and use the Stein equation to predict how the posteriori covariance propagates. The two prediction equations are given below.

$$\boldsymbol{x_k^-} = \boldsymbol{F_{k-1}x_{k-1}^+} + \boldsymbol{G_{k-1}u_{k-1}}$$

$$\boldsymbol{P_k^-} = \boldsymbol{F_{k-1}P_{k-1}^+F_{k-1}}^{\boldsymbol{T}} + \boldsymbol{Q_{k-1}}$$

Time – Update Step*:* The Kalman gain K_k balances the contribution of the measurement and the model prediction. Using the priori estimate, the posteriori estimate is updated, and the covariance is recalculated for the next prediction step:

$$\boldsymbol{K_k} = \boldsymbol{P_k^-H_k^T} \cdot (\boldsymbol{H_kP_k^-H_k^T} + \boldsymbol{R_k})^{-1}$$

$$\boldsymbol{x_k^+} = \boldsymbol{x_k^-} + K_k(\boldsymbol{y_k} - \boldsymbol{H_kx_k^-})$$

$$\boldsymbol{P_k^+} = (\boldsymbol{I} - \boldsymbol{K_kH_k})\boldsymbol{P_k^-}(\boldsymbol{I} - \boldsymbol{K_kH_k})^{\boldsymbol{T}} + \boldsymbol{K_kR_kK_k^T}$$

Here we provide the summary of the Kalman filter taken from source [16], (pg. 128 – 129).

“1. The dynamic system is given by the following equations:

$$\boldsymbol{x_k} = \boldsymbol{F_{k-1}x_{k-1}} + \boldsymbol{G_{k-1}u_{k-1}} + \boldsymbol{w_{k-1}}$$

$$y_k = \boldsymbol{H_k x_k} + v_k$$

$$E\,(\boldsymbol{w_k w_i^T}) = Q_k \delta_{k-i}$$

$$E\,(\boldsymbol{v_k v_i^T}) = R_k \delta_{k-i}$$

$$E\,(\boldsymbol{v_k w_i^T}) = 0$$

2. The Kalman filter is initialized as follows:

$$\hat{\boldsymbol{x}}_{\boldsymbol{0}}^{+} = E(\boldsymbol{x_0})$$

$$\boldsymbol{P_0^+} = E[(\boldsymbol{x_0} - \hat{\boldsymbol{x}}_{\boldsymbol{0}}^{+})(\boldsymbol{x_0} - \hat{\boldsymbol{x}}_{\boldsymbol{0}}^{+})^{\boldsymbol{T}}]$$

1. The Kalman filter is given by the following equations, which are computed for each time step k = 1, 2,..:

$$\boldsymbol{P_k^-} = \boldsymbol{F_{k-1} P_{k-1}^+ F_{k-1}}^{\boldsymbol{T}} + \boldsymbol{Q_{k-1}}$$

$$\boldsymbol{x_k^-} = \boldsymbol{F_{k-1} x_{k-1}^+} + \boldsymbol{G_{k-1} u_{k-1}} = \textit{a priori state estimate}$$

$$\boldsymbol{K_k} = \boldsymbol{P_k^- H_k^T} \cdot (\boldsymbol{H_k P_k^- H_k^T} + \boldsymbol{R_k})^{-1}$$

$$\boldsymbol{x_k^+} = \boldsymbol{x_k^-} + K_k(\boldsymbol{y_k} - \boldsymbol{H_k x_k^-}) = \textit{a posteriori state estimate}$$

$$\boldsymbol{P_k^+} = (\boldsymbol{I} - \boldsymbol{K_k H_k}) \boldsymbol{P_k^-} (\boldsymbol{I} - \boldsymbol{K_k H_k})^{\boldsymbol{T}} + \boldsymbol{K_k R_k K_k^T}$$ ”

Insights on the Kalman Filter Equations

If we examine the priori covariance equation closely, we see that it is simply the result of a linear transformation applied to the posteriori estimate of x from step $k-1$. Consider a random variable x with a Gaussian distribution and parameters (μ, C_X), where C_X is the autocovariance matrix. If y is a linear transformation of x, then y also follows a Gaussian distribution:

$$\boldsymbol{y} = \boldsymbol{Ax} + \boldsymbol{b}$$

$$\boldsymbol{x} \sim N(\boldsymbol{\mu}, \boldsymbol{C_X})$$

$$\boldsymbol{y} \sim \boldsymbol{N}(\boldsymbol{A\mu} + \boldsymbol{b}, \boldsymbol{AC_X A^T})$$

The Q matrix represents the covariance of the added processing noise. Recall that the state is a multivariate stochastic sequence with a Gaussian distribution. Its contours form ellipsoids, representing state values with equal probability [21]. In the prediction step, the ellipsoid either contracts or expands with each iteration, depending on the system’s stability and the amount of processing noise present (the role of F in stability will be discussed in the next section) [17, 21].

Another important aspect is the inverse term in the gain matrix, known as the innovation covariance. When the distribution of y is noisier or more spread out, the gain matrix is smaller. Consequently, the innovation term contributes less to the state estimation [16, 21].

For the posteriori covariance, recall that the gain matrix was selected to minimize the estimation error covariance in RLS. In P_k^+, the term $(I - K_k H_k)$ multiplies P_k^-, reducing the covariance. How much the covariance ellipsoid shrinks depends on K_k [16]. The lower the measurement noise covariance, the smaller the gain matrix, and the closer $(I - K_k H_k)$ is to the identity. This is why the convergence rate depends on measurement noise: it determines how much P_k^- is reduced, while P_k^+ is always less than P_k^- [16].

For the KF, the relative norm of R versus Q ultimately decide how the estimator weights the model versus the measurements. If Q>> R, then the P_k^- increases, K_k increases, and the KF relies more on the measurement, reflecting modelling error [16]. Conversely, if Q << R, $K_k$decreases, and the estimator relies more on the model [16].

Finally, consider the initial estimate. If it is close to the true state, $P_k^-$is small for $k = 1$, resulting in a smaller Kalman gain K_k. The filter may initially give less weight to measurements, which could mathematically slow convergence. In practice, however, a good initial estimate may allow the filter to stabilize more readily and, conversely, speed up convergence. On the other hand, if the initial estimate is far from the true state, $P_k^-$is large for $k = 1$, and the filter relies more heavily on measurements. Over time, the initial estimate has minimal effect on performance [16].

Summary of assumption and conditions on the optimality of the Kalman Filter

We summarize below the assumptions that make the Kalman filter an optimal linear estimator, based on [16]. These could also be considered limitations, since these conditions may not be met perfectly in practical applications. In the next section, we show how the Jacobian is used to linearize the state and measurement models around a nominal point, which enables the Extended Kalman Filter to handle nonlinear systems.

1. The system state or dynamics equation is linear,
2. The system measurement equation is linear.
3. The measurement noise and the processing noise vectors follow are RV and follow a Gaussian distribution,
4. The noise and processing noise have zero-mean,
5. The measurement noise at time k, and the estimation error at time $(k - 1)$ is independent,
6. The measurement noise is white, meaning $E\,(\boldsymbol{v_k v_i}^T) = \boldsymbol{R_k}\delta_{k-i}$, where $\boldsymbol{R_k}$ measurement noise covariance matrix,

7. The processing noise is white, meaning $E(\boldsymbol{w_k w_i}^T) = \boldsymbol{Q_k}\delta_{k-i}$, where $\boldsymbol{Q_k}$ measurement noise covariance matrix, and
8. The measurement and processing noise are independent.

Non - Linear Models and the Discrete-time Extended Kalman Filter

Assuming discretized dynamics of continuous-time systems, the Extended Kalman filter is derived as follows. For a system modelled as shown below [2].

$$\boldsymbol{x_k} = f_{k-1}(\boldsymbol{x_{k-1}}, \boldsymbol{u_{k-1}}, \boldsymbol{w}_{k-1})$$

$$\boldsymbol{y_k} = h_{\boldsymbol{k}}(\boldsymbol{x_k}, \boldsymbol{v_k})$$

$$\boldsymbol{v_k} \sim N(0, \boldsymbol{R_k})$$

$$\boldsymbol{w_k} \sim N(0, \boldsymbol{Q_k})$$

The state function f(.) can be linearized by expanding the function using the Taylor series around the nominal point $\boldsymbol{x_{k-1}} = \boldsymbol{x_{k-1}^+}$ and with $\boldsymbol{v_k} = 0$. For the measurement function, h(.), we linearize the function around the nominal point $\boldsymbol{x_{k-1}} = \boldsymbol{x_k^-}$ and with $\boldsymbol{w_k} = 0$, our best estimate of x following the prediction step. Assuming the system is slowly varying, and the state estimate is close to the nominal point, higher order derivatives of the Taylor series may be reasonably considered to be negligible.

The resulting system state equation, system measurement equation, as well as the resulting Jacobians are given below.

$$y_k = f_{k-1}(\hat{x}_{k-1}^+, 0) + \left.\frac{\partial f_{k-1}}{\partial x}\right|_{\hat{x}_{k-1}^+} (x_{k-1} - \hat{x}_{k-1}^+)$$

$$y_k = h_k(\hat{x}_k^-, 0) + \left.\frac{\partial h_k}{\partial x}\right|_{\hat{x}_k^-} (x_k - \hat{x}_k^-)$$

$$F_{k-1} = \left.\frac{\partial f_{k-1}}{\partial x}\right| \hat{x}_{k-1}^+$$

$$H_k = \left.\frac{\partial h_k}{\partial x}\right| \hat{x}_k^-$$

The discrete- time Extended Kalman filter summary from source [16] is give below: **“**

1) Initialize the filter as follows:

$$\hat{x}_0^+ = E(x_0)$$

$$P_0^+ = E[(x_0 - \hat{x}_0^+)(x_0 - \hat{x}_0^+)^T]$$

2) For k=1,2, …, perform the following:

(a) compute the following partial derivative matrix:

$$F_{k-1} = \frac{\partial f_{k-1}}{\partial x}\bigg|_{\hat{x}_{k-1}^{+}}$$

(b) Perform the time update of the state estimate and estimation-error-covariance as follows:

$$P_k^- = F_{k-1}P_{k-1}^+F_{k-1}^T + Q_{k-1}$$

$$\hat{x}_k^- = f_{k-1}(\hat{x}_{k-1}^+, u_{k-1}, 0)$$

c) Compute the following partial derivative matrix:

$$H_k = \frac{\partial h_k}{\partial x}\bigg|_{\hat{x}_k^-}$$

d) Perform the measurement update of the estate estimate and estimation-error covariance as follows:

$$K_k = P_k^- H_k^T\left(H_k P_k^- H_k^T + R_k\right)^{-1}$$

$$\hat{x}_k^+ = \hat{x}_k^- + K_k[y_k - h_k(\hat{x}_k^-, 0)]$$

$$P_k^+ = (I - K_k H_k)P_k^- \text{ ”.}$$

Data Association and Target Tracking

Data association and target tracking are used to make sense of measurements received from sensors by associating new detections with the history of past observations of a given target or targets. In radar target detection, it is quite common, more so than with cameras, for radar data to produce false alarms or clutter. This motivates the use of data association methods, target tracking, and track management to accurately differentiate true target measurements from false alarms and clutter. In the following sections, we delve deeper into the theory of data association and show how the Kalman filter forms a key component of track formation. In the final section of the theoretical foundation chapter, we present the standards and regulations considered for the data fusion system.

Standards and Regulations

This section outlines the IEEE standards relevant to the design and implementation of the radar–camera fusion system. These standards address software safety, verification and validation, sensor performance, ethical engineering practice, and environmental responsibility. Their inclusion ensures that the system is designed according to recognized international guidelines that promote reliability, safety, and sustainability.

Design and Ethical Standards: IEEE Standards

P1228 - Standard for Software Safety

This standard guides the identification and mitigation of software-related hazards throughout the system life cycle. Its relevance appears in how the project structures the radar parsing, YOLO detection, and Extended Kalman Filter (EKF) modules. Each component must operate safely under real-time constraints, avoid propagating erroneous outputs, and handle edge cases such as missing frames or noisy radar points without compromising system behavior.

IEEE 1012 - Standard for System and Software Verification and Validation

This standard defines the processes required to verify and validate system performance. In this project, it is reflected through staged validation: radar-only testing, camera-only testing, YOLO-on-video verification, calibration accuracy checks, and EKF testing using pre-recorded logs before attempting real-time execution. This structured verification approach ensures that each module satisfies its intended functionality before integration with the full fusion system.

IEEE 2700 - Standard for Sensor Performance Parameter Definitions

IEEE 2700 provides standardized terminology and definitions for sensor attributes such as accuracy, resolution, noise, sensitivity, and uncertainty. These concepts influence radar point-cloud interpretation, EKF measurement tuning, clustering thresholds, and calibration evaluation. Using standardized sensor definitions improves the consistency of performance assessment and documentation.

IEEE7000 - Standard Model Process for Addressing Ethical Concerns During System Design

This standard ensures that ethical considerations are integrated into engineering design processes. It applies to this project through responsible handling of recorded scenes, transparency in documenting system limitations, and prioritizing user safety by validating outputs before real-time use. Ethical decision-making influences choices about data recording, algorithm testing, and reporting performance.

IEEE Code of Ethics (Section 7.8)

The code emphasizes safety, integrity, competence, and responsible data practices. The project follows these principles by maintaining accurate logging, validating detections and tracking results before real-time execution, and clearly reporting system behavior, errors, and limitations.

IEEE 1680 Series — Environmental Assessment Standards for Electronic Products

While the project does not produce a commercial device, the environmental principles of this standard influence hardware selection and system design. Low-power components (Raspberry Pi, TI mmWave radar), reusable mechanical mounts, and lightweight algorithms contribute to reduced energy consumption and minimized environmental footprint.

Applicability in Current Design

The above standards directly shape the development and performance of the fusion system:

Software reliability (P1228):

The radar parsing, YOLO detections, timestamp synchronization, and EKF tracking are implemented with safety considerations such as error-handling, organized modularity, and protection against invalid or missing sensor data.

Verification and validation (IEEE 1012):

Each subsystem is verified independently: radar plots, intrinsic calibration accuracy, YOLO detection correctness, ArUco-based ground truth, and offline EKF tracking, before being combined into the unified system.

Sensor performance definitions (IEEE 2700):

Noise characteristics, resolution, and accuracy metrics guide radar clustering, camera calibration interpretation, and EKF noise tuning.

Ethical engineering practice (IEEE 7000 & IEEE Code of Ethics):

The system development prioritizes transparency, accuracy, and responsible data usage. Recorded scenes are handled securely, and all results are validated before being applied in real-time tests.

Environmental responsibility (IEEE 1680 Series):

Design choices emphasize low-power hardware, efficient algorithms, reduced heat generation, and reusable physical components. These environmentally conscious decisions are consistent with sustainable embedded-system development.

Together, these standards ensure that the radar–camera fusion system is reliable, safe, ethically developed, and environmentally responsible.

Environmental Impacts and Sustainability

Environmental considerations were integrated into both the hardware and software aspects of the radar–camera fusion system. The goal was to design a perception framework that is energy-efficient, computationally lightweight, and aligned with sustainable engineering principles.

On the hardware side, the system relies on low-power embedded components. The Raspberry Pi was selected because it consumes significantly less energy than full GPU-based processing platforms while still providing the required computational capability for radar parsing, calibration procedures, and EKF tracking. Likewise, the TI mmWave radar sensor operates with modest power requirements, enabling continuous operation without excessive heat generation or energy overhead. The mechanical mount

designed for the sensors was reused across experiments, reducing material waste and minimizing additional fabrication.

From a software perspective, sustainability was addressed through algorithmic simplicity and efficiency. The choice to use classical techniques (such as background subtraction for initial experimentation, lightweight YOLO variants for detection, and the Extended Kalman Filter for tracking) kept computational load low. These methods require far fewer operations than heavier deep-learning-based fusion models, lowering power consumption during execution and enabling real-time performance on embedded hardware. Multithreading and offline calibration also reduce redundant computation, further improving the system's energy efficiency.

By combining low-power hardware with computationally efficient algorithms, the project minimizes energy use, reduces heat generation, and supports long-term sustainable operation. These design decisions reflect an environmentally responsible approach appropriate for embedded perception systems.

Concept Generation and Design Evaluation

This section outlines the design alternatives considered for camera-based object detection, data fusion algorithms, the pedestrian motion model, and the estimation of radar and camera noise covariance matrices. We also discuss the choice between synchronous and asynchronous fusion architectures, the machine learning model and feature selection used to improve performance under foggy conditions, the method used to obtain ground-truth states, and finally, a summary of the system's limitations.

Different Design Options and Assessment

In this section, we examine several major project milestones that required engineering reasoning. We present the study of the available options and summarize the rationale behind the final design choices.

Camera Object Detection Methods

Three object detection algorithms were evaluated: Background Subtraction (BSA), FOMO, and YOLO. The criteria considered included latency, accuracy, load on the Raspberry Pi, support for multiple object detections, and object classification. The detailed implementation results are presented in Chapter 5; here we summarize the evaluation used for design selection.

Background Subtraction Algorithm (BSA)

BSA ran efficiently on the Raspberry Pi and reliably detected moving objects in simple scenes. Compared to FOMO and YOLO, BSA lacks both object classification and reliable performance on non-static platforms. The first limitation is crucial in applications such as ADAS, where a moving platform (car) causes static objects to appear as moving, leading to false detections. The second limitation affects the ability to switch EKF motion profiles based on object classification, which would enable increasing the range of reliably trackable target types.

Fast Objects Moving Objects (FOMO)

After initial training, FOMO achieved an accuracy of 0% on a mildly complex dataset. Accuracy increased only slightly when the dataset was expanded with simpler scenes. While FOMO imposes significantly less load on the Raspberry Pi, the training results showed its architectural limitations in handling even moderately complex environments. We concluded that continuing with FOMO would not yield acceptable accuracy, its limitations stem from its architecture, not just dataset size.

You Only Look Once (YOLO)

YOLO provided strong advantages: object classification and high accuracy. The main drawback was latency. However, with the use of an accelerator, this latency can be significantly reduced. Therefore, we proceeded with YOLO despite the additional complexity of integrating the Edge TPU optimizer. This approach reduces the load and memory usage on the Raspberry Pi, enabling high accuracy with

low latency. The final results and implementation challenges of all three object detection algorithms are presented in the next chapter, Implementation and Fabrication. In the next subsection, we discuss

Data Fusion Algorithms

In this section, we outline and evaluates three data fusion methods, (1) Extended Kalman Filter (EKF), (2) the Hungarian algorithm, and (3) the Bayesian-based two-level fusion scheme. To determine the most suitable approach, a weighted matrix was developed to compare the methods across several criteria. Below, we discuss each method briefly before introducing the matrix and the evaluations.

Extended Kalman Filter

As shown in [2], before applying the Extended Kalman Filter (EKF), we first use a Moving Target Indicator (MTI) to filter out static objects from the radar detections. After that, clustering algorithms group radar points that likely belong to the same object, and bounding boxes are drawn around these detections on the image.

The EKF then fuses information from both the radar and the camera to track the object's position over time. It starts with the prediction stage, where a state-space model is used to predict where the object is likely to be based on its past motion. However, because most real-world systems are nonlinear, we calculate the Jacobian, a matrix of partial derivatives, to linearize the system around the current estimate.

Finally, the filter updates its prediction by incorporating new measurements from the sensors. This update step is controlled by the Kalman Gain, which decides how much trust to place in the new sensor readings compared to the predicted state. In simple terms, it balances the uncertainties between what we predict and what the sensors actually measures.

Bayesian-based two-layer fusion scheme

As seen in [6], to use the Bayesian-based two-level fusion method, we first need to match the data from the radar and camera using the Kuhn–Munkres algorithm. We fuse the data provided by both sensors (such as the position). There are three important pieces of data to be used: the previously fused information, the data provided by the camera, and the data provided by the radar.

For the example of position, in the first fusion, the previously fused position is used as the prior, and the current camera position is used as the posterior, which gives you Ptemporary. This is then used in the second fusion as the prior, while the radar position is the posterior, to get the current fused position, thus reducing the uncertainty. In the case of detecting an object for the first time, the camera's current position is used as the prior, and the radar's current position is used as the posterior, to give you the current fused position.

Hungarian Algorithm

When using the Hungarian algorithm, as seen in [5], we first use a radar-to-camera transformation matrix to transform radar data into the same coordinate space as the camera, enabling the projection of radar detections onto the camera image. Since the radar returns are usually very noisy, a density-based clustering scheme is used to cluster the centroids that are very close together, which helps identify groups of radar points that correspond to individual objects.

A centroid for each cluster is computed and projected onto the camera image using the radar-to-camera transformation matrix (which helps translate radar's 3D world coordinates into 2D coordinates on the camera's image). The radar centroids then have a position on the camera image. The bounding boxes produced by the image detection process we selected also have a centroid. The Hungarian algorithm is then used to associate the radar cluster centroids with the bounding box centroids, ensuring that the correct radar detection matches the correct object in the image. In the next section, we introduce the weighted matrix comparison utilized to evaluate each method's effectiveness for the project.

Comparison Matrix

A decision or weighted comparison matrix was utilized to evaluate each method's effectiveness and determine the most suitable approach for our application. The matrix is shown in Table 1. The metrics used are listed below in order of importance.

1- Accuracy: To compare how well each method is at detecting objects. This is the most important factor, so it was given the highest weight
2- Computational capacity: Since the Raspberry Pi has a limited computational capacity, we must make sure that we are using a lightweight method
3- Noise robustness: Radar data is very noisy, so we must make sure our method can detect false positives
4- Ego motion handling: Since our device will be moving it will be important to be able to account for the movement of our sensor
5- Availability of resources: This is important to be able to access similar optimized or simplified versions of the different steps of our project.

The Hungarian algorithm is the least accurate method since it doesn't have a statistical foundation, while the other two methods iteratively reduce uncertainties. Hence it was assigned a score of (6/10 vs. 8/10).

Since the Extended Kalman Filter (EKF) is mathematically complex, it requires significantly more computational capacity than the other two methods. The Bayesian-based fusion technique also demands considerable computational resources due to its two-stage probabilistic updates, whereas the

Table 1. Data Fusion Algorithm Weighted Comparison Matrix

Metric	Weight	Extended Kalman Filter		Bayesian - Based Fusion		Hungarian Algorithm	
		Raw Score	Weighted Score	Raw Score	Weighted Score	Raw Score	Weighted Score
Accuracy	35	8/10	28	8/10	28	6/10	21
Computational capacity	25	4/10	10	5/10	12.5	8/10	20
Noise robustness	15	8/10	12	7/10	10.5	4/10	6
Ego motion handling	15	9/10	13.5	6.5/10	9.75	4/10	6
Availability of resources	10	9/10	9	3/10	3	7/10	7
Total	-	-	72	-	67	-	60

Hungarian algorithm is relatively simple and more suited for devices such as the Raspberry Pi. Hence the following scores were assigned (4/10 vs. 5/10 vs. 8/10), respectively.

When it comes to noise sensitivity, the Hungarian algorithm is the most vulnerable since it does not model prior values, making it more prone to false positives, and hence it was assigned a score of (4/10). In contrast, the other two methods are based on Bayesian theory, which accounts for previous measurements and helps improve robustness against noise. The EKF algorithm was assigned a score of (8/10) while the Bayesian - Based Fusion was assigned a score of (7/10).

Moreover, EKF is well suited for handling ego-motion, as it tracks motion over time (9/10), while the Hungarian algorithm only performs static matching (4/10). The Bayesian-based fusion technique could handle ego-motion but would require additional layers and complexity (6.5/10).

EKF is widely recognized and commonly used in various sensor fusion projects (9/10). On the other hand, the Bayesian-based fusion technique would require more custom adaptations, as there are very few research papers implementing it (3/10). The Hungarian algorithm has been utilized in research but not as frequently as EKF (7/10).

Since EKF scores the highest overall based on accuracy, noise handling, and practicality, we conclude that it will be the method we use in the future to fuse the data from our camera and radar. In the next subsection, we discuss the advantages and disadvantages of the asynchronous and synchronous implementation of the fusion algorithm.

Synchronous and Asynchronous Data Fusion

The selection between whether to apply synchronous or asynchronous fusion depends mainly on the specific application and relates to multiple performance factors. For instance, it is common to use asynchronous fusion in applications involving sensor data transmitted wirelessly, due to the system involving multi-rate sensors and usually an inconsistent update rate caused by competitive channel access and packet collisions introducing uncertainties. Applying asynchronous fusion means updating the Kalman filter upon the receipt of a new measurement from any of the sensors.

Moreover, in systems involving sensors with more reliable update rates, a simpler method is to apply synchronous fusion, which involves updating the Kalman filter using the measurements from all sensors simultaneously. While synchronous fusion is appreciably simpler to implement, this comes at a cost, as source [22] suggests that synchronous fusion tends to be less accurate and undoubtedly introduces considerably greater latency compared to asynchronous fusion. This is because synchronous fusion methods involve waiting for all sensor measurements before updating the Kalman filter. In interpolation, for the slower sensor (with fewer measurements per frame), empty slots may be filled by estimations from previous measurements. The R and H matrices and the y vector of each sensor are merged as shown below. Note the following dimensions: $H_k \in \mathbb{R}^{q \times n}$, $y_k \in \mathbb{R}^{q \times 1}$, and $K_k \in \mathbb{R}^{n \times q}$.

$$\boldsymbol{R}_{combined} = \begin{bmatrix} \boldsymbol{R}_{radar} & 0 \\ 0 & \boldsymbol{R}_{cam} \end{bmatrix}$$

$$\boldsymbol{y}_{combined} = \begin{bmatrix} \boldsymbol{y}_{radar} \\ \boldsymbol{y}_{cam} \end{bmatrix}$$

$$\boldsymbol{H}_{combined} = \begin{bmatrix} \boldsymbol{H}_{radar} \\ \boldsymbol{H}_{cam} \end{bmatrix}$$

In asynchronous fusion, the F and Q matrices are time-variant, since they depend on Δt (calculated as |t_current − t_measurement_frame| [23]), which is non-constant for every iteration of the Kalman filter. Therefore, asynchronous fusion is more computationally intensive. Furthermore, it involves careful debugging to keep track of the received measurements in order to avoid logical errors. Specifically, it requires handling out-of-sequence measurements, which occur when the microprocessor receives a measurement from a sensor processing a frame at t_l while the last Kalman filter update corresponded to a frame taken at t_k, with t_l > t_k, meaning that the received measurement is from the past. Handling out-of-sequence measurements is explained in detail in a later section. Moreover, in an upcoming section, we discuss the rationale behind the selected fusion architecture, taking into account the YOLO implementation results. In the next section, we discuss the background and selection of the system motion model.

Motion Model Selection

The Kalman filter uses the state transition matrix F to relate the system state at time $k-1$ to the state at time k. In our project, F models the evolution of the target's states. Based on kinematics, the discrete-time equations for one-dimensional motion are:

$$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \dot{\boldsymbol{x}}_{k-1}\Delta t + \frac{1}{2}\ddot{\boldsymbol{x}}_{k-1}\Delta t^2$$

$$\dot{\boldsymbol{x}}_k = \dot{\boldsymbol{x}}_{k-1} + \ddot{\boldsymbol{x}}_{k-1}\Delta t$$

$$\ddot{\boldsymbol{x}}_k = \ddot{\boldsymbol{x}}_{k-1}$$

One of the simplest target manoeuvre models based on these kinematic equations is the constant acceleration (CA) model, also referred to as the Wiener-process acceleration (WPA) or nearly constant acceleration (NCA) model [24]. A specific variant, the Wiener-sequence acceleration (WSA) model, has been used in radar-camera data fusion implementations with EKF [2]. As the name implies, the model assumes that the tracked target's acceleration varies gradually. Consequently, the model cannot accurately predict the target's states during rapid changes in acceleration. Compared to simpler motion models, such as constant velocity (CV), the CA model strikes a balance between accuracy and complexity.

The WSA model assumes that acceleration variations behave as white noise [24], meaning the values at any instance are uncorrelated with other instances. Since the acceleration random variable is assumed Gaussian, this uncorrelation implies independence [16], in other words, the acceleration variation is treated as an independent stochastic process.

For 2-dimensional motion, the original model lists the states as: $\boldsymbol{s} = \left(x, y, v_x, v_y, a_x, a_y\right)^T$. However, in our radar-camera project, the measurements are uncoupled to the acceleration states $a_x$and a_y. These states are unobservable even after multiple dynamic steps and are therefore omitted from the model. The resulting F matrix is [2, 24]:

$$\boldsymbol{F} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The process noise covariance matrix represents the noise or the deviation of the target from the modelled motion profile, which is the constant acceleration assumption. The expression for the process noise covariance matrix is shown below [2, 24]; Moreover, the model assumes the process noise for the pairs of states is independent. Its expression is given as [2, 24]. In the next section we discuss the data association methods considered.

$$\boldsymbol{Q} = var(\boldsymbol{w}) \cdot E(\boldsymbol{w} \cdot \boldsymbol{w}^T)$$

$$\boldsymbol{Q} = \begin{bmatrix} \sigma_{ax}^2 & 0 & 0 & 0 \\ 0 & \sigma_{ay}^2 & 0 & 0 \\ 0 & 0 & \sigma_{ax}^2 & 0 \\ 0 & 0 & 0 & \sigma_{ay}^2 \end{bmatrix} \times \begin{bmatrix} \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} & 0 \\ 0 & \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & 0 & \Delta t^2 & 0 \\ 0 & \frac{\Delta t^3}{2} & 0 & \Delta t^2 \end{bmatrix}$$

$$\mathbf{Q} = \begin{bmatrix} \frac{\sigma_{ax}^2 \, \Delta t^4}{4} & 0 & \frac{\sigma_{ax}^2 \, \Delta t^3}{2} & 0 \\ 0 & \frac{\sigma_{ay}^2 \, \Delta t^4}{4} & 0 & \frac{\sigma_{ay}^2 \, \Delta t^3}{2} \\ \frac{\sigma_{ax}^2 \, \Delta t^3}{2} & 0 & \sigma_{ax}^2 \, \Delta t^2 & 0 \\ 0 & \frac{\sigma_{ay}^2 \, \Delta t^3}{2} & 0 & \sigma_{ay}^2 \, \Delta t^2 \end{bmatrix}$$

Data Association Method Comparison

Data association methods aim to assign n observations to k tracks in a way that optimizes a given criterion. Some approaches are deterministic, optimizing a probabilistic likelihood, such as the Suboptimal Nearest Neighbour (SNN) or Global Nearest Neighbour (GNN). The SNN independently assigns each measurement to a track, which can sometimes result in conflicts where two tracks are assigned the same measurement. GNN resolves this by considering all assignments globally, eliminating such conflicts.

Other methods, such as Probabilistic Data Association (PDA), consider all possible associations of a measurement to a single target [25]. For n measurements, PDA generates $n + 1$ hypotheses: H_0 assumes none of the measurements belongs to target k, H_1 assumes measurement 1 belongs to target k while the rest are clutter, and so on [25].

While PDA is more robust than SNN or GNN because it handles assignment probabilistically, it is limited to single-target tracking. In multi-target scenarios, conflicts can still occur. This limitation is addressed by Joint Probabilistic Data Association (JPDA), which extends PDA by considering all combinations of track-to-measurement assignments jointly. The number of hypotheses increases combinatorially as n choose k [25, 26].

In general, the accuracy of methods increases from SNN → GNN, and PDA → JPDA. However, there is a trade-off between accuracy and computational demand. Considering the computational limitations of the Raspberry Pi in our project, we selected GNN, which provides acceptable performance even in fairly complex scenarios.

Machine Learning Model for Fog Level Estimation

To improve the robustness of the system under foggy weather conditions, we aim to enable the model to reliably estimate the fog level and scale the camera sensor autocovariance matrix accordingly. Several methods exist for fog detection, as fog affects various characteristics of an image. Some methods are more computationally intensive than others, making them less suitable for real-time applications.

The fog detection methods considered include:

(1) Determining the dark channel prior, a computationally heavy algorithm based on convolution [27, 28].
(2) Calculating the contrast using the standard deviation of the image pixels [28].
(3) Determining the entropy or randomness present within the image pixels [28].
(4) Computing the global contrast factor, which calculates contrast at multiple resolution levels for a better estimate [28].
(5) Measuring saturation, which conveys the colour level.
(6) Using Weber contrast, which compares the object intensity or colourfulness with respect to the background intensity [28].

Paper [29] implemented a support vector machine (SVM) framework for fog level estimation. The model was trained by associating fog density labels with the six features listed above, along with an additional feature: the distribution of the confidence scores of objects detected in a frame using YOLOv5l. The dataset used is SynFog [30], which produces synthetic images with corresponding fog density labels.

We propose employing a simpler ML model that uses the six-fog detection features along with the average confidence score of YOLO-detected objects to classify the scene as having dense or no fog. Fog density is treated as a classification problem because the SynFog dataset is not available; the dataset we found [31] contains 1500 images categorized as High, Medium or No Fog. To ease the problem, two categorizes where selected. Based on the classification, a mapping function (Table 2) is used to scale the camera autocovariance matrix.

Table 2. Mapping Function: Fog density classification to R_{cam} Scaling Factor

ML Model Classification Result	Scaling Factor (S.F.)
No Fog	0
High Fog	2

The updated camera autocovariance matrix is applied in the EKF update step, giving lower weight to the camera measurement under foggy conditions. Scaling factors can be tuned based on testing results.

$$R_{cam} = (1 + S.F.) \times R_{cam}$$

The ML models considered include Random Forest and Artificial Neural Networks (ANN). A single decision tree was not considered, as it is a simpler special case of a Random Forest. ANNs are known for capturing nonlinear relationships between features and outputs but are not easily interpretable [32]. Random Forests, on the other hand, are more comprehensible while still providing high accuracy [32]. In the next section we compare the two PoVs (BEV vs. FV) and detail the mathematics of the selected unified frame.

FV versus BEV

We previously discussed that FV is the default point of view (PoV) of the camera sensor module, while BEV, or top-down view, is the default PoV for the radar. We also discussed why it is essential to unify the PoV to a single coordinate frame. In this section, we discuss the rationale for selecting the fusion PoV, as well as its mathematical formulation.

In unified FV, the 3D radar measurements are projected into the camera image plane using the projective transformation matrix. In unified BEV, however, the 2D image pixel coordinates are projected into the radar 2D coordinate plane using the inverse projective mapping, or the inverse of the Homography matrix, a plane-to-plane transformation. While FV may seem superior to BEV because it preserves the depth information of the detected objects when projected onto the image plane, BEV assumes all detections lie on the same radar Z plane, losing true depth information and introducing distortion.

In target tracking applications, however, BEV is preferred. It is more robust against the crucial challenge of occlusion [1]. Moreover, in the literature on radar-camera data fusion, implementations tend to unify the coordinate frame to BEV. Therefore, we conclude to use BEV to project the YOLO-detected bounding box centroids into the radar plane $Z = z_{\text{projection}}$. The Homography matrix is derived from the projective transformation matrix as shown below:

$$k\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = P_{3\times4}\begin{bmatrix} X \\ Y \\ z_{projection} \\ 1 \end{bmatrix}$$

$$k\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} col_1(P_{3\times4}) & col_2(P_{3\times4}) & col_3(P_{3\times4}) \times z_{projection} + col_4(P_{3\times4}) \end{bmatrix}\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$H_{3\times3} = \begin{bmatrix} col_1(P_{3\times4}) & col_2(P_{3\times4}) & col_3(P_{3\times4}) \times z_{projection} + col_4(P_{3\times4}) \end{bmatrix}$$

The projected radar coordinate points obtained from the YOLO pixel coordinates are then calculated as follows. The k term is the scaling factor; dividing by k produces the normalized true (X, Y) coordinates:

$$\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \frac{1}{k} H_{3\times3}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

In the next section we compare different ground truth acquisition algorithms and detail the mathematics of the selected method.

Ground Truth Acquisition Method Selection

When deciding on a method to obtain ground truth values, we considered Lidar, RTK-GPS, and ArUco markers. After research, we found that although Lidar offers the required accuracy (2.5–10 cm), it is costly and would exceed the project budget [33]. Additionally, it has high computational demands, which is not ideal for our Raspberry Pi setup. RTK-GPS, while offering real-time position accuracy up to 1 cm [34], requires a base and rover, and can only be used outdoors due to satellite access requirements.

ArUco markers, in contrast, only require printed markers that the camera can detect using the OpenCV library. They provide high accuracy at close range, though accuracy decreases at longer distances [35]. Given these considerations, we selected ArUco markers for our project.

ArUco markers are square fiducial markers with a thick black border and an internal binary matrix [36], which encodes a unique identifier, enabling computer vision systems to quickly and reliably detect and differentiate individual markers. Their design makes them robust to partial occlusions and varying lighting conditions, making them ideal for precise localization [37]. In our project, we use ArUco markers to obtain the 3D location (x, y, z) of objects from camera frames, which allows comparison with radar-estimated locations. Previous research has demonstrated the effectiveness of ArUco markers for accurate object localization [38].

When a camera captures an image containing an ArUco marker, the system identifies the marker's corners in the 2D image plane. Knowing the marker's physical dimensions and the camera's intrinsic parameters allows us to compute its 3D position and orientation. For this, we determine the camera intrinsic matrix and distortion coefficients, which correct for radial and tangential lens distortions [39, 40]. These parameters are obtained during intrinsic calibration. The code used to generate the ArUco markers is given Appendix B, and the code used to extract the ground truth points from a given video is provided in appendix C.

Since our fusion is implemented in BEV (bird's-eye view), the ground truth points obtained in the camera coordinate frame must be projected into the radar coordinate frame, producing the corresponding x and y coordinates.

The extrinsic calibration step produces the rotation matrix $R_{radar \to camera}$, which rotates the radar coordinate frame to align with the camera, and the translation vector $t_{radar \to camera}$, which translates

the radar frame to match the camera's. To obtain BEV coordinates from the ground truth, we apply the inverse transformation from the camera frame to the radar frame. Since $R_{radar \rightarrow camera}$ is orthogonal, its inverse equals its transpose. The transformations are:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = R_{radar \rightarrow camera} \times \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} + t_{radar \rightarrow camera}$$

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = {R_{radar \rightarrow camera}}^{-1} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} - {R_{radar \rightarrow camera}}^{-1} \times t_{radar \rightarrow camera}$$

Where:

$$R_{camera \rightarrow radar} = {R_{radar \rightarrow camera}}^{-1} = {R_{radar \rightarrow camera}}^{T}$$

$$t_{camera \rightarrow radar} = -{R_{radar \rightarrow camera}}^{-1} \times t_{radar \rightarrow camera}.$$

Having summarized all the engineering decisions made, in the next section, we summarize the system constraints.

Project constraints

Based on the system components and the decisions made on this chapter, the system limitations are listed as given below.

1) **The Raspberry Pi's limited computational capacity.**

While the DSP module integrated within the radar EVM takes off some processing load, the computations required by the camera object detection method and the data fusion algorithm may still lead to disappointing results if not carefully optimized

2) **FoV mismatch between the sensors.**

The two sensors have different fields of view. The camera provides a shorter-range, narrower FoV, while the radar offers a longer-range, wider FoV. This may result in one sensor detecting objects that the other does not, complicating data fusion.

3) **Homography Transformation and BEV**

While BEV is preferred over FV due to better robustness against occlusion, depth information from the radar is lost. Additionally, the homography approach depends on the scene matching the assumption that camera measurements lie on the projected plane in the radar frame. When this is not the case, erroneous data may result.

4) **Offline calibration.**

Performing offline calibration assumes that the environment and sensor configuration remain unchanged. Small changes, including lighting or camera position, can introduce alignment errors. To address this, real-time calibration methods may be considered.

5) **Radar providing only high-level data**

In more cluttered environments, SNR analysis is typically used to subtract clutter accurately. Without access to raw-level radar data, the project cannot be easily extended to more complex scenes.

6) **Data association accuracy and load trade-off**

The selected data fusion algorithm, GNN, associates measurements to tracks deterministically, making it suitable for resource-limited embedded systems. However, it is less robust against clutter than more computationally intensive algorithms such as JPDA. This constrains the testing scenes to mostly simple track interactions.

7) **Ground-truth acquisition uncertainty**

The fusion output is benchmarked against ArUco-based ground truth, but there is always some uncertainty regarding the accuracy of the GT data itself.

Implementation and Fabrication

This chapter summarizes the work and results obtained in the project. It covers radar object detection, camera-based object detection using BSA, FOMO, and YOLO, the calibration process and applying those parameters to both the YOLO output and ground-truth datasets, the fusion algorithm logic, implementation considerations, and the data association process. We also include the Random Forest model for fog classification and conclude with a summary of the full algorithmic pipeline.

Radar Object Detection

As previously mentioned, the radar module outputs processed, high – level data that requires parsing. Therefore, reading the radar output, such as the speed and 3D location of the detected objects. This involved performing the following steps, setting up the radar module in the flashing mode and then on the functional mode, setting up the Raspberry Pi, verifying the recognition of the radar's port by the Raspberry Pi, downloading the required libraries and function script for the data parsing code, creating the radar configuration file and studying the parameters configured, testing the communication between the Raspberry Pi and radar module, and finally running the modified data parsing code.

The details of performing these steps are detailed in the report attached in Appendix D. The modified parsing code produces two plots, the XY or range vs. cross-range plot, as well as the velocity versus range plot.

Figures 9(a) – 9(l) shows the data parsing code output plots for the given scenario. When our team member (Engy) is not moving (figure 9(a)), the XY plot (figure 9(b)) shows a total of six detected objects, and the velocity vs. range plot (figure 9(c)) shows the detected objects to have a velocity of zero. When the team member starts to move (figure 9(d)), the velocity of three dots (at range = 4 m) becomes non – zero (figure f), and the XY plot (figure 9(e)) displays new points around the same range (points with coordinates (~4 m, ~1m)).

As the team member moves closer and closer (figure 9(g) and 9(j)), the range of the nonzero velocity points and the x- coordinates of the XY plot decreases, as can be observed in figures 9(h) and 9(k) for the XY plots, and figures 9(i) and 9(l) for the velocity versus range plots.

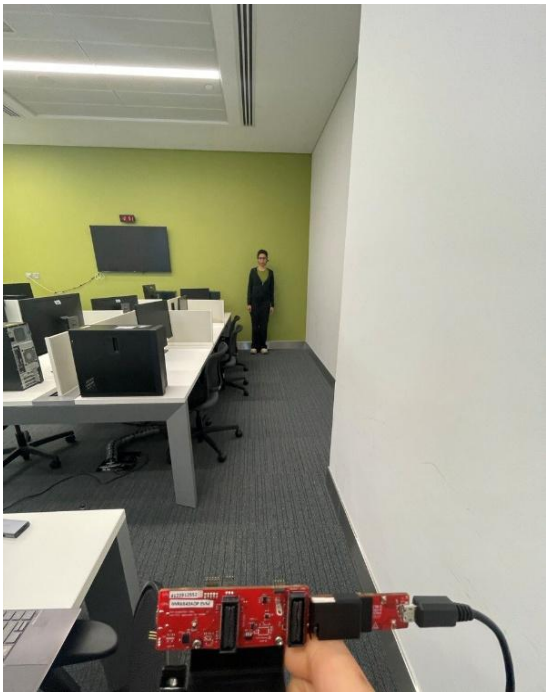


Figure 9(a). First scenario frame. Team member is static.

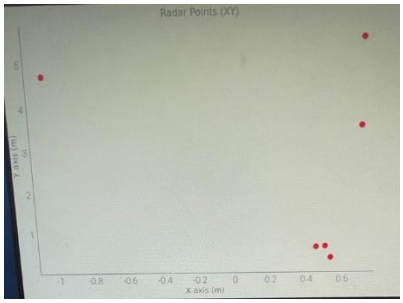


Figure 9(b). XY plot corresponding to the first scenario frame.

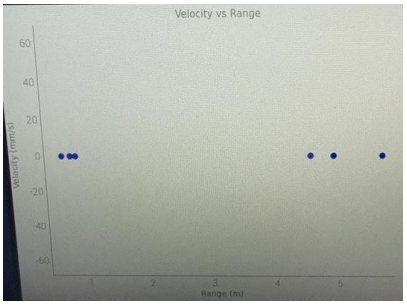


Figure 9(c). Velocity versus Range plot corresponding to the first scenario frame.

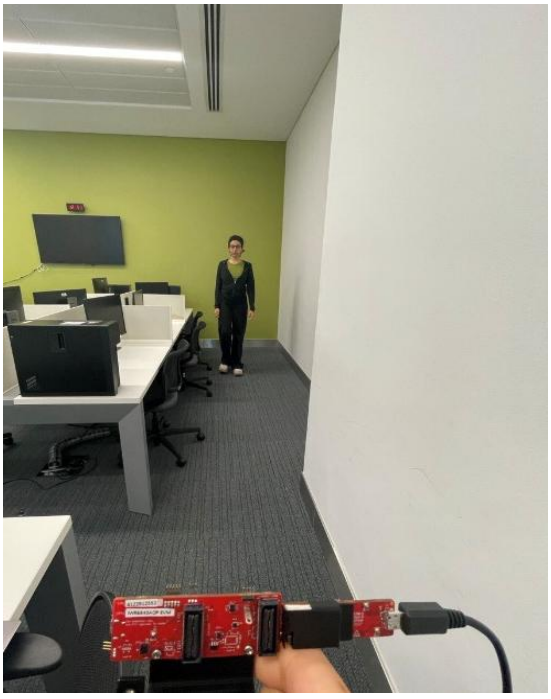


Figure 9(d). Second scenario frame.

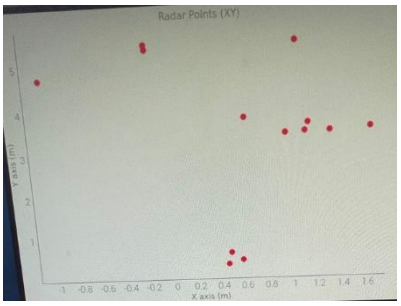


Figure 9(e). XY plot corresponding to the second scenario frame.

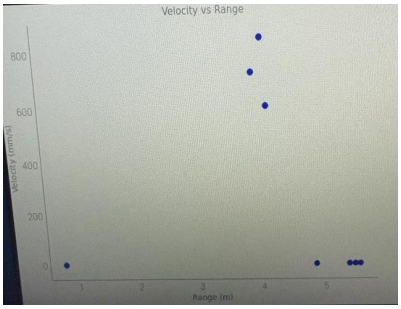


Figure 9(f). Velocity versus Range plot corresponding to the second scenario frame.

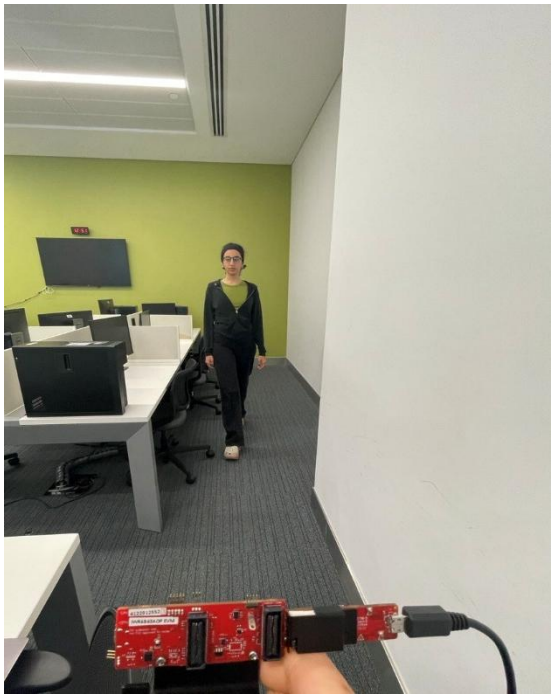


Figure 9g). Third scenario frame.

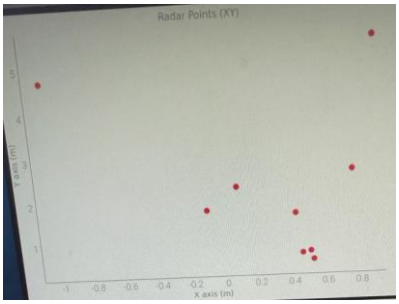


Figure 9(h). XY plot corresponding to the third scenario frame.

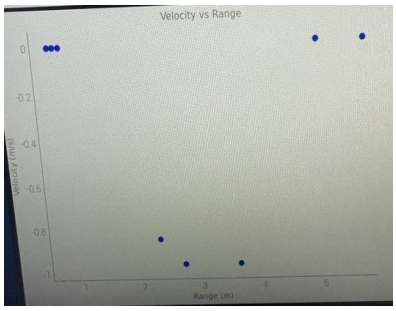


Figure 9(i). Velocity versus Range plot corresponding to the third scenario frame.

Figure 9(j). Fourth scenario frame.

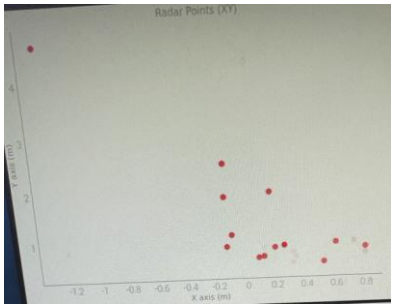


Figure 9(k). XY plot corresponding to the fourth scenario frame.

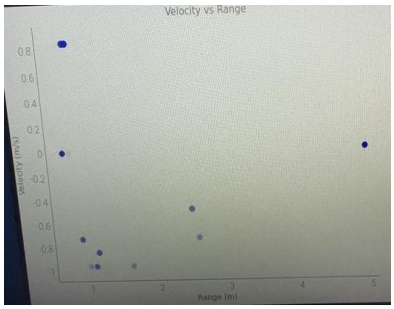


Figure 9(l). Velocity versus Range plot corresponding to the fourth scenario frame.

Camera Object Detection using Background Subtraction Algorithm

The background subtraction algorithm, combined with a Kalman filter for improved accuracy, was successfully implemented. The camera was configured through the libcamera stack, and a GStreamer pipeline was set up to enable video streaming directly into the Python environment via OpenCV.

The motion detection system was adapted to run on the Raspberry Pi. Background subtraction (MOG2) was used to isolate dynamic regions, while a Kalman filter smoothed the bounding box positions and dimensions over time. This helped mitigate the effects of lighting changes and camera noise, providing more stable tracking.

The final implementation produced two synchronized outputs: a live camera frame displaying bounding boxes and a foreground mask highlighting motion (Figure 10). The system demonstrated reliable real-time performance on the Raspberry Pi, confirming the feasibility of the approach for embedded computer vision applications. The Python code for this implementation is included in Appendix E.

Figure 10. Implementation of the background subtraction algorithm on the Raspberry Pi and the HQ- IR Cut camera. The Kalyan filter is integrated for accuracy improvements.

Camera Object Detection using FOMO

To implement object detection using FOMO, we collected our own dataset with the HQ-IR Cut camera. Sample images are shown in Figures 11 and 12. Model training was successfully performed; however, evaluation on the testing dataset produced 0% accuracy (Figure 13). Possible reasons include:

1. The dataset was too complex for the model, since cars and persons were often close together.
2. Initial difficulties connecting the camera to Edge Impulse delayed proper data collection, leading to simplified scenes being added.

Figure 11. A sample of the dataset collected for FOMO, showing cars

Figure 12. A sample of the dataset collected for FOMO, showing a car and a person.

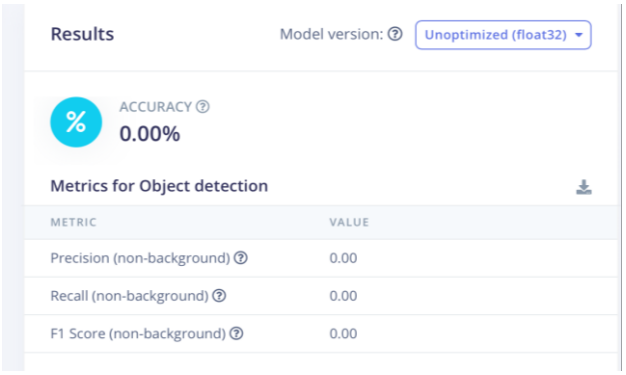


Figure 13. Edge Impulse, testing results.

Evaluation on the expanded dataset to include pictures of simpler scenes showed minimal improvement. As shown in the model evaluation on the training dataset, Figure 14, FOMO detected only a few cars, while many others were misclassified or missed entirely. Humans were completely undetected, as reflected by a 0 F1 score for the person class.

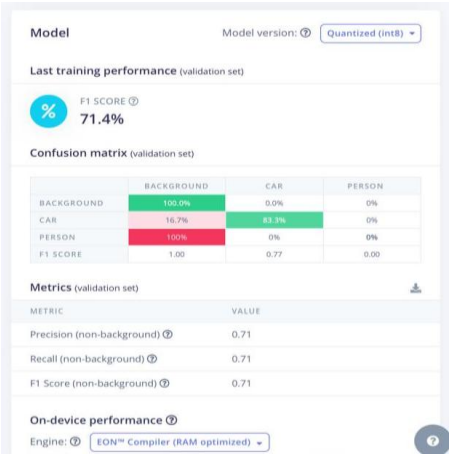


Figure 14. Edge Impulse, model evaluation on the training dataset after expansion.

Testing accuracy reached just 37.5%, as shown in Figure 15. These results demonstrate that FOMO's lightweight architecture lacks the representational capacity for our application, making it unsuitable for the project. The full implementation details are provided in Appendix F.

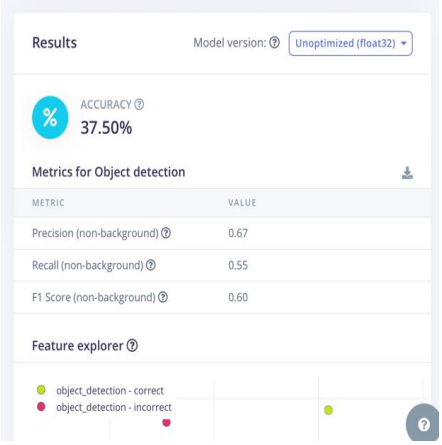


Figure 15. Edge Impulse, model evaluation on the testing dataset after expansion.

Camera Object Detection using YOLO

In implementing the YOLO method, we study its implementation with the integration of the TPU to enable low – latency performance. The implementation of YOLOv11 with the utilization of the Coral Edge TPU significantly improved frame processing speed, achieving an interval of approximately 40 ms per frame. The breakdown is as follows: preprocessing – 1.3 ms, inference – 40 ms, and post-processing – 1.3 ms. For the testing video example shown in Figure 16, the model maintained strong detection confidence, with the highest confidence score of 0.859 for the laptop and the lowest of 0.269 for the bottle.

Detection of people was consistent, with confidence scores of 0.768 and 0.733. Additional features, such as centroid coordinates, were displayed on screen, as shown in Figure 17. For example, the laptop's centroid was located at (490, 430), providing clear positional information within the frame needed later for the fusion process.

Note that real -time implementation of YOLO could not be achieved, this made the goal of the project to be switched from attempting real-time fusion, to only offline fusion. The report detailing the procedures and programming code for its implementation is given in Appendix G.

Figure 16. A frame of a testing video showing YOLO classification.

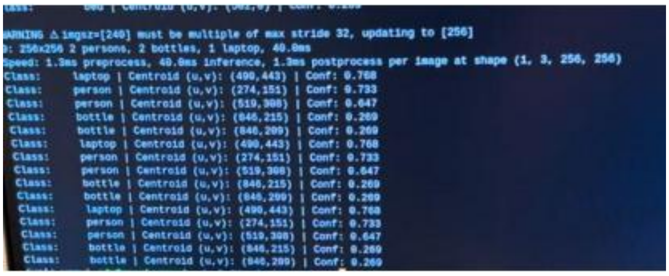


Figure 17. Output results on terminal of the YOLO implementation.

Multithreading Implementation

To ensure stable real-time acquisition from two asynchronous sensors (camera at ~30 FPS and radar at ~10 FPS), a multithreaded architecture was implemented on the Raspberry Pi. Sequential acquisition was not feasible: camera capture occasionally stalls due to encoding overhead, while radar UART parsing is latency-sensitive and susceptible to packet loss if reads are delayed. Independent threads were therefore required to guarantee uninterrupted operation and accurate timestamp alignment for later fusion.

Architecture Overview

The subsystem consists of three components:

Camera Acquisition Thread

Runs a scheduled 30 FPS capture loop using monotonic timers to maintain deterministic spacing. Each frame is logged with:

- frame index
- capture timestamp
- processing timestamp
- placeholder (u, v)

- A synchronized H.264 video is recorded in parallel. The thread includes basic recovery logic for transient capture errors. Logs are streamed to *camera_log.csv*.

Radar Acquisition Thread

Handles UART communication, applies the mmWave configuration, and continuously parses incoming TLV packets.

Main Coordination Thread

Initializes and monitors both sensor threads, manages system shutdown, and triggers:

- radar sensorStop
- closing of UART ports and camera resources
- flushing of all CSV buffers

This prevents locked devices or corrupted logs across sessions.

Timestamping and Synchronization

The system uses:

- monotonic timers for camera frame scheduling
- Unix timestamps for cross-sensor alignment

Camera timestamps are uniformly spaced; radar timestamps reflect true measurement time, forming a clean temporal base for fusion, calibration validation, and ground-truth extraction.

Tests showed 10 minutes of uninterrupted acquisition. The complete implementation (camera thread, radar thread, and main coordinator) is provided in Appendix H, including packet parsing, scheduling logic, thread lifecycle management, and all exception-handling routines.

Calibration Implementation

In this chapter, we present the system hardware model and the full implementation of both intrinsic and extrinsic radar–camera calibration.

Hardware Model

A stable hardware model is essential for reliable calibration. Any change in camera orientation, focus, or radar alignment invalidates the calibration, meaning the entire process must be repeated. To avoid this, we built a rigid mount that locks the radar and camera in fixed positions.

The two sensors were placed as close as possible to maximize the overlap of their fields of view. The model also houses the Raspberry Pi and the power bank. The final model is shown in Figure 18. The positioning of the two sensors in the setup is crucial; the radar antenna is mounted on its narrower side, which is the side kept closer to the camera. The radar comes with its own mount, but the camera

does not, a camera 3D stand was designed and 3D-printed with a 100%-infill (solid) mount to guarantee structural rigidity (Figure 19). In the next section, we discuss the details of the calibration procedure.

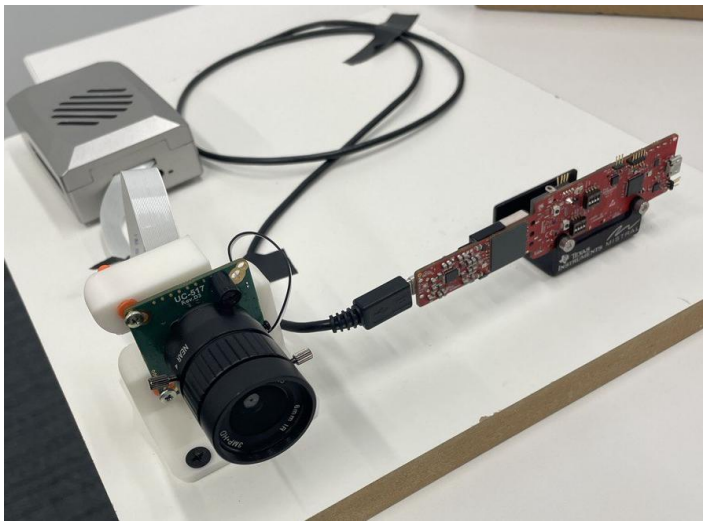


Figure 18. Hardware model.

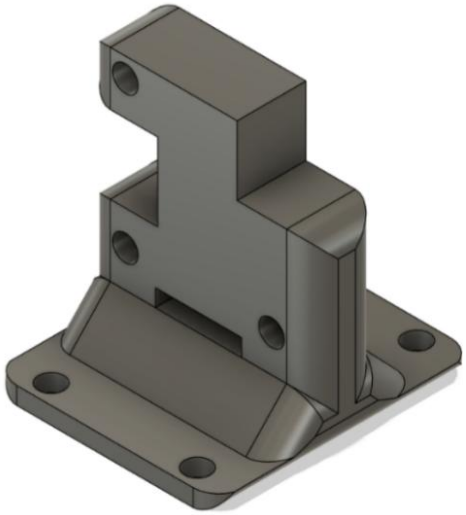


Figure 19. Camera stand 3D model.

Calibration Procedure

Multiple calibration approaches were reviewed [41-44]. The selected method was chosen based on radar capabilities and practical deployability. More advanced techniques may be explored in future work. The adopted procedure is based on [41], and the overall intrinsic–extrinsic flow is shown in Figure 20. The following sections summarize the intrinsic calibration implementation steps and results.

Intrnsic Calibration

The intrinsic parameters were determined using MATLAB's cameraCalibrator toolbox. Images were captured with the HQ–IR Cut camera and uploaded into the toolbox. The initial reprojection error was 0.29 px (Figure 21). After removing outlier images, the error dropped to 0.21 px (Figure 22). MATLAB's tools also allow visualization of the capture angles (Figures 23–24) and detection vs. reprojection points (Figure 25), showing minimal displacement.

The resulting intrinsic matrix and distortion coefficients are:

$$K = \begin{bmatrix} 4053.6085 & -12.6845 & 1979.8909 \\ 0 & 4046.2944 & 1577.1950 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{k} = [\ -0.4916362112 \quad 0.2651989268 \quad 0\ ]$$

$$\boldsymbol{p} = [-0.0029367783 \quad 0.0040783507\ ]$$

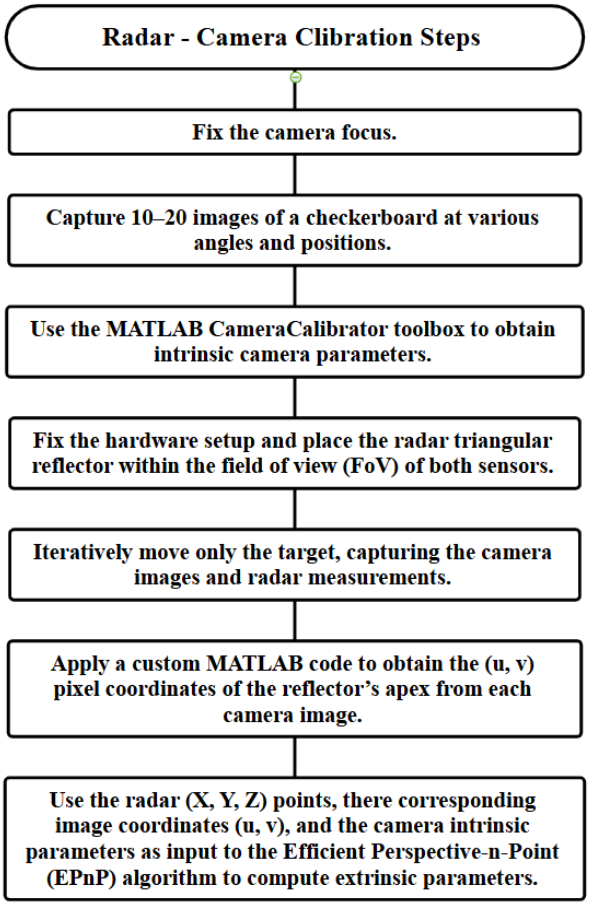


Figure 20. Full Radar-Camera calibration steps.

Extrinsic Radar-Camera Calibration

To obtain the extrinsic parameters, we follow steps 4–7 of the flowcharts presented in Figure 20. A minimum of six images, along with their corresponding radar measurements of the radar reflector, must be collected. It is important to ensure a relatively significant change in the reflector's position for each iteration. Figures 26, and 27 illustrate examples of the gathered radar and camera measurements.

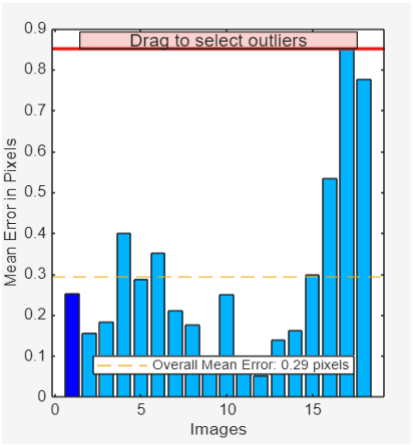


Figure 21. Initial mean reprojection error (in pixels) versus images chart

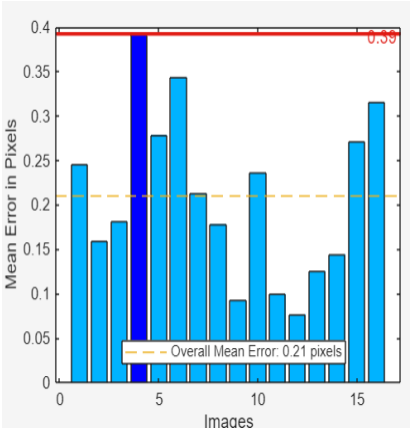


Figure 22. Final mean reprojection error (in pixels) versus images chart after adding more images and removing outliers.

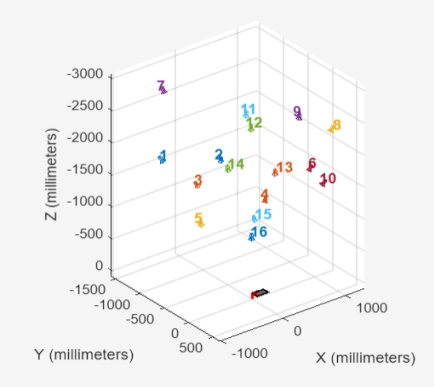


Figure 23. Extrinsic calibration illustration, showing the various angles at which the checkboard was captured.

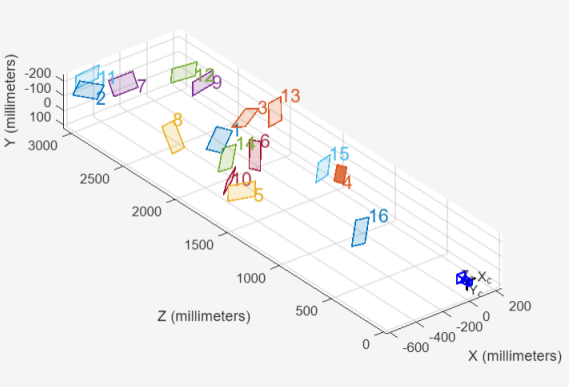


Figure 24. Extrinsic calibration illustration, showing the orientation of the checkboard with respect to the camera.

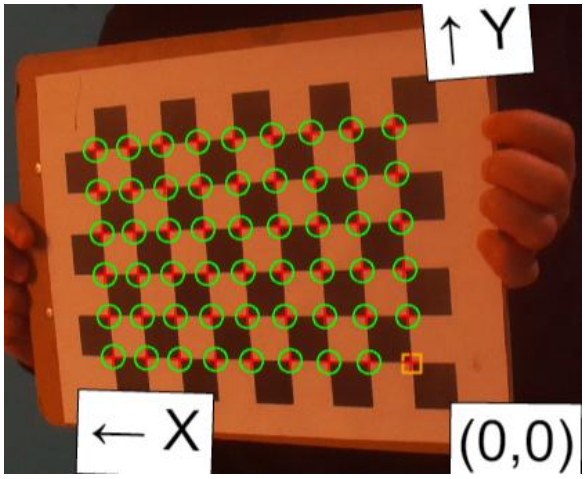


Figure 25. captured image with reprojected crosses plotted on the actual detected corners. This figure shows only a tiny reprojection error in pixels.

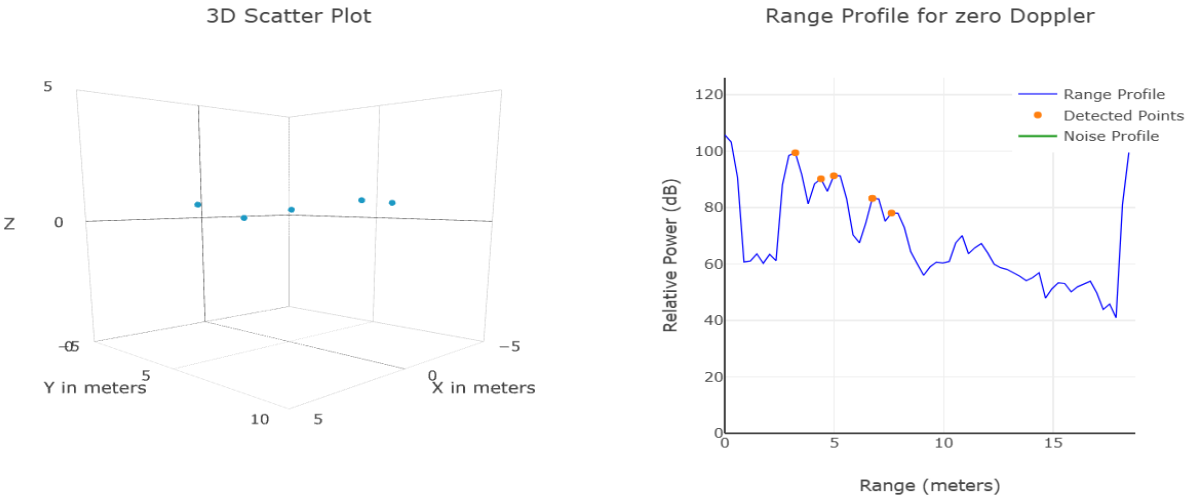


Figure 26. Radar measurement for a reflector in an extrinsic calibration setup on mmWave_Demo_Visualizer webpage

Figure 27. Reflector in Extrinsic Calibration Setup

Before proceeding further, it is necessary to confirm that the collected points are not planar, as this could result in degenerate solutions requiring additional points to be gathered. By visual inspection of the scatter plot, shown in Figure 28, we can verify that sufficient variation exists along all three axes.

Following this, the next step involves obtaining the (u,v) pixel coordinates corresponding to the apex of the radar reflector in each image. This is accomplished using the MATLAB script detailed in Appendix I. The corresponding pixel coordinates are displayed on the image, as shown in Figure 29, and printed to the command window.

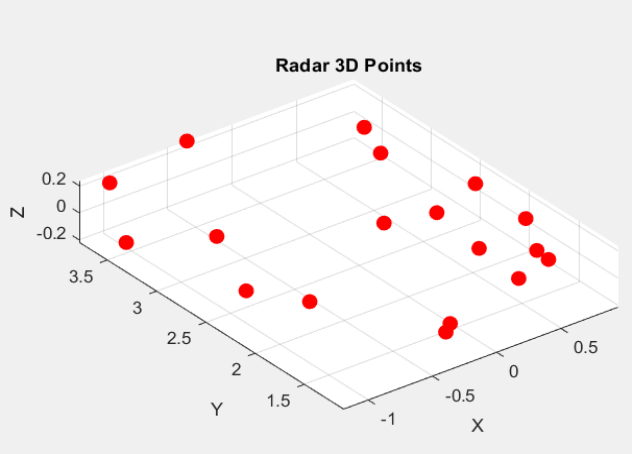


Figure 28. Radar measurement scatter plot used to verify whether there is sufficient variation along all three axes.

Figure 29. Image showing the pixel point corresponding to the apex of the corner reflector.

With the radar points, their corresponding pixel coordinates, and the intrinsic camera parameters available, the EPnP algorithm can be executed. A link to the GitHub repository containing the EPnP code is provided here [45]. Morover, the rotation (R) and translation (t) matrices obtained are given below.

$$R_{radar \rightarrow camera} = \begin{bmatrix} 0.9993975151 & -0.0000758999 & -0.0347073626 \\ -0.0347069625 & -0.0074618466 & -0.9993696751 \\ -0.0001831289 & 0.9999721571 & -0.0074599852 \end{bmatrix}$$

$$t_{radar \rightarrow camera} = [-0.1556986662 \quad 0.0378039639 \quad 0.0776697348]$$

The achieved reprojection error was 54.365 px (Figure 30). Although this may appear high, the image resolution is 4056×3040, and the fusion operates at the object level. Normalizing relative to the diagonal of the image gives: $\frac{54.365}{\sqrt{4056^2+3034^2}} \approx 0.0107$ approximatly 1% error, which is acceptable. In the next section we discuss the application of the obtained parametrrs both on the extracted ground truth states and the YOLO centroids pixels dataset.

Iteration	Func-count	Resnorm	First-order optimality	Lambda	Norm of step
0	7	65550	4.38e+05	0.01	
1	14	59580.4	1.08e+04	0.001	0.0319013
2	21	59577.2	15.9	0.0001	0.000527034
3	28	59577.2	0.18	1e-05	4.96677e-06

Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to its initial value is less than the value of the function tolerance.

<stopping criteria details>
error EPnP_Gauss_Newton: 54.365

Figure 30. EPnP algorithm output results.

Applying Projective Transformation to YOLO centroids and Ground Truth Measurements

Recall that the extracted ArUco markers coordinates are from the camera coordinate frame perspective. Hence extrinsic parameters are used to transform the ground truth measurements into the radar coordinate frame. The resulting coordinates were reasonable and closely matched the radar measurements, validating the calibration procedures, including the translational vector, intrinsic K matrix, and rotational matrix.

To obtain the radar measurements from the YOLO centroid measurements, the pixel coordinates were first resized back to the original resolution, undoing the resizing applied during YOLO implementation:

$$u_{true} = \frac{W_{original}}{W_{YOLO}} \times u'$$

$$v_{true} = \frac{H_{original}}{H_{YOLO}} \times v'$$

Next, the projective transformation matrix was computed as:

$$P_{3\times4} = K \times [R|t] = \begin{bmatrix} 4059.5297 & 1962.5466 & -139.2123 & -472.5638 \\ -132.0744 & 1549.6802 & -4054.7493 & 256.8588 \\ 0.0040 & 1 & -0.0069 & 0.0705 \end{bmatrix}$$

The Homography transformation matrix was computed as shown below:

$$H_{3\times3} = \begin{bmatrix} 4059.5297 & 1962.5466 & -139.2123 \times z_{projection} - 472.5638 \\ -132.0744 & 1549.6802 & -472.5638 \times z_{projection} + 256.8588 \\ 0.0040 & 1 & 0.0069 \times z_{projection} + 0.0705 \end{bmatrix}$$

Homography transformation involves assuming all measurements received are planar, lying on some Z plane in the radar coordinate frame. When choosing Z as zero, meaning all the YOLO detections centroids obtained are projected into the z = 0 plane, the respective points in the radar coordinate frame were not reasonable (negative y-values). Adjusting Z further worsened the errors.

Since the ground truth results were valid, the issue was determined to be the Homography approximation rather than the calibration process. Deviations from the assumed plane increase error. In our implementation, the model was placed on a table. In future implementations, the model should be placed on the ground (as observed in [2]), so that the scene better conforms to the homography approximation. In the nest section, we discuss the fusion algorithm implementation.

Fusion Implementation

This chapter discusses the selected fusion implementation (asynchronous fusion) based on the results of the implementation of camera object detection using YOLO and radar object detection. We also discuss how out of sequence measurements are handled in asynchronous fusion. We then cover the application of the EKF specifically for radar–camera fusion and finally present the estimations for the radar and camera noise covariance matrices.

Selected Fusion Algorithm Implementation

In our application, the implemented YOLO object detection algorithm on the Raspberry Pi achieved an update rate of 70 FPS for a simulation video. However, since the HQ IR-Cut camera used has a 30 FPS output rate, the YOLO output rate is limited by the camera and should achieve an update rate of 30 FPS.

For the radar, the data transmission rate via the UART port can be controlled and adjusted via the configuration file which is sent to the radar module during deployment. The configuration file can be customized using Texas Instruments' mmWave_Demo_Visualizer browser-based application; a screenshot of the application is shown in Figure 31. The figure shows the FPS rate along with other adjustable parameters.These parameters are interrelated, the table in the report in Appendix D summarizes these interdependencies based on [46]. For example, in the "best range" scene (as opposed to best velocity or best range-resolution scenes), increasing the maximum unambiguous range coarsens the range resolution.

For the desirable configuration, we opted for the best_range scene. In the best-velocity scene, the maximum possible radial velocity was 3.84 m/2, which is unsatisfactory for pedestrian/car tracking. In

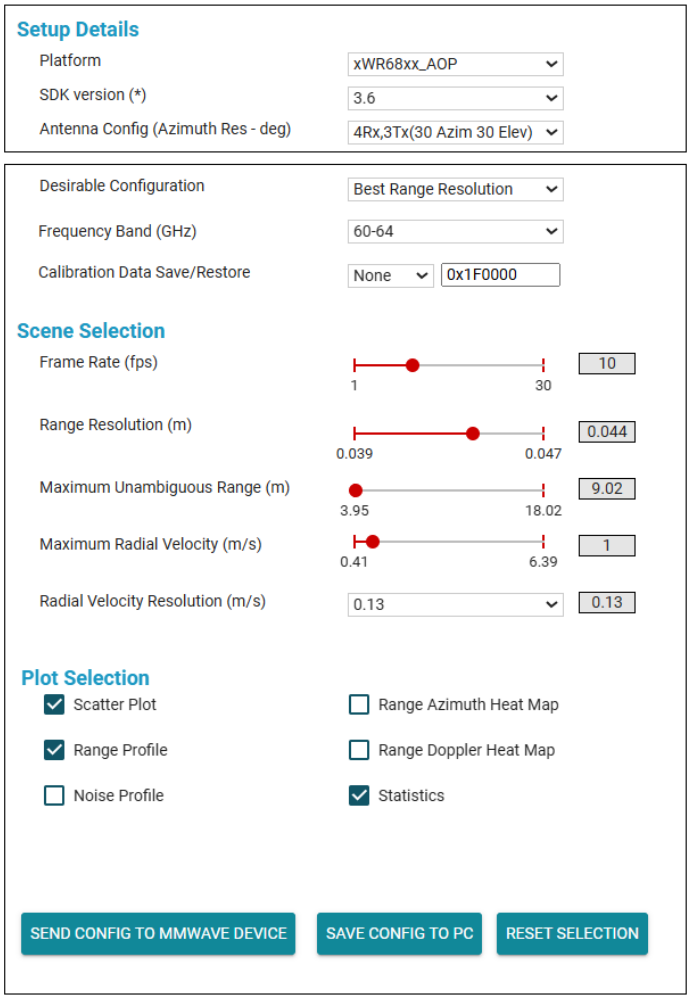


Figure 31. Screenshpt of the mmWave_Demo_Visualizer browser-based application

the best-range-resolution scene, for a maximum radial velocity of 8 m/s (again, well below car velocities of ~30 m/s), the radial velocity resolution was 1.09 m/s, which is again unsatisfactory, since pedestrians average velocity is about 0.95–1 m/s [47], and under those configurations the radar could detect a single target that actually corresponds to two closely spaced pedestrians moving at similar speeds.

The best_range scene provided a reasonable trade-off for this application. The summary of the chosen configuration file is shown below. Note that the radar FPS was kept at 10 FPS, the higher end of the recommended range. According to [6], the frame period T_f can be expressed as $NT_c + T_{\text{IFT}}$, where T_c is the chirp time or sweep time and T_{IFT} is the inter-frame time.

During T_{IFT} the radar processes and transmits captured data. A higher frame rate reduces T_{IFT}, which may lead to data loss if T_{IFT} is insufficient to transmit the captured data over the relatively low baud rate of the UART transmission protocol; it is therefore suggested to select a relatively low frame rate (3 – 10 fps) [6].

% Frequency:60
% Platform:xWR68xx_AOP
% Scene Classifier:best_range
% Azimuth Resolution(deg):30 + 30
% Range Resolution(m):0.293
% Maximum unambiguous Range(m):15
% Maximum Radial Velocity(m/s):19.43
% Radial velocity resolution(m/s):0.31
% Frame Duration(msec):100
% RF calibration data:None
% Range Detection Threshold (dB):15
% Doppler Detection Threshold (dB):15
% Range Peak Grouping:enabled
% Doppler Peak Grouping:enabled
% Static clutter removal:disabled
% Angle of Arrival FoV: Full FoV

Now that the update rates of each sensor are determined, we can determine the best fusion method, synchronous or asynchronous. Since there is a considerable difference between the update rates of YOLO and the radar, we opt for asynchronous fusion. Applying interpolation would introduce considerable uncertainty and increase error, reducing the reliability of the EKF tracker. The second option of dropping YOLO outputs until the radar is ready would introduce significant latency.

The asynchronous fusion logic is shown in Figure 32. The EKF is initialized ($k = 0$); the system then waits for a measurement. Once a measurement is received, $\boldsymbol{\Delta t}$ is determined and $F_{k-1}$and $Q_{k-1}$are computed. The EKF performs the prediction step for sample k, then the states are updated using the received measurement for sample k. After that, the sample index k is incremented, and the system waits for the next measurement.

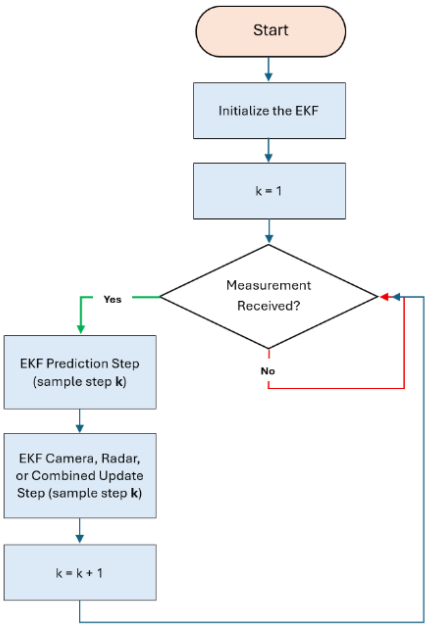


Figure 32. Extended Kalman Filter Asynchronous Logic Flowchart

As discussed before, asynchronous fusion faces the possibility of receiving a stale or out-of-sequence measurement; this is when a measurement received corresponds to a frame at time t_l, while the tracker has already updated its states using a frame corresponding to $t = t_k$, with $t_k > t_l$. The handling of Out of Sequence Measurements (OOSM) is discussed in the next section.

Handling Out of Sequence Measurements

According to sources [25], four approaches exist for handling stale measurements that arrive within 0.5 seconds of the last EKF update. Measurements older than this threshold are discarded. Some approaches require computationally heavy corrections unsuitable for real-time applications.

The simplest approach is to discard stale measurements [25], but this risks losing valuable information. Therefore, we adopt the second-simplest method: which is to retrodict back to the time of the frame that measurement received corresponds to [25].

The retrodicted a priori state is:

$$\boldsymbol{x_l^-} = \boldsymbol{F_{k-1}^{-1}} \cdot \boldsymbol{x_k^-}$$

The priori estimation error covariance matrix remains referenced to time t_k, as given below, yet the process noise covariance matrix, $\boldsymbol{Q}_{k-1}$, is assumed to be zero.

$$\boldsymbol{P_k^-} = \boldsymbol{F_{k-1}^{-1}} \boldsymbol{P_{k-1}^+} (\boldsymbol{F_{k-1}^{-1}})^T$$

In the update step, the posteriori state is determined as given below. The Kalman gain and the posteriori estimation error autocovariance matrix remain unchanged and referenced to time t_k.

$$\boldsymbol{x_k^+} = \boldsymbol{x_k^-} + \boldsymbol{K_k}(\boldsymbol{y_l} - \boldsymbol{H_k x_l^-})$$

$$\boldsymbol{K_k} = \boldsymbol{P_k^- H_k^T} \cdot (\boldsymbol{H_k P_k^- H_k^T} + \boldsymbol{R_k})^{-1}$$

$$\boldsymbol{P_k^+} = (\boldsymbol{I} - \boldsymbol{K_k H_k}) \boldsymbol{P_k^-} (\boldsymbol{I} - \boldsymbol{K_k H_k})^T + \boldsymbol{K_k R_k K_k^T}$$

While this is relatively simple, it falls short in its low accuracy, as it reduces the mathematical complexity of the equation when $\boldsymbol{Q}_{k-1}$ is not zero, which is not an accurate assumption. Nonetheless, it avoids completely discarding a stale measurement which can provide new valuable information. In the next section, we discuss the application of the EKF in radar-camera fusion system.

The EKF and Radar-Camera Data Fusion

In our application, the radar provides x, y, and v_r measurements, while the camera outputs u, v pixels (which are coordinate-transformed to x, y measurements in the radar coordinate frame). The tracked state vector is defined as shown below.

$$\boldsymbol{s} = \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}$$

The radial velocity is not linearly related to the states x, y, v_x and v_y requiring the usage of the EKF. The equation below expresses the radial velocity as a function of the states [2]:

$$v_r = \frac{xv_x + yv_y}{\sqrt{x^2 + y^2}}$$

To linearize this measurement, we apply the Jacobian to v_r, evaluated at the priori state estimate:

$$\left[\frac{\partial v_r}{\partial s}\right]_{\hat{\boldsymbol{x}}_{k-1}^-} = \left[\frac{\partial v_r}{\partial x} \quad \frac{\partial v_r}{\partial y} \quad \frac{\partial v_r}{\partial v_x} \quad \frac{\partial v_r}{\partial v_y}\right]_{\hat{\boldsymbol{x}}_{k-1}^-}$$

The resulting expression is [2]:

$$\left[\frac{\partial v_r}{\partial s}\right]_{\hat{\boldsymbol{x}}_{k-1}^-} = \left[\frac{y(yv_x - xv_y)}{(x^2+y^2)^{3/2}} \quad \frac{x(xv_y - yv_x)}{(x^2+y^2)^{3/2}} \quad \frac{x}{\sqrt{x^2+y^2}} \quad \frac{y}{\sqrt{x^2+y^2}}\right]_{\hat{\boldsymbol{x}}_{k-1}^-}$$

The measurement matrix and measurement vector for the radar are:

$$\boldsymbol{H_{radar}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \frac{y(yv_x - xv_y)}{(x^2+y^2)^{3/2}} & \frac{x(xv_y - yv_x)}{(x^2+y^2)^{3/2}} & \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \end{bmatrix}$$

$$\boldsymbol{y}_{radar} = \begin{bmatrix} x \\ y \\ v_r \end{bmatrix}$$

For the camera, the transformed (u, v) pixels directly provide us x, and y. Therefore, the $\boldsymbol{H}_{cam}$ and $\boldsymbol{y}_{cam}$ are given as shown below:

$$\boldsymbol{H_{cam}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$\boldsymbol{y}_{cam} = \begin{bmatrix} x \\ y \end{bmatrix}$$

This defines the output function $h(x_k)$ and its Jacobian. In the next section, we discuss the estimations of the noise covariance matrices for the radar and camera sensor modules.

The R_{cam}, R_{radar} Covariance Matrices Estimations

The autocovariance matrices define the measurement noise present, and accordingly how confident the EKF should be in each sensor. Ideally, $\boldsymbol{R}_{radar}$ and $\boldsymbol{R}_{cam}$ are sensor-dependent obtained through collecting a dataset from the specific sensor for a given scenario, and based on known ground truth,

comparing the two sets, the autocovariance matrices can be computed. A simpler yet less accurate approach is to estimate the autocovariances from the general sensor knowledge.

Radar measurement includes noise due to several sources. Source [2] lists the contributions from: (1) multipath effects, (2) processing noise including FFT, (3) inherent measurement noise represented by the received signal SNR, which is modelled by Gaussian distribution. (4) angle-dependent antenna gain. However, noise due to (1), (2), and (4) can be mitigated via the TI IWR 6843 advance processing capabilities; source [2] accounts for the contribution of the source (3) only in the estimation of the autocovariance matric. The variances of phase, range, and Doppler are estimated as one-fourth of the radar's resolutions.

The radar, TI IWR6843 outputs x, y, and vr, measurement instead of range, azimuth, and Doppler. The uncertainty or standard deviation in the measurements x and y (cartesian coordinates) can hence be determined from the uncertainty in the range and azimuth measurements (spherical coordinates) using the propagation of error equation given below [48].

$$\sigma_z^2 = \left(\frac{\partial z}{\partial w}\right)^2 \cdot \sigma_w^2 + \left(\frac{\partial z}{\partial x}\right)^2 \cdot \sigma_x^2 + \left(\frac{\partial f}{\partial y}\right)^2 \cdot \sigma_y^2 + \cdots$$

The standard deviation in the x measurement can be computed as shown below.

$$x = \rho \cos(\theta)$$

$$\sigma_x^2 = \left(\frac{\partial \rho \sin(\theta)}{\partial \rho}\right)^2 \cdot \sigma_\rho^2 + \left(\frac{\partial \rho \cos(\theta)}{\partial \theta}\right)^2 \cdot \sigma_\theta^2$$

$$\sigma_x^2 = (\cos(\theta))^2 \cdot \sigma_\rho^2 - (\rho \sin(\theta))^2 \cdot \sigma_\theta^2$$

$$\sigma_x^2 = (\cos(\theta))^2 \cdot \sigma_\rho^2 - (\rho \sin(\theta))^2 \cdot \sigma_\theta^2$$

For the y measurement, the standard deviation is given as shown below:

$$y = \rho \sin(\theta)$$

$$\sigma_y^2 = (\sin(\theta))^2 \cdot \sigma_\rho^2 + (\rho \cos(\theta))^2 \cdot \sigma_\theta^2$$

The radar autocovariance can be given as shown below. The 0s elements in $\boldsymbol{R}_{radar}$ indicate that the noise present in each measurement (x, y, and vr) are uncorrelated.

$$\boldsymbol{R}_{radar} = \begin{bmatrix} (\cos(\theta))^2 \cdot \left(\frac{\Delta\rho^2}{4}\right) - (\rho \sin(\theta))^2 \cdot \left(\frac{\Delta\theta^2}{4}\right) & 0 & 0 \\ 0 & (\sin(\theta))^2 \cdot \left(\frac{\Delta\rho^2}{4}\right) + (\rho \cos(\theta))^2 \cdot \left(\frac{\Delta\theta^2}{4}\right) & 0 \\ 0 & 0 & \frac{\Delta v_d^2}{4} \end{bmatrix}$$

As for $\boldsymbol{R}_{cam}$, camera measurement noise is assumed uncorrelated and dependent on the reprojection errors from the extrinsic calibration. Let Δd be the diagonal reprojection error; then

$$\Delta u = \Delta v = \frac{\Delta d}{\sqrt{2}}$$

$$\begin{bmatrix} \Delta x \\ \Delta y \\ 1 \end{bmatrix} = \frac{1}{k} \times H_{3\times 3}^{-1} \times \begin{bmatrix} \Delta u \\ \Delta v \\ 1 \end{bmatrix}$$

$$\boldsymbol{R}_{cam} = \begin{bmatrix} (\Delta x)^2 & 0 \\ 0 & (\Delta y)^2 \end{bmatrix}$$

In the next section, we discuss the data association algorithm GNN, used to associate the received measurement with the established tracks.

The GNN and the Target Tracking Block Diagram

The Kalman filter is a primary component in target tracking. Figure 33 shows a block diagram of the logic and steps used in implementing target tracking. The flow chart is constructed based on [2, 25, 49].

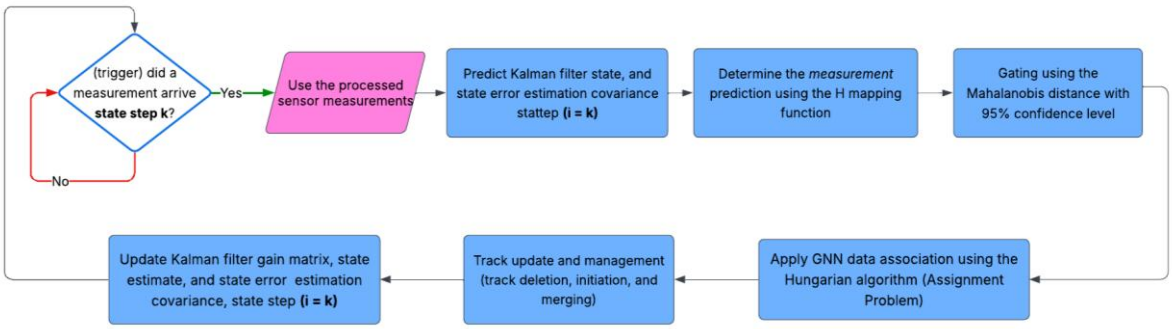


Figure 33. Target Tracking and Data Association Block Diagram.

Since asynchronous fusion is used, the pipeline is triggered upon the arrival of a new measurement from a sensor. The measurement is processed, and the prediction for sample step k is performed (sample step **does not** refer to the clock timestamp of the instant at which the prediction or update was performed).

The state prediction for each target is then used to produce the measurement prediction for each target/track. The Jacobian of the function $h(x_k)$, H, is used to map the state predictions to the measurement prediction space. Gating is then performed to eliminate outliers for a given track, thereby reducing computations required in the association steps [25, 26]. The data association method is then applied, followed by the track management logic, including initiation and deletion of tracks. After that, the Kalman filter update step is performed for sample step k.

In the following sections, we explain the gating step, the data association method selected (GNN), and the track management logic layer.

Gating

The initial step of DA and target tracking is the gating step. As mentioned, it is used to reduce how many measurements-to-track associations are considered [25, 26]. The gating is based on the Mahalanobis distance equation given below; the equation is dependent on the innovation term. The greater the innovation term, the greater the difference between a given received measurement and the predicted measurement for a given track, indicating a smaller likelihood that that measurement should be associated with that track.

$$d_{ij}^2 = (y_j - H\hat{x}_i^-)^T \cdot S^{-1} \cdot (y_j - H\hat{x}_i^-)$$

The equation is also dependent on the inverse of the autocovariance matrix of the innovation term; the smaller the variance, the lower noise is present and hence the more confident we are about the received measurement, and hence the more weight it is given in the sum of the squares of the distance for a given n – dimensional multivariant RV y [25].

A measurement is said to be an outlier with 95% confidence based on the equation given below [25, 50]. We obtain the value of G from the chi-squared distribution with n degrees of freedom [25, 50]. In our case, we are receiving 3 measurements from the radar (x, y, and v_r), and hence $n = 3$, G is approximately 7.815. For the camera with $n = 2$, G is approximately 5.991.

$$(y_j - H\hat{x}_i^-)^T \cdot S^{-1} \cdot \left(y_j - H\hat{x}_i^-\right) < G \approx 7.815 \; for\; Radar$$

$$(y_j - H\hat{x}_i^-)^T \cdot S^{-1} \cdot \left(y_j - H\hat{x}_i^-\right) < G \approx 5.991 \; for\; camera$$

Global Nearest Neighbour (GNN)

In GNN, we choose the data association hypothesis Ω^k which has the highest probability [25]. We construct the probability of a data association algorithm as follows. The probability distribution of an innovation term is given by g below. The probability that a given measurement y_j should be associated with a prediction $H\hat{x}_i^-$ decreases as the difference increases, and the more spread the innovation probability distribution is.

$$g = \frac{1}{\sqrt{(2\pi)^n |C|}} e^{-\frac{1}{2}(y_j - H\hat{x}_i^-)^T \cdot S^{-1} \cdot \left(y_j - H\hat{x}_i^-\right)}$$

The probability of the occurrence of independent events is equal to the multiplication of the probabilities of the individual events; hence, a data association hypothesis probability is given as shown below, $f(\Omega^k)$. This reads as: the probability that measurement y_1 is associated with prediction $H\hat{x}_1^-$ ***and*** measurement y_2 is associated with prediction $H\hat{x}_2^-$, and so on [25].

$$f\left(\Omega^k\right) = \prod_{(i,j)\in\varphi_h} \frac{1}{\sqrt{(2\pi)^n|C|}} e^{-\frac{1}{2}d_{ij}^2}$$

A question arises: how can we find the data association hypothesis that optimizes the probability expression given above? The expression given above is the maximum likelihood estimation (MLE) expression. If we take the natural log on both sides, we obtain the following expression:

$$ln\left[f\left(\Omega^k\right)\right] = -\frac{nj}{2}\ln[2\pi] - \frac{j}{2}\ln[C] - \frac{1}{2}\sum_{(i,j)\in\varphi_h} d_{ij}^2$$

To maximize the above expression, is to minimize $j\ln[C] + \sum_{(i,j)\in\varphi_h} d_{ij}^2$. According to [50], we obtain the same results if we only minimize $\sum_{(i,j)\in\varphi_h} d_{ij}^2$; which is to select the data association hypothesis resulting in the smallest possible sum of squared distances of the n-dimensional measurement to a given track's prediction position.

But how to practically determine the associations that result in the maximum likelihood probability? To answer this, let us consider two scenarios, in the first assume we have three targets, each produces their own ellipsoidal gates of uncertainty. The targets are relatively far apart and there is no overlap in their respective gates. In this case, solving the optimal association problem is relatively simple. Track j is matched to closest observation i in its ellipsoid, the rest of the observations are considered to be potential detected targets or clutter. A more complex scenario is given in Figure 34, adapted based on [25, 26, and 50]

One way is to list all possible associations, which would be equal to n (measurements) – choose – k (tracks), prior to gating. However, after gating, we roughly only need to consider the sum of n measurement per gate (excluding cases of conflicts, multiple measurements assigned to the same track); and measurements outside the gates are considered potential new targets or false alarms.

Practically this corresponds to constructing a matrix with the tracks as rows and the measurements are the columns. The matrix is filled using the logic given below (based on [50]), and the table we obtain for the scenario given in Figure 34 is given in Table 3.

For each track T_k of the set of tracks:
    If P_i is within the gate of T_k:
        Compute d_{ij}^2 as the squared distance from predicted position of T_k to P_i
    Else:
        set d_{ij}^2 to 100

We obtain what is referred to as the cost function, and the problem is to determine the global optimal assignment such that the sum of the corresponding cost function elements is minimal. This is solved using the Hungarian algorithm, discussed in the following section.

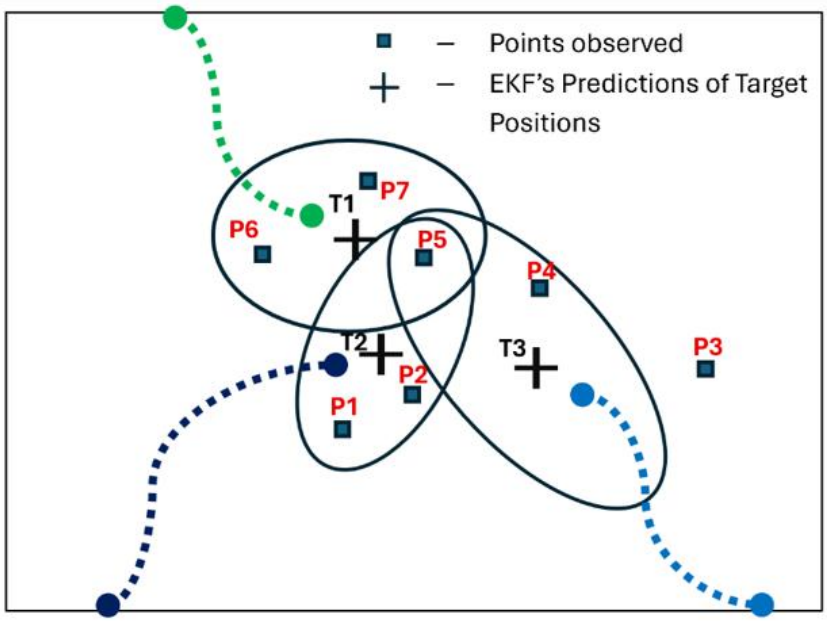


Figure 34. Target Tracking and Data Association Problem Scenario

Table 3. Cost Matrix for Figure 34 Scenario

	P1	P2	P3	P4	P5	P6	P7
T1	100	100	100	100	d_{51}^2	d_{61}^2	d_{71}^2
T2	d_{12}^2	d_{22}^2	100	100	d_{52}^2	100	100
T3	100	100	100	d_{43}^2	d_{53}^2	100	100

The Assignment Problem

The assignment is formulated as given below [26]. The condition of the M matrix makes it a permutation matrix, resulting in only a non-zero element (one) per column and row, hence resulting in a single assignment for each track, and avoiding the occurrence of conflicts. The Hungarian algorithm allows us to obtain the assignment resulting in the smallest possible total cost without needing to consider all possible combinations. The Hungarian algorithm solves the assignment problem in a polynomial time $O(n^3)$ [51].

$$Total\ Cost = \sum_{i=1}^{N}\sum_{j=1}^{N} M_{ij}\, C_{ij}$$

$$\forall i \sum_{j=1}^{N} M_{ij} = 1 \quad and\ \forall j \sum_{i=1}^{N} M_{ij} = 1$$

$$M_{ij} \in \{0, 1\}$$

Based on [26], the Hungarian algorithm steps are as given below.

1. Pad extra rows or columns with rows to construct an N x N square matrix. In the example given in Table 3, we would add four more rows with zeros
2. Subtract the minimal element from each column
3. Subtract the minimal element from each row
4. Draw the minimalist possible number of column and rows to cover the zeros, if they are N lines, then we created our permutation matrix, otherwise, we proceed to step 5.
5. Determine the minimum number from the elements that are not crossed by the lines. Subtract this number from the group of elements that are not crossed by the lines and add this element to the elements at the intersection of the drawn lines.
6. Check again if we created a permutation matrix, otherwise, go back to step 4.

According to [50], GNN may assign a track with an empty gate a measurement that is outside of its gate. We handle this limitation using track management logic, where tracks with empty gates are flagged as partially inactive and their rows is deleted from the constructed cost function before the Hungarian is applied. Track management is discussed in an upcoming section. In the following section, we describe a special case where the EKF is updated using the combined radar and camera measurements.

Special Case of the EKF implementation: Combined Measurement Update

In this section, we discuss how a special case of received measurements is handled. Whenever a camera and the radar capture a measurement of nearly the same frame and arrive within a 0.05 sec delay window (representing capture and processing latency), the EKF update is performed jointly, updating a track's state estimates using the combined camera and radar measurements, this case is flagged as case 3. Radar-alone update is flagged as case 2, while camera-alone update is flagged as case 1.

In the case 3 flag, the measurement prediction step is performed assuming both camera and radar measurements (using $H_{\text{cam}}$and H_{radar}). In the GNN step of constructing the cost function, both the radar and camera cost functions are constructed independently. After that, the Hungarian algorithm is applied to both cost functions. For tracks which get assigned both a camera measurement and a radar measurement, their assignment count increments, and the assigned measurement is saved as shown below.

$$y_{assigned} = \begin{bmatrix} y_{assigned,radar} \\ y_{assigned,cam} \end{bmatrix}$$

For tracks assigned only either a camera or a radar measurement (meaning a sensor sees a measurement that the other sensor does not detect), their deletion count increments, and the EKF update step is

skipped for these tracks. Note that the H, R, and $y_{\text{predicted}}$ matrices and vector are formulated in the same way as for synchronous fusion, this was discussed in the Synchronous and Asynchronous Data Fusion section. In the next section, we discuss track management.

Track Management

Track management involves a logic layer that handles track deletion and initiation [49]. After the GNN data association step, measurements left unassigned are initiated as potential new tracks or false alarms, and whenever a track is not assigned any measurement (empty gate) for 30 consecutive frames, the track is deleted [49].

We use a structure of Tracks; the detection field is incremented every time a track is not assigned any measurement and is reset to zero every time the track is assigned a measurement. The assignment field is incremented every time a given track is assigned a measurement. When the assignment exceeds 30, the track is said to be established and is displayed as a confirmed track.

The block diagram given in Figure 35 provides a comprehensive description of how tracks are managed for all three cases: camera-alone, radar-alone, and combined EKF update steps.

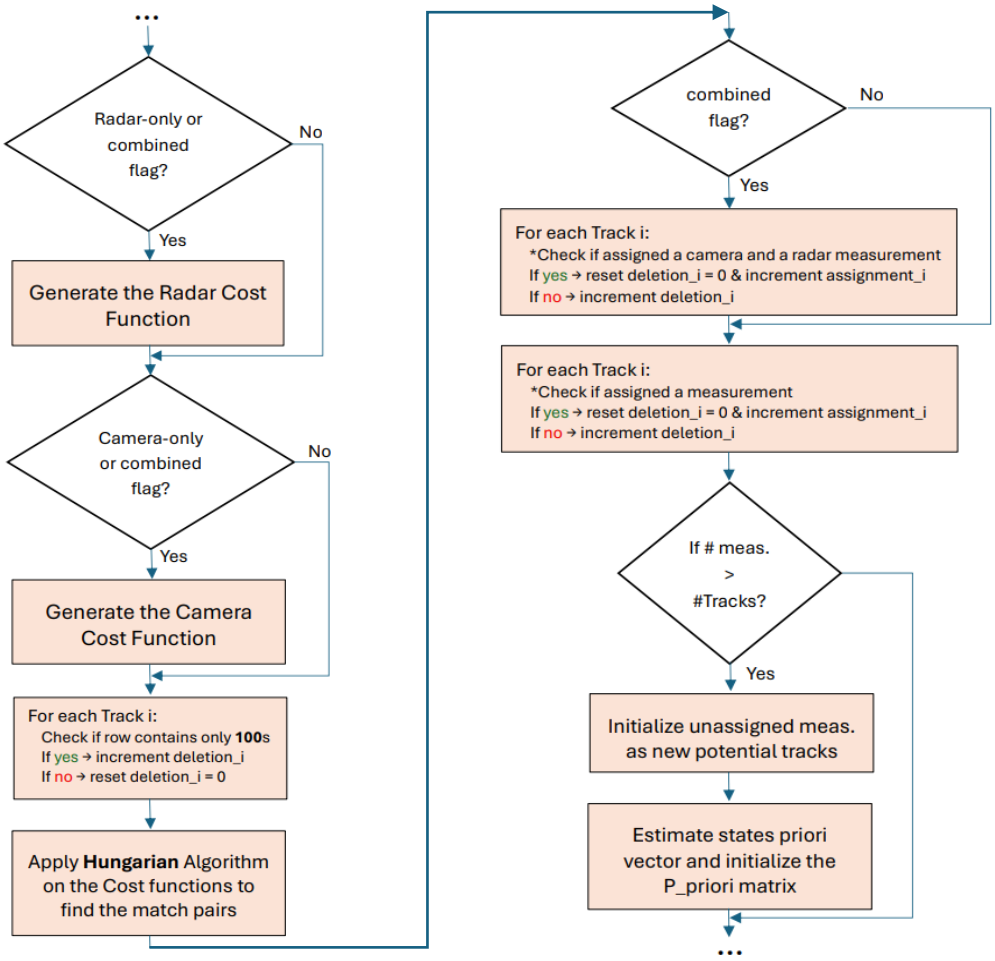


Figure 35. Target tracking logic flowchart.

Random Forest ML Model and Fog Classification

In this section, we discuss the implementation steps and results of design the ML model for fog level estimation.

From the dataset cited here [31], seven visual features listed above were extracted from each image using Python, and each sample was labelled according to its fog condition. The two datasets were combined into one table to be used for model development.

Some features had missing values (like YOLO confidence scores), so we filled those using the median value of the respective feature across the dataset. Features that had almost no variance or were highly correlated were removed to reduce redundancy. The remaining features were scaled to a 0–1 range using MinMax normalization.

The dataset was split: 70% for training, 20% for validation, and 10% for final testing. A Random Forest classifier was trained on the training set, with constraints like max tree depth and minimum samples per leaf to avoid overfitting. Hyperparameters were tuned using RandomizedSearchCV, checking number of trees, max depth, leaf size, max features, bootstrap ratio, and pruning.

The validation set was used to check performance, with metrics including accuracy, precision, recall, F1-score. The learning curve (Figure 36) showed training and validation accuracy were close, so the model wasn't overfitting.

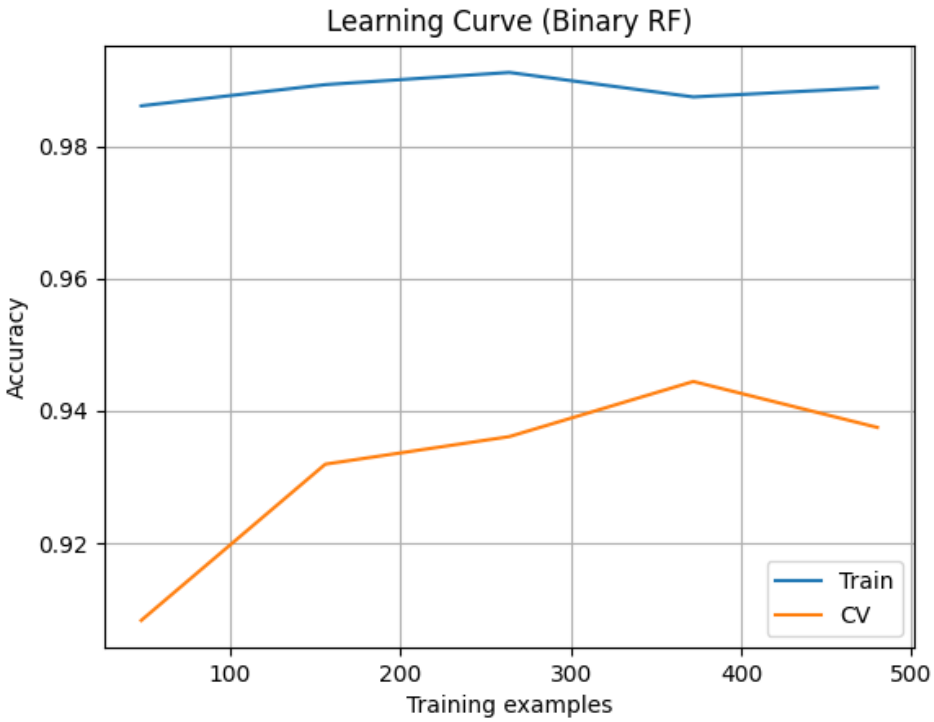


Figure 36. Learning curve of the Random Forest, showing training and validation accuracy versus the epochs.

The final test set confirmed generalization, the performance metrics scores given in Table 4, and the confusion matrix of the validation and testing dataset are shown in Tables 5 and 6, respectively.

The code used is given in Appendix J. The trained Random Forest was saved as a .joblib file so it can be integrated in the full MATLAB fusion code. New images from the testing scene can be pre-processed the same way (features extracted, missing values filled, scaled) before classification. In the next section, we summarize the full Pipeline of Radar-Camera data fusion using the EKF for target tracking.

Table 4. Performance Metrics Results on the Validation and Testing Dataset

	Validation Dataset			Testing Dataset		
	precision	recall	f1-score	precision	recall	f1-score
Dense	0.96	0.91	0.94	0.98	0.96	0.97
No Fog	0.92	0.97	0.94	0.96	0.98	0.97
Accuracy:	0.93888			0.97		

Table 5. Validation Dataset Confusion Matrix

	True No Fog Class	True Dense Fog Class
Predicted No Fog Class	82	8
Predicted Dense Fog Class	3	87

Table 6. Testing Dataset Confusion Matrix

	True No Fog Class	True Dense Fog Class
Predicted No Fog Class	48	2
Predicted Dense Fog Class	1	49

Full Fusion and Tracking Pipeline Summary

This project implements an offline Radar–Camera data fusion system using the EKF for target tracking. Fusion is executed asynchronously, with the pipeline triggered by the arrival of new measurements. This required handling out-of-sequence measurements, as described previously.

The EKF update combines camera and radar measurements (similar to synchronous fusion) in cases where both sensors capture a measurement corresponding to the same frame and arrive within a small time window of 0.05 s. The EKF is integrated within the GNN data association framework used to optimize probabilistic measurement-to-track assignments. Moreover, for camera measurements, the integrated Random Forest is intended to classify frames as dense fog or no fog based on extracted features and to scale the camera noise covariance matrix accordingly.

Track management includes deletion of tracks not updated for 30 consecutive frames and initialization of new tracks for unassigned measurements. Tracks with more than 30 assigned measurements are displayed as established tracks. The developed MATLAB code, from scratch, is provided in Appendix K implements the described system. Note that the plotting module was developed using the assistance of an AI model.

Figure 37 shows how the radar and camera measurements are sorted and organized before being inputted into the data fusion pipeline. Row 1 corresponds to the timestamp at which the measurement frame was captured. Row 2 corresponds to the timestamp at which the Raspberry Pi received the measurement after processing (parsing for radar, YOLO algorithm for camera). Row 3 corresponds to the data sensor flag (1 for radar, 0 for camera). Row 4 corresponds to the frame ID.

Row 5 corresponds to EKF update flag, as mentioned, 1 for camera-only, 2 for radar-only, 3 for combined, and 4 means that the received measurements are static radar measurement, which are skipped. Row 6 is the x measurement (replaced by NaN when the measurement is a static radar measurement). Row 7 is the y measurement. Row 8 is the radar radial velocity measurement. Row10 is the OOSM flag row. Moreover, a sample of the Tracks structure from the MATLAB code, showing how states, covariance errors, a priori and a posteriori estimates, and other fields are organized is shown in Figure 38.

	1	2	3	4	5	6	7	8	9	10
1	7626e+09	1.7626e+09	1	1	4	NaN	3.3113	0	1	0
2	7626e+09	1.7626e+09	1	1	4	NaN	4.9506	0	1	0
3	7626e+09	1.7626e+09	1	1	4	NaN	6.2053	0	1	0
4	7626e+09	1.7626e+09	1	1	4	NaN	9.5955	0	1	0
5	7626e+09	1.7626e+09	1	2	4	NaN	3.3985	0	2	0
6	7626e+09	1.7626e+09	1	2	4	NaN	4.9506	0	2	0
7	7626e+09	1.7626e+09	1	2	4	NaN	6.1788	0	2	0
8	7626e+09	1.7626e+09	1	3	2	0.9374	3.6245	-1.2137	3	0
9	7626e+09	1.7626e+09	1	7	2	0.9374	3.6846	-0.6068	7	0
10	7626e+09	1.7626e+09	0	5	1	1.7379	6.4875	0	5	1
11	7626e+09	1.7626e+09	0	6	1	1.1344	3.9643	0	6	1
12	7626e+09	1.7626e+09	0	9	1	1.6996	6.2503	0	9	0
13	7626e+09	1.7626e+09	0	10	1	1.3504	4.8174	0	10	0
14	7626e+09	1.7626e+09	1	14	2	1.0545	3.6302	-0.6068	14	0
15	7626e+09	1.7626e+09	0	11	1	1.0438	3.5550	0	11	1
16	7626e+09	1.7626e+09	0	12	1	0.8500	2.7571	0	12	1
17	7626e+09	1.7626e+09	0	13	1	0.7628	2.3958	0	13	1
18	7626e+09	1.7626e+09	1	18	4	NaN	3.3113	0	18	0
19	7626e+09	1.7626e+09	1	18	4	NaN	4.9387	0	18	0
20	7626e+09	1.7626e+09	1	18	4	NaN	6.1446	0	18	0

Figure 37. The sorted data table screenshot from the MATLAB fusion pipeline code.

Last minute deployment of YOLO was not successful, instead, YOLO was executed offline on a laptop instead. The code in Appendix L provides the code to implement YOLO on the captured scene; the code uses a csv saving the timestamps of the video camera frames, and saves the corresponding YOLO processing time, detected objects centroids and confidence scores per frame. As mentioned, due to the Homography projective estimation errors resulting in unreasonable data, the obtained YOLO measurements were commented out. Consequently, the Random Forest integration code is not included in the final implementation. In the next section, we present the testing and validation of the results.

Fields	states_priori	states_posteriori	P_priori	P_posteriori	H	R	assigned_meas	status	deletion_increment	assignment_count	y_predicted	sensor	Rsub
1	[-1.3279;3.789...	[-1.2390;3.6510;-0....	4x4 double	4x4 double	3x4 double		[0.2343,3.7566,1.5...	'inactive'	70	101	[-1.3279;3.789...	'radar'	[0.9673,0,0;...
2	[0.6289;1.7896;...	[0.5696;1.6240;0.29...	4x4 double	4x4 double	3x4 double		[-0.0901,1.3398,0....	'inactive'	70	16	[0.6289;1.7896...	'radar'	[0.1231,0,0;...
3	[2.5402;7.6883;...	[2.8714;7.9580;-0.8...	4x4 double	4x4 double	3x4 double		[2.8752,7.9578,-0....	'inactive'	70	1	[2.5402;7.6883...	'radar'	[4.3428,0,0;...
4	[1.0006;1.1939;...	[1.0351;1.3615;-0.3...	4x4 double	4x4 double	3x4 double		[1.0095,1.6451,-2....	'inactive'	70	6	[1.0006;1.1939...	'radar'	[0.1914,0,0;...
5	[-7.8493;10.17...	[-7.4418;10.3174;-2...	4x4 double	4x4 double	3x4 double		[-5.6783,11.8381,...	'inactive'	70	10	[-7.8493;10.17...	'radar'	[9.6091,0,0;...
6	[-5.2716;5.567...	[-4.1001;5.7549;-1....	4x4 double	4x4 double	3x4 double		[-4.8671,3.2442,0....	'inactive'	70	15	[-5.2716;5.567...	'radar'	[0.7362,0,0;...
7	[1.9690;8.0488;...	[0.7157;7.5449;1.79...	4x4 double	4x4 double	3x4 double		[-2.5778,7.1538,0....	'inactive'	70	10	[1.9690;8.0488...	'radar'	[3.5100,0,0;...
8	[-1.0643;3.223...	[-1.0357;3.0612;-0....	4x4 double	4x4 double	3x4 double		[-1.2979,3.2228,1....	'inactive'	70	8	[-1.0643;3.223...	'radar'	[0.7149,0,0;...
9	[1.5770;6.0104;...	[0.3551;5.4864;1.74...	4x4 double	4x4 double	3x4 double		[-0.8563,5.4984,0....	'inactive'	70	5	[1.5770;6.0104...	'radar'	[2.0726,0,0;...
10	[-6.1786;2.433...	[-5.5525;2.8086;-1....	4x4 double	4x4 double	3x4 double		[-5.4891,2.6760,0....	'inactive'	70	2	[-6.1786;2.433...	'radar'	[0.5082,0,0;...
11	[8.2877;8.2302;...	[8.6231;8.3712;-0.8...	4x4 double	4x4 double	3x4 double		[8.0849,8.0457,-0....	'inactive'	70	4	[8.2877;8.2302...	'radar'	[4.4475,0,0;...
12	[1.8065;1.8108;...	[1.8020;2.0177;0.04...	4x4 double	4x4 double	3x4 double		[1.0726,1.6047,-1....	'partially in...	55	13	[1.8065;1.8108...	'radar'	[0.1831,0,0;...
13	[2.6734;4.8767;...	[3.4565;4.9913;-1.1...	4x4 double	4x4 double	3x4 double		[4.3534,4.3323,-0....	'inactive'	70	9	[2.6734;4.8767...	'radar'	[1.2972,0,0;...
14	[-1.3613;2.015...	[-1.4444;2.0588;0.8...	4x4 double	4x4 double	3x4 double		[0.7571,1.7495,-1....	'inactive'	70	23	[-1.3613;2.015...	'radar'	[0.2132,0,0;...
15	[0.5974;1.5527;...	[0.6981;1.7146;-0.5...	4x4 double	4x4 double	3x4 double		[0.4416,1.9010,-1....	'inactive'	70	23	[0.5974;1.5527...	'radar'	[0.2488,0,0;...

Figure 38. Tracks structure screenshot from the MATLAB fusion pipeline code.

Testing and Validation

In this section, we discuss the verification, validation, and evaluation of the system.

Verification

Below is a summary of how the implemented system compares to the requirements and the met deliverables.

Milestones completed successfully:

1. Interfacing between the TI IWR6843 (Radar module) and the Raspberry.
2. The optimization of the *offline* YOLOv11 using the Coral Edge TPU was implemented successfully and documented. Last minute deployment of the YOLOv11 was not successful, and instead YOLO was implemented in a laptop instead.
3. Camera intrinsic calibration and radar–camera extrinsic calibration were both implemented successfully and achieving acceptable reprojection errors.
4. Comparison matrix of fusion algorithms was implemented, concluding the usage of the EKF as the optimal approach.
5. Theoretical foundation on the EKF including the motion model, the measurement equations, as well as the noise covariance matrices estimation was completed.
6. *From scratch* offline MATLAB code for radar–camera data fusion and a data association method for track formation was completed.
7. Design the setup and code for obtaining ground-truth states to benchmark fusion results was completed successfully.

Limitations:

8. The ML classifier model for fog level was trained and integrated into MATLAB, but homography projection errors prevented documenting the results.
9. System performance, including RMSE, could only be tested with radar-only data. Future work should capture the scene with reliable YOLO data to fully test fusion performance and the ML model's effect in improving system robustness.

In the next section, we detail into the procedures of validating the system.

Validation

To validate the system, multiple scenes capturing multithreaded camera and radar data were recorded. Two scenes were relatively simple, and one was more complex. The fusion pipeline's plotting module displayed synchronized animations of ArUco marker measurements, tracker fusion output, YOLO output, RMSE per frame, and radar measurements (X vs Y and range vs radial velocity).

Fine-tuning was required for:

- Deletion increment threshold
- Initialized covariance matrix
- Initialized tracker states
- Standard deviation of the target's acceleration in X and Y
- Radar measurement gating threshold

This fine-tuning was completed for one of the simple scenes, with the resulting plots shown in the linked video: https://youtu.be/H9NKajhr28o. A screenshot from the video is shown in figure 39.

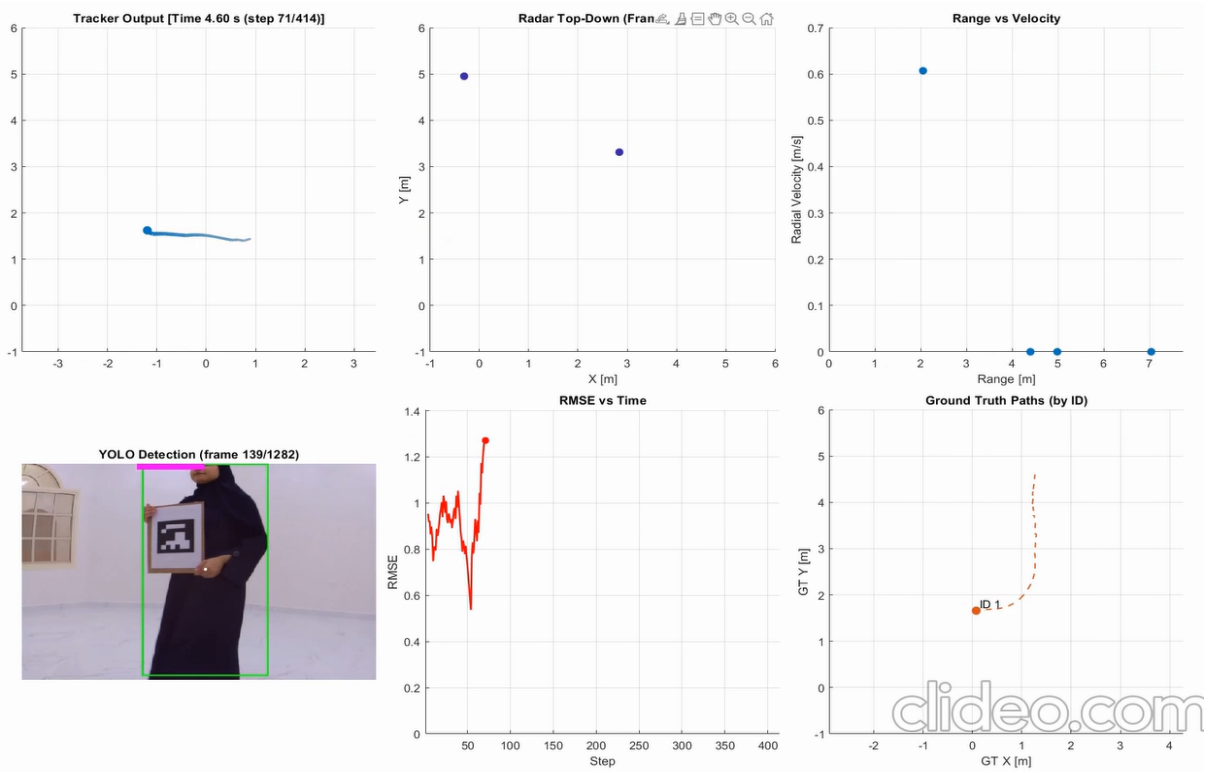


Figure 39. MATLAB fusion pipeline output results.

Evaluation

The RMSE plot for the tested scene is shown in Figure 40. RMSE spikes correspond to frames where ArUco markers were not visible. Some frames also showed the radar detecting only static objects, even when a moving target was present. In other frames, RMSE reached values as low as 0.5 m.

Although multiple datasets were recorded, each required individual parameter fine-tuning. Due to time limitations, tuning was completed for only one dataset.

Regarding data association performance, when a second target enters from the same side the first target exits, the GNN tracker assumes it is the same target and continues along its path for some frames.

Another track is established only after a few frames. Consequently, RMSE towards the end reflects errors between mismatched ground-truth targets and track IDs, as the tracker continued comparing the first established track states with the ground-truth values plotted.

This led to higher RMSE spikes toward the end, where the second target appears. This issue should be resolved once the tracker reliably distinguishes multiple targets, allowing easier ground-truth–track ID matching.

Moreover, after correcting the homography transformation, the fused transformed YOLO detections should support the tracker in producing more accurate measurements. The GNN data association is then expected to be more reliable, distinguishing closer tracks, resulting in more stable tracker output and lower, more consistent RMSE.

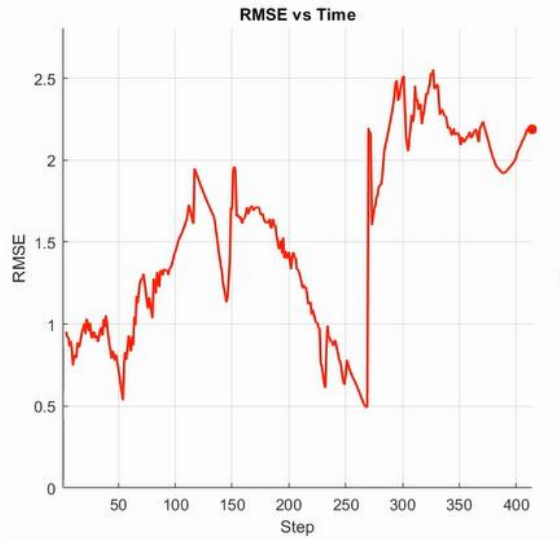


Figure 40. Fusion pipeline RMSE plot output results for the capture scenes of the scenes.

Conclusions

This project set out to design and implement a radar–camera fusion and tracking system using the Extended Kalman Filter and GNN for data association. Although the full real-time fusion system could not be deployed due to YOLO implementation issues on the Raspberry Pi and the incorrect homography setup, the offline fusion framework, calibration, and radar-only tracking pipeline were completed successfully. The results demonstrated that the EKF formulation, the motion and measurement models, the calibration procedures, and the overall tracking structure are valid. The radar-only pipeline achieved RMSE values as low as 0.5 m.

Summary of work done

Listed below is the completed work.

1. Interfaced the TI IWR6843 radar with the Raspberry Pi to obtain the high-level radar data.
2. Implemented multithreading on the Raspberry Pi to record simultaneous radar and camera data.
3. Completed camera intrinsic calibration and radar–camera extrinsic calibration with acceptable reprojection errors.
4. Deployed the optimized YOLOv11 model offline using the Coral Edge TPU on the Raspberry Pi.
5. Developed the full EKF fusion pipeline from scratch and the included the codes for handling out-of-sequence measurements.
6. Implemented the GNN data association method for track formation.
7. Selected BEV as the unified frame and applied the homography transformation matrix, however, inaccuracy in testing setup resulted in unrealistic camera data.
8. Collected several testing scenes using the multithreaded system and generated ground-truth states using ArUco markers.
9. Tuned, tested, and evaluated the system on one dataset, producing radar-only tracking results.

Critical appraisal of work done

Several major components of the system were implemented successfully. The calibration procedures, the EKF framework, and the GNN data association were all completed correctly, and the radar-only tracker produced reasonable accuracy. The primary limitations were: (1) the homography transformation was inaccurate making camera data unusable, and (2) YOLOv11 could not be deployed in real time on the Raspberry Pi, forcing the project to shift to offline processing.

Additionally, system performance was sensitive to parameter tuning, which restricted full validation to a single dataset. Despite these issues, the project demonstrated a solid theoretical understanding and produced a functioning, testable pipeline.

Recommendations for further work

1. Correct the homography setup by adjusting the physical platform, allowing a reasonably accurate projection of YOLO detections into the radar frame.

2. Test and integrate the fog-level ML model, allowing adaptive scaling of the camera measurement noise.

3. Incorporate motion-model multiplexing, where YOLO classification determines whether a constant-acceleration motion model is used or other manoeuvre motion models are used for cars.

4. Extend the system toward navigation or autonomous platform applications, using fused tracks for path planning or collision avoidance.

5. Explore face or human-ID tracking, leveraging the strengths of the camera while using radar when visibility degrades.

6. Improve multi-target handling, possibly moving from GNN to JPDA or MHT.

References

[1]. S. Yao *et al.*, “Radar-Camera Fusion for Object Detection and Semantic Segmentation in Autonomous Driving: A Comprehensive Review,” *arXiv (Cornell University)*, Apr. 2023, doi: https://doi.org/10.1109/tiv.2023.3307157.

[2]. R. Zhang and S. Cao, “Extending Reliability of mmWave Radar Tracking and Detection via Fusion With Camera,” *IEEE Access*, vol. 7, pp. 137065–137079, 2019, doi: https://doi.org/10.1109/access.2019.2942382.

[3]. X. Shuai, Y. Shen, Y. Tang, S. Shi, L. Ji, and G. Xing, “milliEye,” *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, May 2021, doi: https://doi.org/10.1145/3450268.3453532.

[4]. T.-Y. Lim *et al.*, “Radar and Camera Early Fusion for Vehicle Detection in Advanced Driver Assistance Systems,” *Qualcomm AI Research*, 2019.

[5]. A. Sengupta, A. Yoshizawa, and S. Cao, “Automatic Radar-Camera Dataset Generation for Sensor-Fusion Applications,” *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 2875–2882, Apr. 2022, doi: https://doi.org/10.1109/lra.2022.3144524.

[6]. Z. Xu, S. Zhao, and R. Zhang, “An efficient multi-sensor fusion and tracking protocol in a vehicle-road collaborative system,” *IET Communications*, vol. 15, no. 18, pp. 2330–2341, Sep. 2021, doi: https://doi.org/10.1049/cmu2.12273.

[7]. K. Aziz, Eddy De Greef, Maxim Rykunov, A. Bourdoux, and H. Sahli, “Radar-camera Fusion for Road Target Classification,” Sep. 2020, doi: https://doi.org/10.1109/radarconf2043947.2020.9266510.

[8]. sgs-weather-and-environmental-systems, “TI-mmWave-SDK/mmwave_sdk_01_02_00_05/docs/mmwave_sdk_user_guide.pdf at master · sgs-weather-and-environmental-systems/TI-mmWave-SDK,” *GitHub*, 2018. https://github.com/sgs-weather-and-environmental-systems/TI-mmWave-SDK/blob/master/mmwave_sdk_01_02_00_05/docs/mmwave_sdk_user_guide.pdf (accessed Mar. 20, 2025).

[9]. R. Hartley, A. Zisserman, S. Wei, and Quanbing Zhang, *计算机视觉中的多视图几何 = Multiple view geometry in computer vision / Ji suan ji shi jue zhong de duo shi tu ji he = Multiple view geometry in computer vision*. 机械工业出版社, Beijing: Ji Xie Gong Ye Chu Ban She, 2020.

[10]. Mahdi Chamseddine, J. Rambach, and D. Stricker, “CaRaCTO: Robust Camera-Radar Extrinsic Calibration with Triple Constraint Optimization,” pp. 534–545, Jan. 2024, doi: https://doi.org/10.5220/0012369700003654.

[11]. B. Fisher, “Projective Transformations,” *homepages.inf.ed.ac.uk*, Nov. 07, 1997. https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/BEARDSLEY/node3.html

[12]. First Principles of Computer Vision, “Intrinsic and Extrinsic Matrices | Camera Calibration,” *YouTube*, Apr. 18, 2021. https://www.youtube.com/watch?v=2XM2Rb2pfyQ (accessed Apr. 26, 2025).

[13]. “Fisheye Calibration Basics - MATLAB & Simulink,” *www.mathworks.com*. https://www.mathworks.com/help/vision/ug/fisheye-calibration-basics.html (accessed Apr. 26, 2025).

[14]. “What Is Camera Calibration?,” *Mathworks.com*, 2019. https://www.mathworks.com/help/vision/ug/camera-calibration.html (accessed Apr. 26, 2025).

[15]. J. Heikkila and O. Silven, “A four-step camera calibration procedure with implicit image correction,” *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, doi: https://doi.org/10.1109/cvpr.1997.609468.

[16]. D. Simon, *Optimal State Estimation*. John Wiley & Sons, 2006.

[17]. Al-khwarizmi (الخوارزمی), “Sensor Fusion: Extended Kalman Filter - Autonomous Car Motion Estimation,” *YouTube*, Mar. 19, 2023. https://www.youtube.com/watch?v=y2JswR4g9lI (accessed Aug. 26, 2025).

[18]. Steve Brunton, “The Kalman Filter [Control Bootcamp],” *YouTube*, Feb. 06, 2017. https://www.youtube.com/watch?v=s_9InuQAx-g&list=PLMrJAkhIeNNR20Mz-VpzgfQs5zrYi085m&index=18 (accessed Aug. 26, 2025).

[19]. “Lecture 24: Observability and Constructibility 7 Observability and Constructibility.” Accessed: Aug. 26, 2025. [Online]. Available: https://cim.mcgill.ca/~boulet/304-501A/L24.pdf

[20]. Steve Brunton, “Control Bootcamp: Observability,” *YouTube*, Feb. 06, 2017. https://www.youtube.com/watch?v=iRZmJBcg1ZA&list=PLMrJAkhIeNNR20Mz-VpzgfQs5zrYi085m&index=16 (accessed Aug. 26, 2025).

[21].]. B. Douglas, “#2: The Kalman Filter,” *Engineering Media*, Feb. 25, 2021. https://engineeringmedia.com/controlblog/the-kalman-filter

[22]. K. Zhang, Z. Wang, L. Guo, Y. Peng, and Z. Zheng, “An Asynchronous Data Fusion Algorithm for Target Detection Based on Multi-Sensor Networks,” *IEEE Access*, vol. 8, pp. 59511–59523, 2020, doi: https://doi.org/10.1109/access.2020.2982682.

[23]. Kalman, “Signal Processing Stack Exchange,” *Signal Processing Stack Exchange*, Sep. 04, 2019. https://dsp.stackexchange.com/questions/60511/kalman-filter-how-to-combine-data-from-sensors-with-different-measurement-rate/60513#60513 (accessed Aug. 26, 2025).

[24]. X. Li and V. P. Jilkov, “Survey of maneuvering target tracking: dynamic models,” Jul. 2000, doi: https://doi.org/10.1117/12.391979.

[25]. D. Waard, “UvA-DARE (Digital Academic Repository) A new approach to distributed data fusion,” 2008. Accessed: Aug. 26, 2025. [Online]. Available: https://pure.uva.nl/ws/files/4318786/59162_07.pdf

[26]. B. Collins, “Introduction to Data Association,” 2012. https://www.cse.psu.edu/~rtc12/CSE598C/datassocPart1.pdf (accessed Aug. 26, 2025).

[27]. R. Castelli, P. Frolkovic, C. Reinhardt, C. C. Stolk, J. Tomczyk, and A. Vromans, “Fog detection from camera images,” pp. 25–43, Jan. 2016.

[28]. Z. Li, S. Zhang, Z. Fu, F. Meng, and L. Zhang, “Confidence-Feature Fusion: A Novel Method for Fog Density Estimation in Object Detection Systems,” *Electronics*, vol. 14, no. 2, pp. 219–219, Jan. 2025, doi: https://doi.org/10.3390/electronics14020219.

[29]. Y. Guan, L. Yu, S. Hao, L. Li, X. Zhang, and M. Hao, “Slope Failure and Landslide Detection in Huangdao District of Qingdao City Based on an Improved Faster R-CNN Model,” *GeoHazards*, vol. 4, no. 3, pp. 302–315, Aug. 2023, doi: https://doi.org/10.3390/geohazards4030017.

[30]. Z. Li, S. Zhang, Z. Fu, F. Meng, and L. Zhang, “Confidence-Feature Fusion: A Novel Method for Fog Density Estimation in Object Detection Systems,” *Electronics*, vol. 14, no. 2, pp. 219–219, Jan. 2025, doi: https://doi.org/10.3390/electronics14020219.

[31]. Yessica, “Foggy Cityscapes Images,” *Kaggle.com*, 2025. https://www.kaggle.com/datasets/yessicatuteja/foggy-cityscapes-image-dataset (accessed Nov. 23, 2025).

[32]. P. Roßbach, “Neural Networks vs. Random Forests – Does it always have to be Deep Learning?,” *frankfurt*. https://blog.frankfurt-school.de/wp-content/uploads/2018/10/Neural-Networks-vs-Random-Forests.pdf (accessed Nov. 23, 2025).

[33]. *LiDAR: What Is It and How Does It Work? | YellowScan* (2024) *YellowScan*. Available at: https://www.yellowscan.com/knowledge/how-does-lidar-work/? (Accessed: 29 August 2025).

[34]. *Real time operation with cm level accuracy* (2025) *Sbg-systems.com*. Available at: https://support.sbg-systems.com/sc/kb/latest/inertial-sensors-operation/real-time-operation-with-cm-level-accuracy?utm (Accessed: 29 August 2025).

[35]. Tocci, T., Capponi, L. and Rossi, G. (2021) ‘ArUco marker-based displacement measurement technique: uncertainty analysis’, *Engineering Research Express*, 3(3), p. 035032. Available at: https://doi.org/10.1088/2631-8695/ac1fc7.

[36]. “Detecting ArUco Markers with OpenCV and Python,” GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/computer-vision/detecting-aruco-markers-with-opencv-and-python-1/

[37]. S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," Pattern Recognition, vol. 47, no. 6, pp. 2280–2292, Jun. 2014.

[38]. A. Poroykov, P. Kalugin, S. Shitov, and I. Lapitskaya, "Modeling ArUco Markers Images for Accuracy Analysis of Their 3D Pose Estimation," in Proceedings of the CEUR Workshop, vol. 2744, 2020. [Online]. Available: https://ceur-ws.org/Vol-2744/short14.pdf

[39]. "Camera Calibration," MathWorks Documentation. [Online]. Available: https://www.mathworks.com/help/vision/ug/camera-calibration.html

[40]. "Detection of ArUco Markers," OpenCV Documentation. [Online]. Available: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html

[41]. J. WEN, J. YANG, M. FU, J. ZHANG, and W. YANG, "A Calibration Algorithm for Maximum Likelihood Estimation of Extrinsic Parameters of a Camera and a 3D Laser Radar," *ROBOT*, vol. 33, no. 1, pp. 102–106, Aug. 2011, doi: https://doi.org/10.3724/sp.j.1218.2011.00102.

[42]. E. Wise, Juraj Peršić, C. Grebe, I. Petrović, and J. Kelly, "A Continuous-Time Approach for 3D Radar-to-Camera Extrinsic Calibration," *arXiv (Cornell University)*, pp. 13164–13170, May 2021, doi: https://doi.org/10.1109/icra48506.2021.9561938.

[43]. C. Song, G. Son, H. Kim, D. Gu, J.-H. Lee, and Y. Kim, "A Novel Method of Spatial Calibration for Camera and 2D Radar Based on Registration," Jul. 2017, doi: https://doi.org/10.1109/iiai-aai.2017.62.

[44]. D. Kim and S. Kim, "Extrinsic parameter calibration of 2D radar-camera using point matching and generative optimization," pp. 99–103, Oct. 2019, doi: https://doi.org/10.23919/iccas47443.2019.8971568.

[45]. Adynathos, "EPnP: An Accurate O(n) Solution to the PnP Problem," *Github.com*, 2025. https://github.com/cvlab-epfl/EPnP

[46]. "mmWave Demo Visualizer User's Guide mmWave Demo Visualizer," 2017. Accessed: Mar. 20, 2025. [Online]. Available: https://www.ti.com/lit/ug/swru529c/swru529c.pdf?ts=1742455933180&ref_url=https%253A%252F%252Fwww.google.com%252F

[47]. M. F. Mohamad Ali, M. S. Abustan, S. H. Abu Talib, I. Abustan, N. Abd Rahman, and H. Gotoh, "A Case Study on the Walking Speed of Pedestrian at the Bus Terminal Area," *E3S Web of Conferences*, vol. 34, p. 01023, 2018, doi: https://doi.org/10.1051/e3sconf/20183401023.

[48]. V. Lindberg, "Uncertainties and Error Propagation," *www.geol.lsu.edu*, Jul. 01, 2000. http://www.geol.lsu.edu/jlorenzo/geophysics/uncertainties/Uncertaintiespart2.html

[49]. K. Aziz, Eddy De Greef, Maxim Rykunov, A. Bourdoux, and H. Sahli, “Radar-camera Fusion for Road Target Classification,” Sep. 2020, doi: https://doi.org/10.1109/radarconf2043947.2020.9266510.

[50]. Pavlina Konstantinova, A. Udvarev, and Tzvetan Semerdjiev, “A study of a target tracking algorithm using global nearest neighbor approach,” Jan. 2003, doi: https://doi.org/10.1145/973620.973668.

[51]. Wikipedia Contributors, “Hungarian algorithm,” *Wikipedia*, Aug. 27, 2019. https://en.wikipedia.org/wiki/Hungarian_algorithm

Appendices

Appendix A – Supplementary Sections on the KF theory

Least Squares and Weighted Least Squares Estimation Algorithms

There is a vector $\boldsymbol{x}$ is an n – dimensional column vector that we attempt to estimate. We cannot directly measure $\boldsymbol{x}$ but we can measure $\boldsymbol{y}$, yet since no measurement device achieved infinite precision, there is a noise v_k associated with every measurement y_k. A matrix $\boldsymbol{H}$ describes the mapping from the unknow vector $\boldsymbol{x}$ to vector $\boldsymbol{y}$.

$\boldsymbol{x} \ni \mathbb{R}^{\boldsymbol{n\times 1}}, \boldsymbol{y} \ni \mathbb{R}^{\boldsymbol{k\times 1}}, \boldsymbol{H} \ni \mathbb{R}^{\boldsymbol{k\times n}}, \boldsymbol{v} \ni \mathbb{R}^{\boldsymbol{k\times 1}}$. When k > n, our system becomes over constrained, or overdetermined, we have more measurements than we have variables [1, 2]. The maximum dimension that can be spanned by H is min (k, n) = n. We, therefore, usually cannot express $\boldsymbol{y}$ to be exactly equal to $\boldsymbol{Hx}$ due to the presence of noise, we always tend to obtain as many measurements (k) to improve our estimate. H can only span a subspace of the space $\mathbb{R}^{\boldsymbol{k\times 1}}$. We express y as shown below. We obtain the best approximate for x, when the error is minimized.

$$\boldsymbol{y} = \boldsymbol{H\hat{x}} + \boldsymbol{e}$$

Envision the geometric vector $\boldsymbol{y}$, represented as a sum its projection into the column space of $\boldsymbol{H}$, which is the first term $\boldsymbol{H\hat{x}}$ [2] (the coordinates of $\hat{\boldsymbol{x}}$ forms a linear combination of the columns of $\boldsymbol{H}$, and hence $\boldsymbol{H\hat{x}}$ is an element of the Col(A) space). The vector $\boldsymbol{e}$ joins the end of $\boldsymbol{H\hat{x}}$ vector to the $\boldsymbol{y}$ vector [2].

Moreover, we can geometrically envision that the optimal approximation of $\boldsymbol{x}$ is one such that $\boldsymbol{H\hat{x}}$ is the orthogonal projection of $\boldsymbol{y}$ into Col(H), and $\boldsymbol{e}$ being orthogonal to space spanned by the Columns of H [2, 3]. This is also illustrated in Figures 1 and 2. All other vectors forms of e for different $\boldsymbol{H\hat{x}}$ would yield a greater distance e [3].

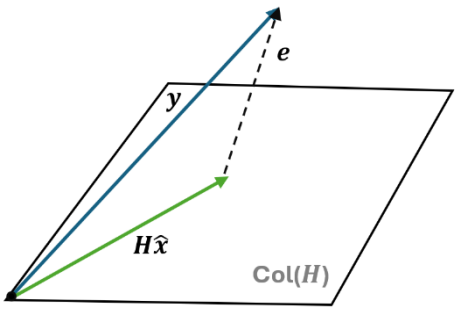


Figure 1. Projection of the y matrix on the column space of H, when e is not orthogonal

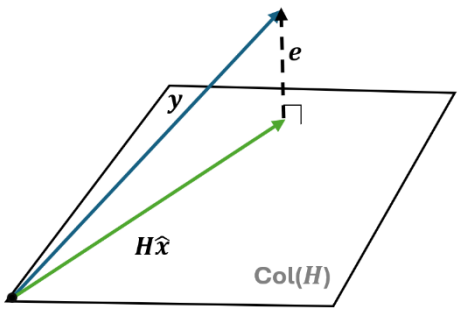


Figure 2. Projection of the y matrix on the column space of H, when e is not orthogonal

We can formulate the problem mathematically to obtain the expression for $\hat{\boldsymbol{x}}$. We formulate the problem of minimize the error distance $\boldsymbol{e}$ as follows. We define the cost function (J) of which we want

to minimize to the be the sum of the squares of the difference between the measured values y and the approximation $H\hat{x}$, **J** is given as shown below [1]. We choose minimizing the sum of the squares instead of for example minimizing the absolute values, since it is differentiable while the latter is not.

$$\boldsymbol{J} = (y_1 - H\hat{x}_1)^2 + (y_2 - H\hat{x}_2)^2 + (y_3 - H\hat{x}_3)^2 + \ldots + (y_k - H\hat{x}_k)^2$$

$$\boldsymbol{J} = || \boldsymbol{y} - \boldsymbol{H\hat{x}} ||^2 = (\boldsymbol{y} - H\hat{\boldsymbol{x}})^T \cdot (\boldsymbol{y} - H\hat{\boldsymbol{x}})$$

$$\boldsymbol{J} = \boldsymbol{y^T y} - \hat{\boldsymbol{x}}^T \boldsymbol{H^T y} - \boldsymbol{y^T H\hat{x}} + \hat{\boldsymbol{x}} \boldsymbol{H^T H\hat{x}}$$

Next, to minimize J as a function $\hat{\boldsymbol{x}}$ we find its partial derivative and set it to zero (refer to matrix calculus formulas) [1]. For $\hat{\boldsymbol{x}}$ to exist, $\boldsymbol{H}$ must be full rank, the measurement needs to be linearly independent [1].

$$\frac{\boldsymbol{\partial J}}{\boldsymbol{\partial \hat{x}}} = -\boldsymbol{2y^T H} + \boldsymbol{2\hat{x}^T H^T H} = \boldsymbol{0}$$

$$\left(\hat{\boldsymbol{x}}^T \boldsymbol{H^T H}\right)^T = \left(\boldsymbol{y^T H}\right)^T$$

$$\boldsymbol{H^T H\hat{x}} = \boldsymbol{H^T y}$$

$$\hat{\boldsymbol{x}} = \left(\boldsymbol{H^T H}\right)^{-1} \boldsymbol{H^T y}$$

In Weighted Least Squares, the variance of the noise ($\boldsymbol{v}$) associated with each measurement is assumed to be known [1]. The noise is also assumed to be zero-mean and independent [1]. The autocovariance matrix of the noise is given by,

$$R = E((\boldsymbol{v} - \bar{v})(\boldsymbol{v} - \bar{v})^T) = E(vv^T)$$

$$R = \begin{bmatrix} E(v_1^2) & \ldots & E(v_1 v_k) \\ \vdots & \ldots & \vdots \\ E(v_k v_1) & \ldots & E(v_k^2) \end{bmatrix} = \begin{bmatrix} E(v_1^2) & \ldots & E(v_1)E(v_k) \\ \vdots & \ldots & \vdots \\ E(v_k)E(v_1) & \ldots & E(v_k^2) \end{bmatrix}$$

Therefore, R is simplified as,

$$R = \begin{bmatrix} \sigma_1^2 & \ldots & 0 \\ \vdots & \ldots & \vdots \\ 0 & \ldots & \sigma_k^2 \end{bmatrix}$$

The lower the variance, the more compact (less spread) the distribution of out measurement and the more certain we are about the accuracy of the measurement, and vice versa. The cost function must be penalized when the error is greater for more accurate measurements than for noisier measurements. We incorporate the measurement noise into the cost function J as shown below [1].

$$\boldsymbol{J} = \frac{(y_1 - H\hat{x}_1)^2}{\sigma_1^2} + \frac{(y_2 - H\hat{x}_2)^2}{\sigma_2^2} + \frac{(y_3 - H\hat{x}_3)^2}{\sigma_3^2} + \ldots + \frac{(y_k - H\hat{x}_k)^2}{\sigma_k^2}$$

The inverse of a diagonal matrix is matrix with the reciprocal of the elements along the diagonal, hence:

$$\boldsymbol{J} = (\boldsymbol{y} - \boldsymbol{H}\hat{\boldsymbol{x}})^{\boldsymbol{T}}\boldsymbol{R}^{-\boldsymbol{1}}(\boldsymbol{y} - \boldsymbol{H}\hat{\boldsymbol{x}})$$

Performing the same steps for the LS, we obtain the expression for $\hat{\boldsymbol{x}}$ as [1]:

$$\hat{\boldsymbol{x}} = \left(\boldsymbol{H}^{\boldsymbol{T}}\boldsymbol{R}^{-\boldsymbol{1}}\boldsymbol{H}\right)^{-\boldsymbol{1}}\boldsymbol{H}^{\boldsymbol{T}}\boldsymbol{R}^{-\boldsymbol{1}}\boldsymbol{y}$$

For the expression of $\hat{\boldsymbol{x}}$ to exist, the noise autocovariance matrix must be non-singular, all its eigenvalues must non-zero, meaning all measurements must have some non-zero noise [1], and as with LS, $\boldsymbol{H}$ must be full rank. In the next section, we discuss the recursive least squares, a computationally efficient method of determining the optimal estimate of $\boldsymbol{x}$.

Recursive Least Squares Estimation Algorithm

Assume we are solving the problem of obtaining the parameters of a discrete time FIR (finite impulse response) filter, which could have M = 64, 128, 256, or 1024 parameters. The parameters represent the $\boldsymbol{x}$ vector we attempt to approximate; the $\boldsymbol{y}$ vectors represent the data obtained of the filter response to a know input.

The equation below expresses the response of a filter as function of discrete finite input values and finite discrete response values using convolution. Using LS to approximate the w[k] parameter would involve finding the inverse of large matrices, and as the number of measurements increases, the computation costs increase significantly. Therefore, it is an inefficient algorithm.

$$y[n] = \sum_{k=0}^{M-1} w[k]\, x[n-k]$$

The RLS algorithm to estimate the x vector using only the current measurement and the previous iteration estimation [1], hence recursively improving the estimation of x as more measurements arrive without require as much computational cost.

The RLS estimator is written in the form shown below:

$$y_k = \boldsymbol{H_k x} + v_k$$

$$\hat{\boldsymbol{x}}_{\boldsymbol{k}} = \hat{\boldsymbol{x}}_{\boldsymbol{k-1}} + K_k(\boldsymbol{y_k} - \boldsymbol{H_k}\hat{\boldsymbol{x}}_{\boldsymbol{k-1}})$$

$\hat{\boldsymbol{x}}_{\boldsymbol{k-1}}$ is the previous estimate, $y_k - \boldsymbol{H_k}\hat{\boldsymbol{x}}_{\boldsymbol{k-1}}$ is the correction term, and $K_k$is the estimation gain matrix [1]. The gain matrix, K_k determines how much trust, or weight is put on the new measurement y_k when updating the estimate of $\boldsymbol{x}$ for t = k. Intuitively, therefore, the gain matrix is expected to be a an inverse function of the current measurement noise, the greater the noise present in the

measurement, the smaller the gain, the smaller the effect the new measurement has on updating our estimate of $\boldsymbol{x}$.

Moreover, we track how close our estimate is to the true value of $\boldsymbol{x}$ using the autocovariance of the RV $\boldsymbol{x}$ that we attempt to estimate. In order to ensure that the covariance decays as more measurements are obtained, we determine the optimal value of $\boldsymbol{K_k}$ which shrinks it at every iteration [1]. We recursively update $\boldsymbol{K_k}$ based on previous update covariance and current covariance. The greater the gain, $\boldsymbol{K_k}$, the more reliable the measurement is, the more correction applied, and the faster our convergence to the smallest possible uncertainty of the $\hat{\boldsymbol{x}}_{\boldsymbol{k}}$ RV estimate.

Source [1] shows the detailed mathematical expression for $\boldsymbol{P_k}$ equation given below (pg. 84 - 85), simplifying the expression $E[(\boldsymbol{x} - \hat{\boldsymbol{x}}_{\boldsymbol{k}})^T(\boldsymbol{x} - \hat{\boldsymbol{x}}_{\boldsymbol{k}})]$ as:

$$\boldsymbol{P_k} = (\boldsymbol{I} - \boldsymbol{K_k H_k})\boldsymbol{P_{k-1}}(\boldsymbol{I} - \boldsymbol{K_k H_k})^T + \boldsymbol{K_k R_k K_k^T}$$

This equation assumes that **(1)** the noise v_k is zero – mean. **(2)** The noise vector and the error of the estimation at $t_{k-1}$are independent, thus the expect value of there multiplication is zero.

As stated above, the equation for $\boldsymbol{K_k}$ is determined such that the variances of the n- elements RV $\boldsymbol{x}$ is minimized, which is the also the trace of $\boldsymbol{P_k}$ matrix [1]. We define **J**, the cost function as follows,

$$J_k = E[(x_1 - \hat{x}_1)^2] + \cdots + E[(x_k - \hat{x}_k)^2] = E[(\boldsymbol{x} - \hat{x}_k)^T(\boldsymbol{x} - \hat{x}_k)] = E\big[Tr[(\boldsymbol{x} - \hat{x}_k)(\boldsymbol{x} - \hat{x}_k)^T]\big]$$

$$\boldsymbol{J_k} = Tr\,\boldsymbol{P_k}$$

$\boldsymbol{K_k}$ is found by finding the partial derivative of $\boldsymbol{J_k}$ with respect $\boldsymbol{K_k}$ and set to zero. Solving for $\boldsymbol{K_k}$ we obtain the following equation,

$$\boldsymbol{K_k} = \boldsymbol{P_{k-1}H_k^T} \cdot (\boldsymbol{H_k P_{k-1} H_k^T} + \boldsymbol{R_k})^{-1}$$

Next, we explore the bias of our RLS estimator, which is through finding the mean of the estimation error at time *k*, the simplified equation from [2] is given below,

$$E(\boldsymbol{x} - \hat{x}_k) = (\boldsymbol{I} - \boldsymbol{K_k H_k})E(\boldsymbol{x} - \hat{x}_{k-1}) - \boldsymbol{K_k}E(v_k)$$

The RLS estimator has zero bias for the same conditions made for the equation $\boldsymbol{P_k}$, which means that the RLS estimation of x should on average be equal to the true value of x [1]. This is satisfied for all values of the gain Kk, hence, with the gain optimized to minimize the cost function, the estimation error should also be regularly close to zero [1]. Moreover, the RLS algorithm assumes that the noise at time k is uncorrelated to the noise at time i for all integers expect when i =k, that is, $E(\boldsymbol{v_k v_i}) = R_k\delta_{k-i}$ [1] (R_k being the autocorrelation for the noise signal), this means the noise is white and the

augmented R matrix for all measurements is diagonal. In the next section, we explore generalizing from the RLS estimator to the Kalman filter under dynamic systems.

RLS Estimation to Kalman Filter

As mentioned in the introduction, the RLS algorithm estimates a constant or a slowly varying random variable (RV), while the Kalman filter assumes the system is dynamic, meaning the variable we attempt to estimate is governed by dynamics we can model and are somewhat certain about. For that reason, we refer to the variable x in the Kalman filter as a stochastic process, a random variable whose characteristics or distribution vary with time [1]. The random variable state may be multivariate, meaning it involves multiple variables of different characteristics considered jointly.

In RLS, the algorithm has one stage comprising three steps: (0) initialize the estimate for the states x and the estimation error covariance matrix P0; (1) calculate the gain matrix Kk; (2) estimate x using Kk, the correction term, and the previous estimate; (3) calculate the covariance matrix of the estimate at t=k. In the second iteration, the gain matrix is updated, taking into account the previous iteration's covariance matrix and the current measurement noise.

In the Kalman filter, however, we have two stages: the prediction step and the measurement (or time-update) step. Since the state we are tracking is governed by a known, modelled dynamic, we cannot simply refine the estimate based on measurements and previous values. The state is not constant, its previous values do not provide sufficient information about where it could be in the future.

This highlights the importance of the prediction step: we use the model we have of the state, which may include some modelled noise denoted as Q, and refine this estimate using the measurement. How much we allow the new measurements to contribute to estimating the state at t=k depends on how noisy the measurement is relative to the modelling or process noise Q. We will see in detail how the prediction step estimates the state prior to measurement by propagating forward how the mean of the state should change with time [1], which is what we are attempting to track.

Over time, the mean of the state estimation error is zero, meaning our estimator makes the Kalman filter, on average, unbiased, neither consistently above nor below the true expected value of x. Of course, the same conditions that ensure unbiasedness in RLS also apply to the Kalman filter: the measurement noise vk must be zero-mean and uncorrelated with the signal.

The gain matrix Kk in the Kalman filter is updated based on the dynamic model of the system, the measurement noise, the modelling noise, and the previous covariance, thus allowing it to adaptively refine its gain matrix in a way that decreases the current covariance. For the state estimate after measurement, Kk balances the weight placed on the prediction and the measurement. We explore the details of the derivation and further insights in the upcoming sections.

References

[1]. D. Simon, *Optimal state estimation : Kalman, H [infinity symbol] and nonlinear approaches*. Hoboken, N.J.: Wiley-Interscience, 2006.

[2]. T. Michael and Heath, “Chapter 3 -Linear Least Squares.” Accessed: Aug. 26, 2025. [Online]. Available: https://heath.cs.illinois.edu/scicomp/notes/chap03_8up.pdf

[3]. B. P. Lathi and Z. Ding, *Modern digital and analog communication systems*. New York: Oxford University Press, 2019.

Appendix B – MATLAB Code for Generating ArUco Markers

```
# Generated using the assistance of an AI model

import cv2
import numpy as np
import os
print("Current working directory:", os.getcwd())

aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_250)
marker_ids = [0, 1, 2, 3]
dpi = 300
marker_size_mm = 50
padding_mm = 20
output_dir = "aruco_markers_A4"

mm_to_inches = 1 / 25.4
marker_size_px = int(marker_size_mm * dpi * mm_to_inches)
padding_px = int(padding_mm * dpi * mm_to_inches)
a4_width_px = int(210 * dpi * mm_to_inches)
a4_height_px = int(297 * dpi * mm_to_inches)
rows, cols = 2, 2

os.makedirs(output_dir, exist_ok=True)

markers = []
for marker_id in marker_ids:
    marker = cv2.aruco.generateImageMarker(aruco_dict, marker_id, marker_size_px)
    path = os.path.join(output_dir, f"marker_{marker_id}.png")
    cv2.imwrite(path, marker)
    markers.append(marker)
    print(f"Saved {path}")

page = np.ones((a4_height_px, a4_width_px), dtype=np.uint8) * 255
grid_width = cols * marker_size_px + (cols - 1) * padding_px
grid_height = rows * marker_size_px + (rows - 1) * padding_px
start_x = (a4_width_px - grid_width) // 2
start_y = (a4_height_px - grid_height) // 2

index = 0
for r in range(rows):
    for c in range(cols):
        if index >= len(markers):
            break
        x = start_x + c * (marker_size_px + padding_px)
        y = start_y + r * (marker_size_px + padding_px)
        page[y:y+marker_size_px, x:x+marker_size_px] = markers[index]
        cv2.putText(page, f"ID: {marker_ids[index]}", (x, y + marker_size_px + int(0.04 *
a4_height_px)),
                    cv2.FONT_HERSHEY_SIMPLEX, 2, (0,), 4, cv2.LINE_AA)
        index += 1

output_path = os.path.join(output_dir, "aruco_markers_A4.png")
cv2.imwrite(output_path, page)
```

Appendix C – MATLAB Code for Extracting Ground Truth Using ArUco Markers

```
# Generated using the assistance of an AI model

import argparse
import time
from pathlib import Path
import sys

import cv2
import numpy as np
import pandas as pd

# ================ USER CONFIG ================
# Replace these with the camera matrix and distortion you gave in the message
CAMERA_MATRIX = np.array(
    [
        [4053.60854628705, 0.0, 1979.89094280910],
        [0.0, 4046.29446186445, 1577.19501797494],
        [0.0, 0.0, 1.0],
    ],
    dtype=np.float64,
)

# You gave k = [-0.514753000110433 0.269120171613837 0] (looks like two radial terms)
# We'll keep the common 5-term form but only fill what you provided.
DIST_COEFFS = np.array([-0.514753000110433, 0.269120171613837, 0.0, 0.0, 0.0],
dtype=np.float64)

MARKER_SIZE_M = 0.1  # meters (set correctly for your physical markers)
INPUT_CSV_CANDIDATES = ["camera.csv", "camera.csv.csv", "camera_log.csv"]
OUTPUT_CSV_NAME = "camera_with_markers.csv"
SAVE_DEBUG_FRAMES = 1  # save annotated debug JPGs
UPSCALE_FACTOR = 2  # detection upscaling; 1 = no upscale
PRINT_EVERY = 50
# ============================================
# Try a few dictionaries
TRY_DICTS = [
    ("DICT_ARUCO_ORIGINAL", cv2.aruco.DICT_ARUCO_ORIGINAL),
    ("DICT_4X4_50", cv2.aruco.DICT_4X4_50),
    ("DICT_5X5_100", cv2.aruco.DICT_5X5_100),
    ("DICT_6X6_250", cv2.aruco.DICT_6X6_250),
    # APRILTAG dictionary may or may not be present depending on OpenCV build:
    ("DICT_APRILTAG_36h11", getattr(cv2.aruco, "DICT_APRILTAG_36h11", None)),
]
def get_script_dir() -> Path:
    """Resolve script directory; fallback to cwd when __file__ is missing."""
    try:
        return Path(__file__).resolve().parent
    except Exception:
        return Path.cwd()
```

```
SCRIPT_DIR = get_script_dir()
OUTPUT_CSV = SCRIPT_DIR / OUTPUT_CSV_NAME


def make_params_tuned():
    """Create a DetectorParameters object with more permissive settings for small tags."""
    try:
        params = cv2.aruco.DetectorParameters()
    except Exception:
        try:
            params = cv2.aruco.DetectorParameters_create()
        except Exception:
            params = cv2.aruco.DetectorParameters()

    # Tune a few parameters if present
    if hasattr(params, "adaptiveThreshWinSizeMin"):
        params.adaptiveThreshWinSizeMin = 3
    if hasattr(params, "adaptiveThreshWinSizeMax"):
        params.adaptiveThreshWinSizeMax = 53
    if hasattr(params, "adaptiveThreshWinSizeStep"):
        params.adaptiveThreshWinSizeStep = 4
    if hasattr(params, "adaptiveThreshConstant"):
        params.adaptiveThreshConstant = 7

    if hasattr(params, "minMarkerPerimeterRate"):
        params.minMarkerPerimeterRate = 0.02
    if hasattr(params, "maxMarkerPerimeterRate"):
        params.maxMarkerPerimeterRate = 4.0

    # Corner refinement if available
    if hasattr(cv2.aruco, "CORNER_REFINE_SUBPIX") and hasattr(params,
"cornerRefinementMethod"):
        params.cornerRefinementMethod = cv2.aruco.CORNER_REFINE_SUBPIX
    if hasattr(params, "cornerRefinementWinSize"):
        params.cornerRefinementWinSize = 5
    if hasattr(params, "cornerRefinementMaxIterations"):
        params.cornerRefinementMaxIterations = 50
    if hasattr(params, "cornerRefinementMinAccuracy"):
        params.cornerRefinementMinAccuracy = 0.01
    if hasattr(params, "minDistanceToBorder"):
        params.minDistanceToBorder = 1

    return params
def build_detector(dict_id):
    """Create dictionary + detector object (if available on this OpenCV build)."""
    dictionary = cv2.aruco.getPredefinedDictionary(dict_id)
    params = make_params_tuned()
    try:
        detector = cv2.aruco.ArucoDetector(dictionary, params)
        return dictionary, params, detector, True
    except Exception:
        # older API fallback: detector None (we'll call detectMarkers directly)
        return dictionary, params, None, False
def detect_once_on_frame(frame_bgr, dictionary, params, detector, new_api):
    """
```

```
    Detect markers on the provided frame (frame_bgr).
    Returns: corners (list of Nx4x2 arrays), ids (Nx1), where corners are in pixel coords of frame_bgr.
    """
    gray = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2GRAY)
    if new_api and detector is not None:
        corners, ids, _ = detector.detectMarkers(gray)
    else:
        corners, ids, _ = cv2.aruco.detectMarkers(gray, dictionary, parameters=params)
    return corners, ids


def estimate_pose(corners, camera_mat_for_pose, dist_coeffs):
    """Robust pose estimator that works even if estimatePoseSingleMarkers is missing."""
    if corners is None or len(corners) == 0:
        return None, None

    # 1) Try the direct function first (new API)
    if hasattr(cv2.aruco, "estimatePoseSingleMarkers"):
        try:
            rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(
                corners, MARKER_SIZE_M, camera_mat_for_pose, dist_coeffs
            )
            return rvecs, tvecs
        except Exception:
            pass  # fallback below

    # 2) Manual fallback using solvePnP
    rvecs = []
    tvecs = []

    # Marker model points (square in 3D)
    half = MARKER_SIZE_M / 2.0
    objp = np.array([
        [-half,  half, 0],
        [ half,  half, 0],
        [ half, -half, 0],
        [-half, -half, 0],
    ], dtype=np.float64)

    for c in corners:
        c = c.reshape((4, 2)).astype(np.float64)
        ok, rvec, tvec = cv2.solvePnP(
            objp, c, camera_mat_for_pose, dist_coeffs, flags=cv2.SOLVEPNP_ITERATIVE
        )
        if not ok:
            return None, None
        rvecs.append(rvec)
        tvecs.append(tvec)

    return np.array(rvecs), np.array(tvecs)

def resolve_csv_path():
    """Find a CSV to operate on."""
    for name in INPUT_CSV_CANDIDATES:
        p = SCRIPT_DIR / name
```

```
        if p.exists():
            print(f"[info] Using CSV: {p}")
            return p
    csvs = list(SCRIPT_DIR.glob("*.csv"))
    if len(csvs) == 1:
        print(f"[info] Using the only CSV in folder: {csvs[0]}")
        return csvs[0]
    raise FileNotFoundError(f"CSV not found. Looked for: {INPUT_CSV_CANDIDATES} in
{SCRIPT_DIR}")


def open_or_cache(cap_cache, path_str):
    cap = cap_cache.get(path_str)
    if cap is not None and cap.isOpened():
        return cap
    cap = cv2.VideoCapture(path_str)
    cap_cache[path_str] = cap
    return cap

def seek_frame(cap, frame_idx=None, ts_s=None):
    """Seek cap to frame index or timestamp (seconds)."""
    if frame_idx is not None:
        cap.set(cv2.CAP_PROP_POS_FRAMES, int(frame_idx))
    elif ts_s is not None:
        cap.set(cv2.CAP_PROP_POS_MSEC, float(ts_s) * 1000.0)


def scale_camera_matrix(K: np.ndarray, scale: float) -> np.ndarray:
    """Return a copy of K with focal lengths and principal point scaled by `scale`."""
    K2 = K.copy().astype(np.float64)
    K2[0, 0] *= scale
    K2[1, 1] *= scale
    K2[0, 2] *= scale
    K2[1, 2] *= scale
    return K2
def main(scale_calibrate: float = None):
    csv_path = resolve_csv_path()
    df = pd.read_csv(csv_path)

    # alias common column names
    rename_map = {}
    if "source" in df.columns and "video_path" not in df.columns:
        rename_map["source"] = "video_path"
    if "frame_id" in df.columns and "frame_idx" not in df.columns:
        rename_map["frame_id"] = "frame_idx"
    if "t_frame" in df.columns and "timestamp_s" not in df.columns:
        rename_map["t_frame"] = "timestamp_s"
    df = df.rename(columns=rename_map)

    # alias "camera" -> filename
    if "video_path" in df.columns:
        df["video_path"] = df["video_path"].astype(str).apply(
            lambda s: "camera_video.mp4" if s.strip().lower() == "camera" else s
        )
    else:
```

```
    raise ValueError("Input CSV must contain a 'video_path' column (or 'source').")

if "frame_idx" in df.columns:
    df["frame_idx"] = pd.to_numeric(df["frame_idx"], errors="coerce").astype("Int64")
if "timestamp_s" in df.columns:
    df["timestamp_s"] = pd.to_numeric(df["timestamp_s"], errors="coerce")

if "timestamp_s" in df.columns:
    df["timestamp_rel"] = df.groupby("video_path")["timestamp_s"].transform(lambda s: s - s.min())
else:
    df["timestamp_rel"] = np.nan

cap_cache = {}
fps_cache = {}
out_rows = []

selected = None
t0 = time.time()

# Build candidate dictionaries; filter out None entries (like missing APRILTAG)
try_list = [(n, d) for (n, d) in TRY_DICTS if d is not None]

for ridx, row in df.iterrows():
    video_path = str(row["video_path"]).strip()
    vp = Path(video_path)
    if not vp.is_absolute():
        vp = SCRIPT_DIR / video_path
    if not vp.exists():
        alt = csv_path.parent / video_path
        if alt.exists():
            vp = alt

    cap = open_or_cache(cap_cache, str(vp))
    if not cap or not cap.isOpened():
        if ridx % PRINT_EVERY == 0:
            print(f"[warn] Cannot open video: {vp}")
        continue

    key = str(vp)
    if key not in fps_cache:
        fps = cap.get(cv2.CAP_PROP_FPS)
        if not fps or fps <= 0:
            fps = 30.0
        fps_cache[key] = fps
    fps = fps_cache[key]
  # choose frame index or timestamp
    frame_idx = None
    if "frame_idx" in df.columns and not pd.isna(row.get("frame_idx", np.nan)):
        frame_idx = int(row["frame_idx"])
        ts_rel = frame_idx / float(fps)
    else:
        ts_val = row.get("timestamp_rel", np.nan)
        frame_idx = None
        ts_rel = float(ts_val) if not pd.isna(ts_val) else None
```

```
        try:
            seek_frame(cap, frame_idx=frame_idx, ts_s=ts_rel if frame_idx is None else None)
        except Exception:
            continue

        ok, frame = cap.read()
        if not ok or frame is None:
            continue

        # If UPSCALE_FACTOR != 1, we will run detection on a resized frame
        if UPSCALE_FACTOR != 1:
            frame_proc = cv2.resize(
                frame,
                (int(frame.shape[1] * UPSCALE_FACTOR), int(frame.shape[0] * UPSCALE_FACTOR)),
                interpolation=cv2.INTER_LINEAR,
            )
            scale_for_proc = UPSCALE_FACTOR
        else:
            frame_proc = frame
            scale_for_proc = 1.0

        corners, ids = None, None
        if selected is not None:
            (name, dict_id, dictionary, params, detector, new_api) = selected
            corners, ids = detect_once_on_frame(frame_proc, dictionary, params, detector, new_api)
        else:
            # try different dictionaries until we detect something
            for name, dict_id in try_list:
                dictionary, params, detector, new_api = build_detector(dict_id)
                c, i = detect_once_on_frame(frame_proc, dictionary, params, detector, new_api)
                if i is not None and len(i) > 0:
                    selected = (name, dict_id, dictionary, params, detector, new_api)
                    print(f"[info] Selected dictionary: {name}")
                    corners, ids = c, i
                    break
        if selected is None:
            if ridx % PRINT_EVERY == 0:
                print("[info] No detections yet; still trying dictionaries...")
            continue

        if ids is None or len(ids) == 0:
            if ridx % PRINT_EVERY == 0:
                print(f"[info] f={frame_idx} ts_rel={ts_rel if ts_rel is not None else float('nan'):.3f}s
markers=0")
            continue
        # Build a camera matrix that matches the pixel coords of `frame_proc` (scaled intrinsics)
        if scale_for_proc != 1.0:
            camera_proc = scale_camera_matrix(CAMERA_MATRIX, scale_for_proc)
        else:
            camera_proc = CAMERA_MATRIX

        rvecs, tvecs = estimate_pose(corners, camera_proc, DIST_COEFFS)
        if rvecs is None:
            continue
```

```
        if ridx % PRINT_EVERY == 0:
            print(f"[info] f={frame_idx} ts_rel={ts_rel if ts_rel is not None else float('nan'):.3f}s
markers={len(ids)}")

        # Save rows
        for i, mid in enumerate(ids.flatten().tolist()):
            tvec = tvecs[i].flatten().astype(float)
            rvec = rvecs[i].flatten().astype(float)
            rec = {c: row[c] for c in df.columns if c in row}
            rec.update(
                {
                    "video_path": str(vp),
                    "frame_idx": int(frame_idx) if frame_idx is not None else np.nan,
                    "timestamp_s": ts_rel,
                    "marker_id": int(mid),
                    "x": float(tvec[0]),
                    "y": float(tvec[1]),
                    "z": float(tvec[2]),
                    "rvec0": float(rvec[0]),
                    "rvec1": float(rvec[1]),
                    "rvec2": float(rvec[2]),
                }
            )
            out_rows.append(rec)
            # Save debug frame (draw in original frame resolution)
            if SAVE_DEBUG_FRAMES:
                disp = frame.copy()
                # draw frame axes using original camera matrix so projection goes to original resolution
                try:
                    cv2.drawFrameAxes(disp, CAMERA_MATRIX, DIST_COEFFS, rvecs[i], tvecs[i],
MARKER_SIZE_M * 0.5)
                except Exception:
                    # sometimes drawFrameAxes expects specific types
                    pass
                label = f"id={mid} z={tvec[2]:.3f}m ts={ts_rel if ts_rel is not None else float('nan'):.3f}s
f={rec['frame_idx']}"
                cv2.putText(disp, label, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1.0, (255, 255, 255),
2)
                dbg_dir = Path(str(vp).rsplit(".", 1)[0] + "_aruco_debug")
                dbg_dir.mkdir(parents=True, exist_ok=True)
                out_name = f"{Path(vp).stem}_f{rec['frame_idx'] if rec['frame_idx']==rec['frame_idx'] else
'ts'+str(int((ts_rel or 0)*1000))}.jpg"
                (dbg_dir / out_name).write_bytes(cv2.imencode(".jpg", disp)[1].tobytes())

    # release resources
    for cap in list(cap_cache.values()):
        try:
            if cap:
                cap.release()
        except Exception:
            pass
    if len(out_rows) == 0:
        cols = list(df.columns) + ["marker_id", "x", "y", "z", "vx", "vy", "rvec0", "rvec1", "rvec2",
"timestamp_s", "frame_idx"]
        pd.DataFrame(columns=list(dict.fromkeys(cols))).to_csv(OUTPUT_CSV, index=False)
```

```
        print(f"No detections found. Wrote empty template to: {OUTPUT_CSV}")
        return
    det = pd.DataFrame(out_rows)
    # optional scale calibration using known true distance: if scale_calibrate provided,
    # compute median ratio true / measured_z and print — you can then multiply z,x,y by that factor
    if scale_calibrate is not None:
        measured_median = det["z"].median()
        if measured_median > 0:
            scale_ratio = scale_calibrate / measured_median
            print(f"[calibrate] Provided true distance {scale_calibrate:.3f} m, median measured
z={measured_median:.3f} m")
            print(f"[calibrate] Computed scale ratio to apply to (x,y,z): {scale_ratio:.6f}")
            # apply scale to positions
            det["x"] = det["x"] * scale_ratio
            det["y"] = det["y"] * scale_ratio
            det["z"] = det["z"] * scale_ratio
    # Compute vx, vy per group
    def add_vel(g):
        g = g.sort_values("timestamp_s")
        t = g["timestamp_s"].values
        x = g["x"].values
        y = g["y"].values
        if len(g) >= 3:
            vx = np.gradient(x, t, edge_order=2)
            vy = np.gradient(y, t, edge_order=2)
        else:
            # fallback with simple differences; gradient will still work but be noisy
            vx = np.gradient(x, t)
            vy = np.gradient(y, t)
        g["vx"] = vx
        g["vy"] = vy
        return g
    if "timestamp_s" not in det.columns:
        det["timestamp_s"] = det.get("timestamp_s", np.nan)
    det = det.groupby(["video_path", "marker_id"], group_keys=False).apply(add_vel)

    lead = ["video_path", "frame_idx", "timestamp_s", "marker_id", "x", "y", "z", "vx", "vy", "rvec0",
"rvec1", "rvec2"]
    cols = [c for c in lead if c in det.columns] + [c for c in det.columns if c not in lead]
    det = det[cols]
    det.to_csv(OUTPUT_CSV, index=False)
    print(f"Wrote: {OUTPUT_CSV} in {time.time()-t0:.2f}s")
if __name__ == "__main__":
    ap = argparse.ArgumentParser(description="ArUco detection + pose with proper intrinsics
scaling.")
    ap.add_argument("--calibrate-distance-m", type=float, default=None,
                    help="If you know a true representative distance (meters) for detected markers in the
CSV, provide it to compute/apply a scale correction.")

    args = ap.parse_args(args=[])  # prevents Jupyter from passing -f ...
    main(scale_calibrate=args.calibrate_distance_m)

    main(scale_calibrate=args.calibrate_distance_m)
```

Appendix D – Radar Object Detection Report

Report Title:

Interfacing IWR6843AOPEVM Radar Module with Raspberry Pi: Setup, Configuration, and Results

Report Objectives:

- Interface the radar module with the Raspberry Pi and to document the steps and setup.
- Create the radar parameter's configuration file and to understand and study the implications of each parameter and the inter-relations.
- Modify the data parsing code to plot the velocity versus range plot.
- Test the data parsing code and to document the results obtained for a given scenario.

Executive Summary:

This report details the process of interfacing the radar module with the Raspberry Pi, including obtaining XY and velocity versus range plots for a given scenario. The interfacing involved setting up the radar module in the flashing mode and then on the functional mode, setting up the Raspberry Pi, verifying the recognition of the radar's port by the Raspberry Pi, downloading the required libraries and function script for the data parsing code, creating the radar configuration file and studying the parameters configured, testing the communication between the Raspberry Pi and radar module, and finally running the modified data parsing code.

Work Contribution:

This task was implemented and documented by Shayma Alteneiji

I. Task Description

In this report, the steps and code used to successfully interface the radar module with the Raspberry Pi are detailed. Additionally, the obtained XY (or Range versus Cross-Range) and the Velocity versus Range plots are included for a given scenario. The following section lists the components and equipment used to implement the task.

II. Components and Equipment

1. IWR6843AOPEVM Radar Module
2. Radar Module Mounter
3. Micro USB-to-USB cable
4. Raspberry Pi 5
5. Micro SD card 64 GB
6. SD Card Reader
7. Power Supply for the Raspberry Pi
8. Micro HDMI-to-HDMI Cable
9. Monitor
10. Keyboard
11. Mouse
12. Raspberry Pi Case and Fan
13. USB

III. Methodology

The phases involved in achieving the specified task included: setting up the radar module on the flashing mode and then on the functional mode, setting up the Raspberry Pi, ports recognition verification, downloading the required libraries and function script, creating and using the radar configuration file, testing the communication between the Raspberry Pi and radar module, and finally running the IWR6843 data parsing code.

III.I Setting up the Radar Module in the Flashing and Functional Modes

In section, we detail the steps of setting up the EVM on the functional mode before it can be interfaced with the Raspberry Pi. However, this requires setting up the radar module in the flashing mode first. In the flashing mode, the EVM is flashed with a newer SDK version than the one it comes pre-installed with [1]. The loaded serial flash includes a valid MSS (Management Information System) application and DSS (Decision Support System) application that the EVM's bootloader uses in the functional mode.

To carry out this task, the instructions detailed in the "Hardware Setup for IWR6843AOP" by TI [2] were followed. The following additional steps were not thoroughly explained in the video and are detailed below.

- At minute 3:58, if the COM ports "Silicon Labs Dual CP210x USB to UART Bridge: Enhanced COM Port (COMx)" and "Silicon Labs Dual CP210x USB to UART Bridge: Standard COM Port (COMy)" do not appear in the Device Manger tab, instead, two devices appear under "Other Devices", as shown in Figure 1, it means the required drivers are not installed. To fix this problem, use the link provided in [3] to download the "CP210x VCP (Virtual COM Port) driver", then restart your PC, and the EVM's ports should appear under "Ports (COM & LPT)".

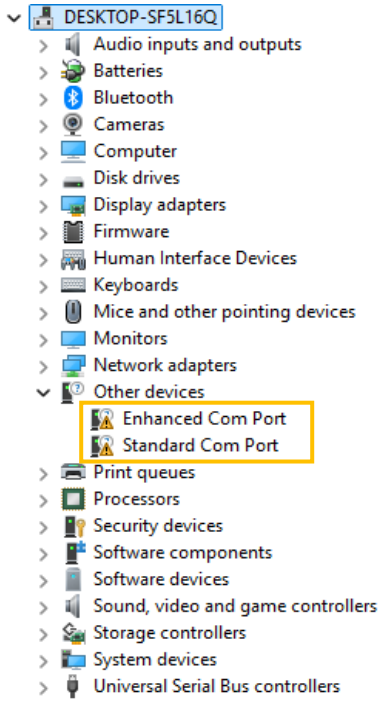


Figure 1. The device manager tab before installing the CP210x VCP (Virtual COM Port) driver.

- At minute 5:12, to install and locate the binary image file to be flashed into the EVM, download TI's Radar Toolbox folder found at this link [4]; the download button can be found at the top right side of the webpage. Use the following directory: <RADAR_TOOLBOX_INSTALL_DIR>\source\ti\examples\Out_Of_Box_Demo\prebuilt_binaries\ to locate the "out_of_box_6843_aop.bin" file in the downloaded Radar Toolbox folder [5]. The used .bin image file has been uploaded into the "Radar Module Related documents" folder in the MS teams group.

After completing the steps in the video, the radar module should be successfully set in the functional mode. The next section describes the setup and steps of setting up the Raspberry Pi.

III. II Setting Up the Raspberry Pi

This section details the steps and setup of setting up the Raspberry pi. Using the equipment and components mentioned in section II, follow the follow steps:

1) first use the SD card reader to attach the micro-SD card to the PC.
2) Search “Raspberry Pi imager” and download the package for Windows (or Mac or Linux, based on the device used).
3) When the installation is over, open the downloaded file using the “Run as administrator” option. The Raspberry Pi imager tab, shown in figure 2, should appear.
4) For the “CHOOSE DEVICE” option, select “Raspberry Pi 5” from the list of Raspberry Pi devices. For the “CHOOSE OS” option, select “Raspberry Pi OS (64-bit)” from the list of operating systems. For the “CHOOSE STORAGE” option, select the option that appears.
5) Press “Next”, and the tab shown in figure 3 should appear; Select “Edit Settings” and configure the Wireless LAN with the information of the server to connect to when using the Raspberry Pi, and then select “Save”.
6) Finally, select “yes” for the tab shown in figure 3.

After completing the previous steps, the operating system should be successfully installed into the micro-SD card, and it can be removed from the SD card reader and attached to the Raspberry Pi.

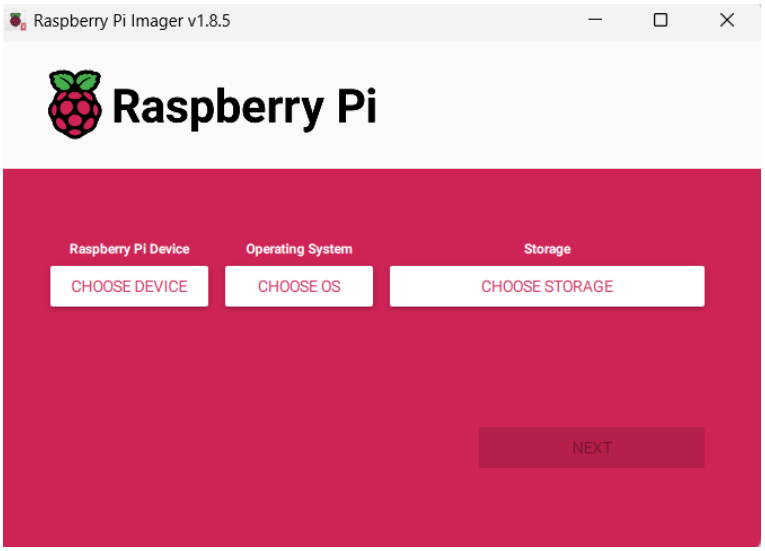


Figure 2. The Raspberry Pi imager tab.

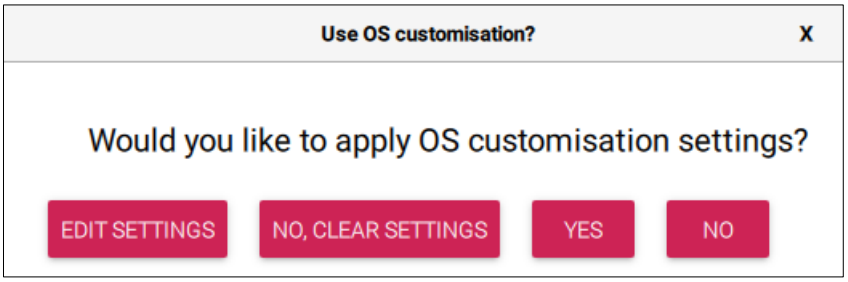


Figure 3. The “Use OS customization” tab.

After that, use the Hardware setup shown in figure 4 for the connections for the Raspberry Pi with the peripherals and the radar module. When turning on the Raspberry Pi's power supply, the Raspberry Pi's desktop must appear. Open the terminal window found on the top left side of the desktop screen. The next section discusses the procedures to verify that the Raspberry Pi is able to detect the connected radar module.

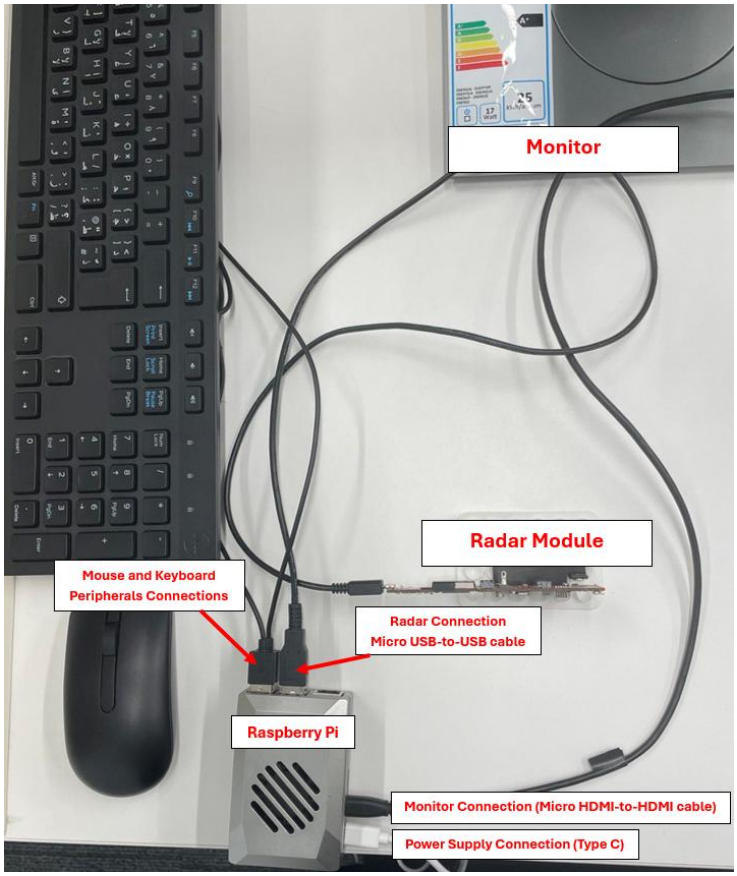


Figure 4. Hardware setup for the Raspberry Pi and Radar Module interfacing.

III. III Port Recognition Verification

To verify that the Raspberry Pi can recognize and detect the radar module, run the following code on terminal:

```
ls /dev/ttyUSB*
```

If the radar module is successfully detected, the terminal would print the following:

```
/dev/ttyUSB0    /dev/ttyUSB1
```

To ensure that the Raspberry Pi can access the ports, run the following code on Terminal:

```
sudo usermod -aG dialout $(whoami)
```

Then reboot the Rassberry Pi using the following code on Terminal:

```
sudo reboot
```

The next section discusses the terminal codes for downloading the libraries and the function script required for the data parsing code.

III. IV Downloading the Required Libraries and Function Script

Initially it was planned to create a virtual environment to run the data parsing code, however due to the difficulty of downloading the PyQt5 library on the virtual environment, instead all libraries were installed system wide. To install the numpy, run the following code on terminal:

```
sudo apt upgrade
sudo apt install numpy
```

To install the pyserial librabry:

```
sudo pip install --break-system-packages pyserial
```

To install PyQtGraph and PyQt5:

```
sudo apt install python3-pyqtgraph
sudo apt install python3-pyqt5
```

To install the parser_mmw_demo.py function script, we do the following:

1) To install the function script into the Raspberry Pi, use the link to the Github file and run the following code:

```
wget https://raw.githubusercontent.com/kirkster96/IWR6843-Read-Data-Python-MMWAVE-SDK/main/parser_mmw_demo.py
```

2) To move the parser function file to the same directory as that of the data parsing script (/home/Shayma/), run the following code:

```
mv parser_mmw_demo.py /home/Shayma/
```

In the following section we discuss creating the configuration file and studying the parameters configured.

III. V Creating the Radar Configuration File

The configuration file can be created and customized to the EVM's application using the Texas Instrument's mmWave_Demo_Visualizer browser-based application. Figure 5 shows the "Configure"

tab of the mmWave_Demo_Visualizer webpage. Figure 6 shows an example of the content of a configuration file. The details of the parameters included in the configuration is discussed below.

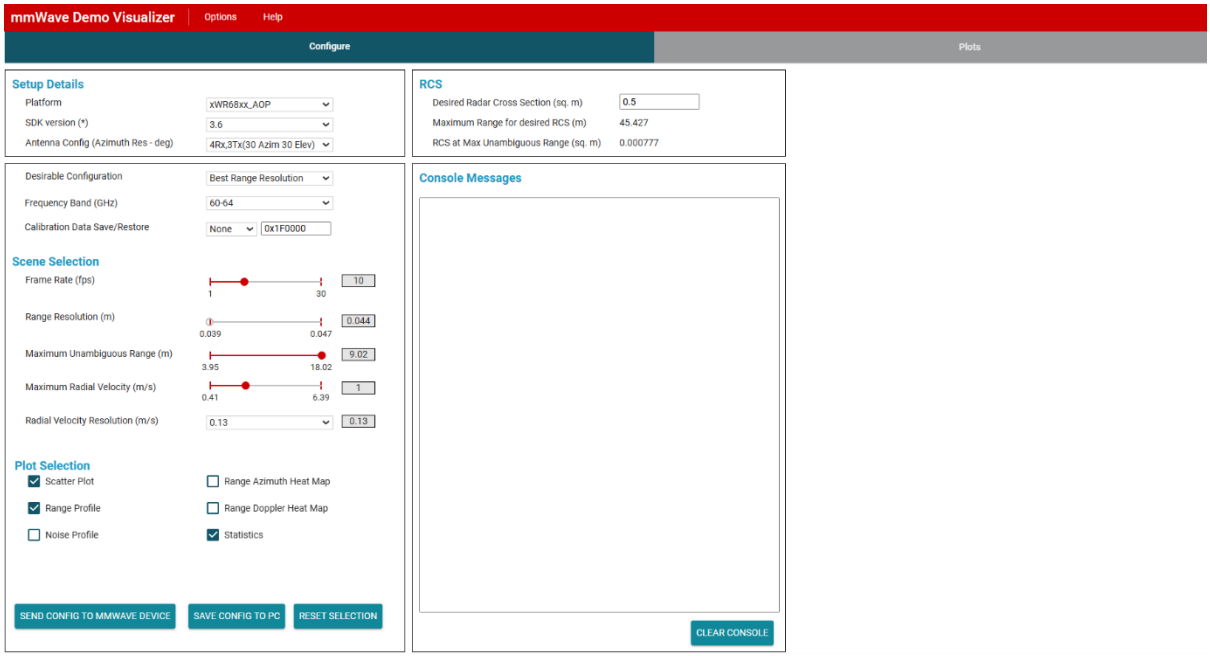


Figure 5. Webpage screen of the mmWave_Demo_Visualizer browser-based application, Configure tab.

```
% ****************************************************************
% Created for SDK ver:03.06
% Created using Visualizer ver:3.6.0.0
% Frequency:60
% Platform:xWR68xx_AOP
% Scene Classifier:best_range_res
% Azimuth Resolution(deg):30 + 30
% Range Resolution(m):0.039
% Maximum unambiguous Range(m):11.99
% Maximum Radial Velocity(m/s):1.54
% Radial velocity resolution(m/s):0.2
% Frame Duration(msec):1000
% RF calibration data:None
% Range Detection Threshold (dB):15
% Doppler Detection Threshold (dB):15
% Range Peak Grouping:enabled
% Doppler Peak Grouping:enabled
% Static clutter removal:disabled
% Angle of Arrival FoV: Full FoV
% Range FoV: Full FoV
% Doppler FoV: Full FoV
% ****************************************************************
sensorStop
flushCfg
dfeDataOutputMode 1
channelCfg 15 7 0
adcCfg 2 1
adcbufCfg -1 0 1 1 1
profileCfg 0 60 70 7 200 0 0 20 1 384 2000 0 0 158
chirpCfg 0 0 0 0 0 0 0 1
chirpCfg 1 1 0 0 0 0 0 2
chirpCfg 2 2 0 0 0 0 0 4
frameCfg 0 2 16 0 1000 1 0
lowPower 0 0
guiMonitor -1 1 1 0 0 0 1
cfarCfg -1 0 2 8 4 3 0 15 1
cfarCfg -1 1 0 4 2 3 1 15 1
multiObjBeamForming -1 1 0.5
clutterRemoval -1 0
calibDcRangeSig -1 0 -5 8 256
extendedMaxVelocity -1 0
lvdsStreamCfg -1 0 0 0
compRangeBiasAndRxChanPhase 0.0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1 0
measureRangeBiasAndRxChanPhase 0 1.5 0.2
CQRxSatMonitor 0 3 19 125 0
CQSigImgMonitor 0 127 6
analogMonitor 0 0
aoaFovCfg -1 -90 90 -90 90
cfarFovCfg -1 0 0 12.00
cfarFovCfg -1 1 -1.54 1.54
calibData 0 0 0
sensorStart
```

Figure 6. Example of the content of a configuration file.

In figure 5, for the “Setup Details”, the xWR68xx_AOP was selected as the platform, matching the TI IWR6843AOP radar sensor used. The “SDK version” is a drop- down menu that asks the user to select the SDK version that the mmWave device was flashed with. In case there is a mismatch in the selected SDK version, the graphical user interface (GUI) will show an error when using the “Send Config to mmWave Device button” or “Load config from PC and send” buttons [6]. The antenna configuration is also a drop- down menu, figure 5 shows the correct configuration to be selected (4 Rx, 3 Tx).

The “Desirable Configuration” is a drop – down menu that allows the user to select the specification that is ensured to be met above the other parameters. This is because certain parameters affect the values of other parameters. When a particular desirable configuration is selected, the range or values of the parameters given under the “Scene Selection” (figure 5) are adjusted accordingly. Moreover, how different parameters affect other parameters also change. Table 1 summarizes each parameter’s dependent parameters and the relation between them [6]. As the table shows, the desirable configuration options include best range resolution, best velocity resolution, or best range. The default settings uses the best range resolution option (figure 5).

The frequency band is set by default depending on the selected platform. Under “Scene Selection,” the parameter frame rate describes the rate at which the frames are shipped out of the mmWave radar sensor [6]. A frame consists of N chirps, each frame is separated by a duration called the inter frame time. Figure 7 shows a typical FMCW chirp, and figure 8 shows a typical frame structure, showing N chirps (frame) and M chirps (anther frame) separated by the inter frame time duration.

The frame period (T_f) can therefore be expressed as $N*Tc + T_{IFT}$, where Tc is the chirp time or sweep time and T_{IFT} is the inter frame time. During T_{IFT}, the radar processes and transmits the captured data. A higher frame rate reduces the frame periodicity (T_f) and hence T_{IFT}, which may lead to data loss if the resulting T_{IFT} is insufficient to transmit the large amount of captured data over the relatively low baud rate of the transmission protocol [6]; it is therefore suggested to select a relatively low frame rate (3 – 10 fps) [6].

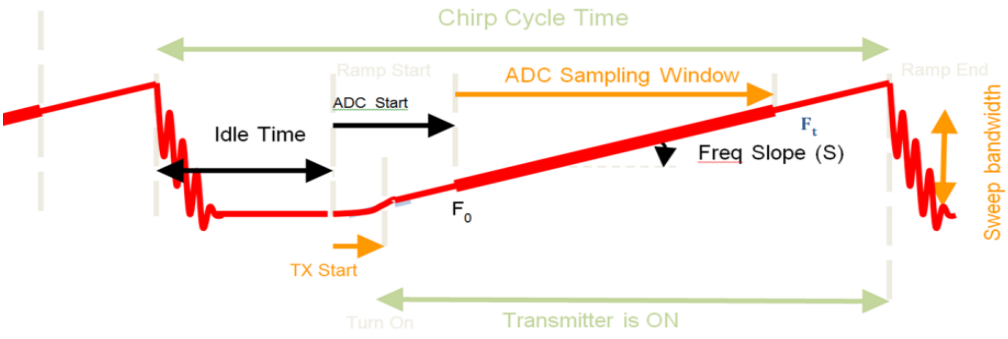


Figure 7. Typical FMCW chirp [6].

Table 1. Summary of the Dependency on the Relations Between the Radar Configuration Parameters for a given Desired Configuration

	Scene Selected											
	Best Range Resolution				Best Range Resolution						Best Range Resolution	
	Dependent Parameters		Relation		Dependent Parameters			Relation			Dependent Parameters	Relation
Configuration Parameters	P1	P2	P1	P2	P1	P2	P3	P1	P2	P3	P1	P1
Frame Rate (fps)	The minimum value of the maximum radial velocity	-	Proportional relation. The greater the frame rate, the larger is the minimum maximum radial velocity.	-	The minimum value of the maximum radial velocity	Radial Velocity Resolution	Range Resolution	Proportional relation. The greater the frame rate, the larger is the minimum maximum radial velocity.	The lower the frame rate, the finer the radial velocity resolution.	Higher frame rates provide more finer range resolution options.	The minimum value of the maximum radial velocity	Proportional relation. The greater the frame rate, the larger is the minimum maximum radial velocity.
Range Resolution (m)	Maximum Unambiguous Range	Maxim-um Radial Velocity	Proportional. The finer the range resolution is, the longer is the maximum unambiguous range.	Inverse relation. The finer the range resolution, the lower the Max radial velocity.	Maximum Unambig-uous Range	-	-	Proportional. The finer the range resolution, the shorter the maximum unambiguous range.	-	-	Maximum Radial Velocity	The coarser the selected range resolution, the more and higher the options for the maximum radial velocity.
Maximum Unambiguous Range (m)	Radial Velocity Resolution (drop – down menu)	-	The longer the maximum unambiguous range, the lesser the number of better options for the radial velocity resolution.	-	Range Resolution	-	-	The longer the Max unambiguous range, the finer the range resolution.	-	-	Range Resolution	The lower the maximum unambiguous range, the lower but finer options for range resolution.
Maximum Radial Velocity (m/s)	Radial Velocity Resolution	-	Inverse relation. The lower the maximum radial velocity, the finer the radial velocity resolution.	-	Range Resolution	Maximum Unambiguous Range	-	Inverse relation. The greater the maximum radial velocity, the coarser the range resolution.	Inverse relation. The greater the maximum radial velocity, the shorter the maximum unambig-uous range.	-	Radial Velocity Resolution	The lower Max. radial velocity, the finer the radial velocity resolution.
Radial Velocity Resolution (m/s)	Drop – down menu, constrained by the other parameters.				A direct function of the frame rate. The radial velocity resolution is fixed to the optimal possible value for the selected frame rate.						Drop – down menu, constrained by the other parameters.	

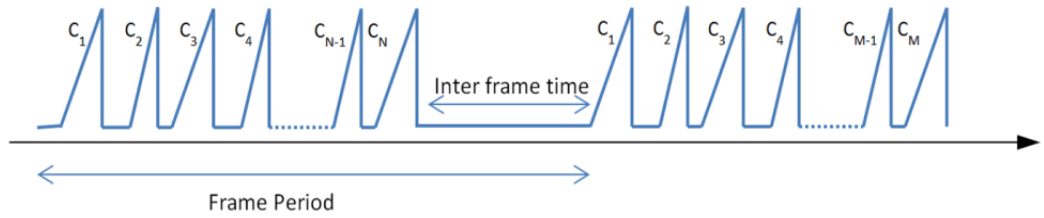


Figure 8. Typical frame structure [6].

The equations of the other four parameter under "Scene Selection" is given in Table 2 (the derivation of the equations is planned to be discussed in a separate report) [7]. Moreover, due to the complexity in the dependency of the parameters as a function of the selected desirable configuration, the relation between the parameters cannot be easily understood using the equations given. We may instead use Table 1 to study the relations and fine – tune the parameters to achieve the desired specifications. For that reason, for the current stage, the default values of the parameters have been used.

The "Plot Selection" is specified when interfacing the radar module with the PC. After having understood the mmWave device configuration parameters and how to control them to obtain a particular requirement, next we explore using sending the configuration file to the sensor via the Raspberry Pi to test the success of the communication between it and the EVM.

Table 2. Radar Configuration Parameters Equations

	Resolution	Maximum
Range	$\frac{c}{2B}$	$\frac{cF_s}{2\mu}$
Radial Velocity	$\frac{\lambda}{2T_f}$	$\frac{\lambda}{4T_C}$

III. VI Testing the Communication between the mmWave device and the Raspberry Pi

After having configured the radar parameters according to a given application specification, click on "SAVE CONFIG TO PC" found at the bottom of the screen of the configure tab (figure 5). This saves a file with .cfg extension. Upload this file into the Raspberry Pi using a USB. Run the command given below on terminal to determine the directory to the uploaded configuration file.

```
find ~ -name "*.cfg"
```

The Python code in Appendix A verifies access to the mmWave device's serial ports, sends the configuration file, and reads incoming data. Use the output of the terminal command given above (e.g. /home/Shayma/sdp/bin/config_1.cfg) and copy it in place of the configuration file name in the code. Use the output of the terminal command given in section III. III for lines five and six of the code, respectively.

Since the code does not parse data, we expect the Raspberry Pi to print gibberish, as shown in Figure 9. Once the Raspberry Pi is verified to successfully communicate with the sensor module, we may upload the data parsing code with lesser potential issues to debug. The modifications and the running of the data parsing code is explored in the next section.

III. VII The Data Parsing Code

The used data parsing code is given in Appendix B, which was taken from the github source given at source [8]. The code has been modified to display the velocity versus range plot. As with the previous section, use the output of the terminal command given in section III. III for the (CLI) and (DATA) ports directory, and the output of the terminal command given in section III. VI for the configuration file directory. The following section shows the XY and velocity versus range plots obtained for a particular scenario.

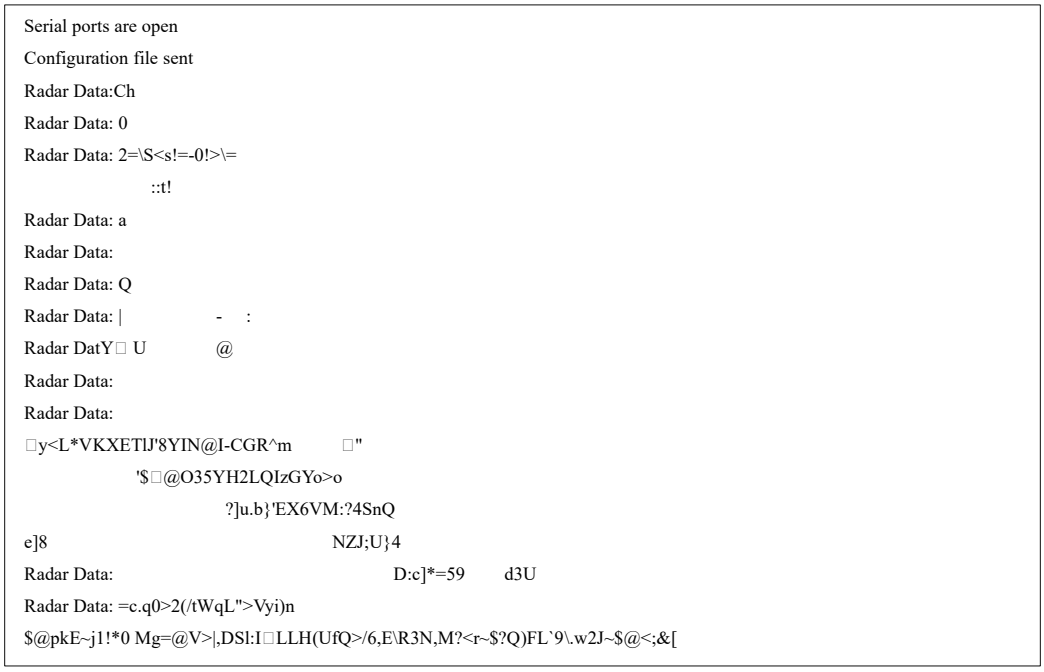


Figure 9. Terminal output when running the Python code given in appendix A.

IV. Results

Figures 10(a) – 10(l) shows the data parsing code output plots for the given scenario. When our team member (Engy) is not moving (figure 10(a)), the XY plot (figure 10(b)) shows a total of six detected objects, and the velocity vs. range plot (figure 10(c)) shows the detected objects to have a velocity of zero. When the team member starts to move (figure 10(d)), the velocity of three dots (at range = 4 m) becomes non – zero (figure f), and the XY plot (figure 10(e)) displays new points around the same range (points with coordinates (~4 m, ~1m)).

As the team member moves closer and closer (figure 10(g) and 10(j)), the range of the nonzero velocity points and the x- coordinates of the XY plot decreases, as can be observed in figures 10(h) and 10(k) for the XY plots, and figures 10(i) and 10(l) for the velocity versus range plots. On terminal, the following is outputted continuously for each of the data packets parsed.

{'numObj': 6, 'range': [0.6104857094483974, 1.0901529643987067, 1.2209713293732982, 2.6599731940321663, 8.459586957524877, 0.17442446723933996], 'x': [0.12879982590675354, 0.8871416449546814, 0.8831987977027893, 0.16034263372421265, -2.2947397232055664, 0.015771405771374702], 'y': [0.594185471534729, 0.6301546096801758, 0.8027286529541016, 1.12474524974823, 5.371487140655518, 0.11358580738306046], 'z': [0.05519992485642433, -0.06571419537067413, 0.2575996518135071, -2.405139446258545, 6.1193060874938965, 0.13142839074134827], 'v': [0.0, 0.0, 0.0, 0.0, 0.0, 0.36213648319244385]}
Bytes read: 768

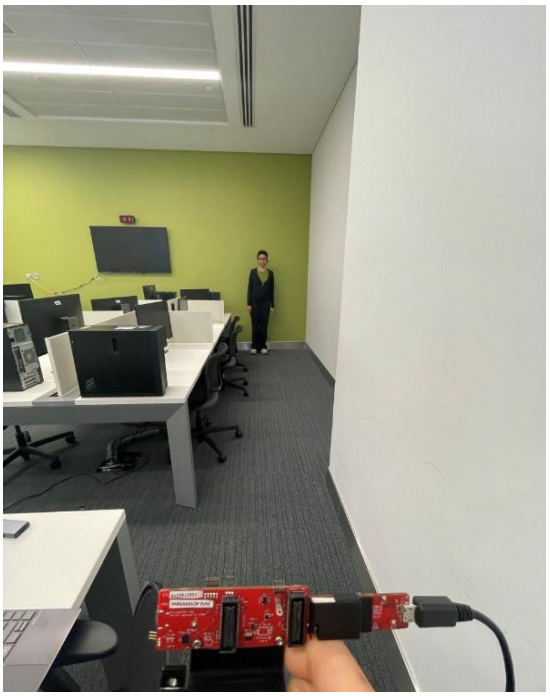


Figure 10(a). First scenario frame. Team member is static.

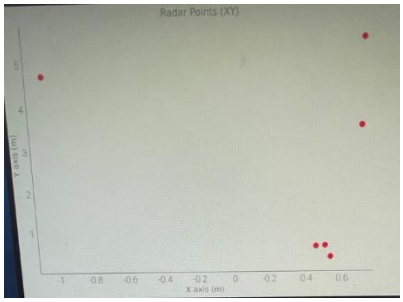


Figure 10(b). XY plot corresponding to the first scenario frame.

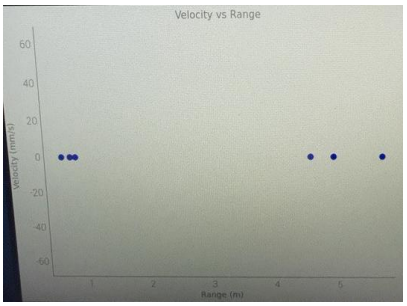


Figure 10(c). Velocity versus Range plot corresponding to the first scenario frame.

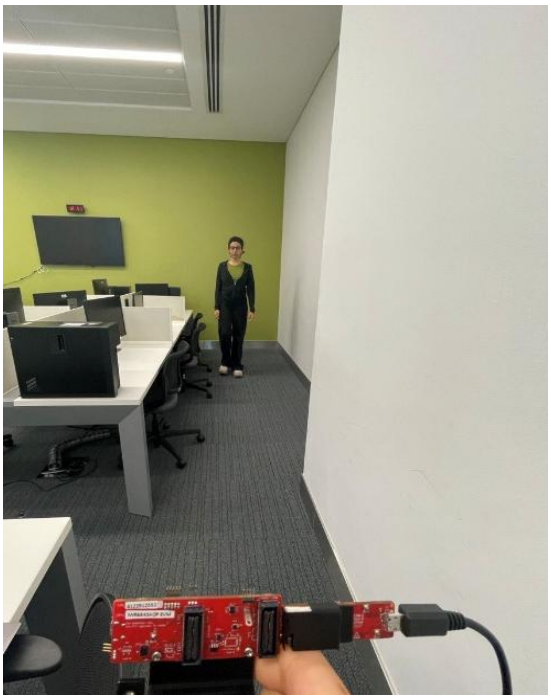


Figure 10(d). Second scenario frame.

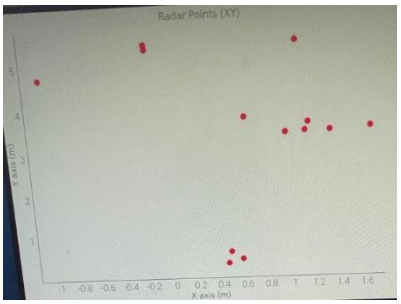


Figure 10(e). XY plot corresponding to the second scenario frame.

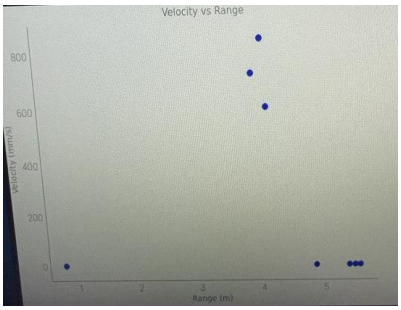


Figure 10(f). Velocity versus Range plot corresponding to the second scenario frame.

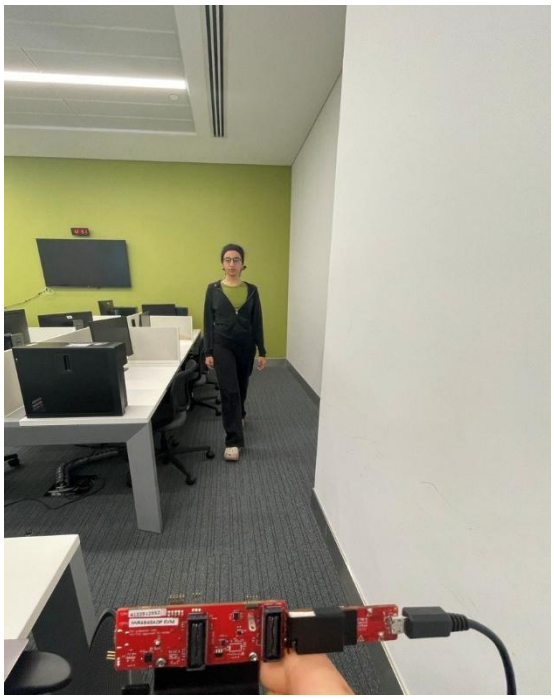


Figure 10(g). Third scenario frame.

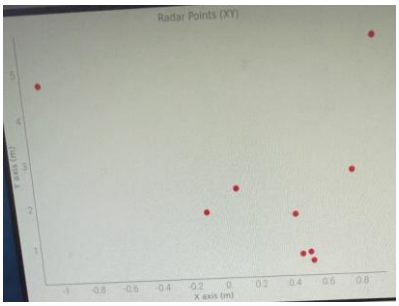


Figure 10(h). XY plot corresponding to the third scenario frame.

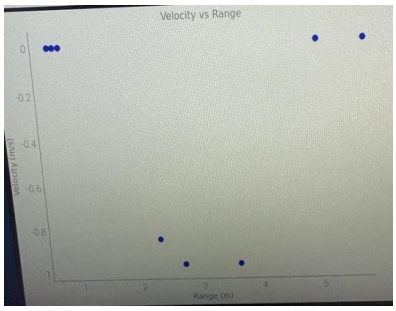


Figure 10(i). Velocity versus Range plot corresponding to the third scenario frame.

Figure 10(j). Fourth scenario frame.

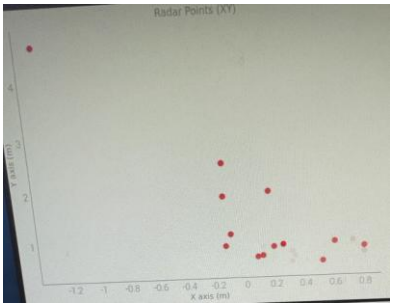


Figure 10(k). XY plot corresponding to the fourth scenario frame.

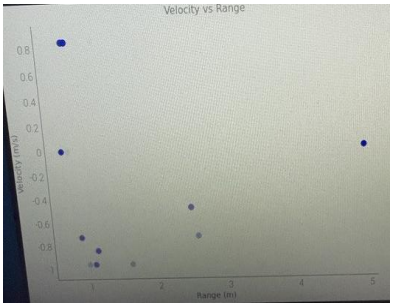


Figure 10(l). Velocity versus Range plot corresponding to the fourth scenario frame.

V. References

[1]. sgs-weather-and-environmental-systems, "TI-mmWave-SDK/mmwave_sdk_01_02_00_05/docs/mmwave_sdk_user_guide.pdf at master · sgs-weather-and-environmental-systems/TI-mmWave-SDK," *GitHub*, 2018. https://github.com/sgs-weather-and-environmental-systems/TI-mmWave-SDK/blob/master/mmwave_sdk_01_02_00_05/docs/mmwave_sdk_user_guide.pdf (accessed Mar. 20, 2025).

[2]. "Hardware Setup for IWR6843AOP," *Ti.com*, 2023. https://www.ti.com/video/6205846491001 (accessed Mar. 20, 2025).

[3]. "Silicon Labs," *Silabs.com*, 2024. https://www.silabs.com/developer-tools/usb-to-uart-bridge-vcp-drivers?tab=downloads (accessed Mar. 20, 2025).

[4]. "Radar Toolbox for mmWave Sensors," *TI Resource Explorer* . https://dev.ti.com/tirex/explore/node?node=A__AEIJm0rwIeU.2P1OBWwlaA__radar_toolbox__1AslXXD__LATEST (accessed Mar. 20, 2025).

[5]. "Out Of Box Demo User Guide," *Ti.com*, 2025. https://dev.ti.com/tirex/explore/node?node=A__AXAenV2u4woV.FhTlAk68Q__radar_toolbox__1AslXXD__LATEST (accessed Mar. 20, 2025).

[6]. "mmWave Demo Visualizer User's Guide mmWave Demo Visualizer," 2017. Accessed: Mar. 20, 2025. [Online]. Available: https://www.ti.com/lit/ug/swru529c/swru529c.pdf?ts=1742455933180&ref_url=https%253A%252F%252Fwww.google.com%252F

[7]. Q. Chaudhari, "FMCW Radar Part 3 - Design Guidelines | Wireless Pi," *Wireless Pi*, Nov. 30, 2023. https://wirelesspi.com/fmcw-radar-part-3-design-guidelines/ (accessed Mar. 20, 2025).

[8]. kirkster96, "IWR6843-Read-Data-Python-MMWAVE-SDK/parser_mmw_demo.py at main · kirkster96/IWR6843-Read-Data-Python-MMWAVE-SDK," *GitHub*, 2021. https://github.com/kirkster96/IWR6843-Read-Data-Python-MMWAVE-SDK/blob/main/parser_mmw_demo.py (accessed Mar. 20, 2025).

VI. Appendices

VI. I Appendix A – Python code for Verifying the Communication between the mmWave device and the Raspberry Pi

Generated using the assistance of an AI model

```
import serial
import time

CMD_PORT = "/dev/ttyUSB0"
DATA_PORT = "/dev/ttyUSB1"

# Open command and data ports
cmd_uart = serial.Serial(CMD_PORT, baudrate=115200, timeout=1)
data_uart = serial.Serial(DATA_PORT, baudrate=921600, timeout=1)

if cmd_uart.is_open and data_uart.is_open:
    print("Serial ports are open")
else:
    print("Failed to open serial ports")

def send_config_file(filename):
    """Send a configuration file line by line to the radar."""
    with open(filename, 'r') as file:
        for line in file:
            cmd_uart.write(line.encode() + b'\r\n')  # Send each line
            time.sleep(0.05)  # Delay for processing
    print("Configuration file sent")
def read_data():
    """Read and print data from the radar."""
    while True:
        if data_uart.in_waiting > 0:
            data = data_uart.readline().decode(errors="ignore").strip()
            print("Radar Data:", data)
# Send the config file before reading data
send_config_file("/home/Shayma/sdp/bin/config_1.cfg")
# Start reading radar data
read_data()
```

VI. II Appendix B – Data Parsing Code

```
# Adapted from a cited Github source and edited using the assistance of an AI model


import serial
import time
import numpy as np
import os
import sys
from PyQt5 import QtWidgets, QtCore
import pyqtgraph as pg
from pyqtgraph.Qt import QtGui
# import the parser function
from parser_mmw_demo import parser_one_mmw_demo_output_packet

# Change the configuration file name
configFileName = '/home/Shayma/sdp/config_2.cfg'

# Change the debug variable to use print()
DEBUG = False

# Constants
maxBufferSize = 2**15;
CLIport = {}
Dataport = {}
byteBuffer = np.zeros(2**15,dtype = 'uint8')
byteBufferLength = 0;
maxBufferSize = 2**15;
magicWord = [2, 1, 4, 3, 6, 5, 8, 7]
detObj = {}
frameData = {}
currentIndex = 0
# word array to convert 4 bytes to a 32 bit number
word = [1, 2**8, 2**16, 2**24]

# Function to configure the serial ports and send the data from
# the configuration file to the radar
def serialConfig(configFileName):

    global CLIport
    global Dataport
    # Open the serial ports for the configuration and the data ports

    # Raspberry pi
    CLIport = serial.Serial('/dev/ttyUSB0', 115200)
    Dataport = serial.Serial('/dev/ttyUSB1', 921600)

    print("CLIport open:", CLIport.isOpen())  # Debugging line
    print("Dataport open:", Dataport.isOpen())  # Debugging line

    # Read the configuration file and send it to the board
    config = [line.rstrip('\r\n') for line in open(configFileName)]
    for i in config:
```

```
        CLIport.write((i+'\n').encode())
        print(i)
        time.sleep(0.01)

    return CLIport, Dataport

# Function to parse the data inside the configuration file
def parseConfigFile(configFileName):
    configParameters = {} # Initialize an empty dictionary to store the configuration parameters

    # Read the configuration file and send it to the board
    config = [line.rstrip('\r\n') for line in open(configFileName)]
    for i in config:

        # Split the line
        splitWords = i.split(" ")

        # Hard code the number of antennas, change if other configuration is used
        numRxAnt = 4
        numTxAnt = 3

        # Get the information about the profile configuration
        if "profileCfg" in splitWords[0]:
            startFreq = int(float(splitWords[2]))
            idleTime = int(splitWords[3])
            rampEndTime = float(splitWords[5])
            freqSlopeConst = float(splitWords[8])
            numAdcSamples = int(splitWords[10])
            numAdcSamplesRoundTo2 = 1;

            while numAdcSamples > numAdcSamplesRoundTo2:
                numAdcSamplesRoundTo2 = numAdcSamplesRoundTo2 * 2;

            digOutSampleRate = int(splitWords[11]);

        # Get the information about the frame configuration
        elif "frameCfg" in splitWords[0]:

            chirpStartIdx = int(splitWords[1]);
            chirpEndIdx = int(splitWords[2]);
            numLoops = int(splitWords[3]);
            numFrames = int(splitWords[4]);
            framePeriodicity = int(splitWords[5]);


    # Combine the read data to obtain the configuration parameters
    numChirpsPerFrame = (chirpEndIdx - chirpStartIdx + 1) * numLoops
    configParameters["numDopplerBins"] = numChirpsPerFrame / numTxAnt
    configParameters["numRangeBins"] = numAdcSamplesRoundTo2
    configParameters["rangeResolutionMeters"] = (3e8 * digOutSampleRate * 1e3) / (2 *
freqSlopeConst * 1e12 * numAdcSamples)
    configParameters["rangeIdxToMeters"] = (3e8 * digOutSampleRate * 1e3) / (2 * freqSlopeConst *
1e12 * configParameters["numRangeBins"])
    configParameters["dopplerResolutionMps"] = 3e8 / (2 * startFreq * 1e9 * (idleTime +
rampEndTime) * 1e-6 * configParameters["numDopplerBins"] * numTxAnt)
```

```
    configParameters["maxRange"] = (300 * 0.9 * digOutSampleRate)/(2 * freqSlopeConst * 1e3)
    configParameters["maxVelocity"] = 3e8 / (4 * startFreq * 1e9 * (idleTime + rampEndTime) * 1e-6
* numTxAnt)

    return configParameters

##################################################################################################
# USE parser_mmw_demo SCRIPT TO PARSE ABOVE INPUT FILES
##################################################################################################
def readAndParseData14xx(Dataport, configParameters):
    #load from serial
    global byteBuffer, byteBufferLength

    # Initialize variables
    magicOK = 0 # Checks if magic number has been read
    dataOK = 0 # Checks if the data has been read correctly
    frameNumber = 0
    detObj = {}

    readBuffer = Dataport.read(Dataport.in_waiting)
    byteVec = np.frombuffer(readBuffer, dtype = 'uint8')
    byteCount = len(byteVec)
    print(f"Bytes read: {byteCount}")

    # Check that the buffer is not full, and then add the data to the buffer
    if (byteBufferLength + byteCount) < maxBufferSize:
        byteBuffer[byteBufferLength:byteBufferLength + byteCount] = byteVec[:byteCount]
        byteBufferLength = byteBufferLength + byteCount

    # Check that the buffer has some data
    if byteBufferLength > 16:

        # Check for all possible locations of the magic word
        possibleLocs = np.where(byteBuffer == magicWord[0])[0]
        print(f"Byte buffer: {byteBuffer[:byteBufferLength]}")

 # Confirm that is the beginning of the magic word and store the index in startIdx
        startIdx = []
        for loc in possibleLocs:
            check = byteBuffer[loc:loc+8]
            if np.all(check == magicWord):
                startIdx.append(loc)

        # Check that startIdx is not empty
        if startIdx:

            # Remove the data before the first start index
            if startIdx[0] > 0 and startIdx[0] < byteBufferLength:
                byteBuffer[:byteBufferLength-startIdx[0]] = byteBuffer[startIdx[0]:byteBufferLength]
                byteBuffer[byteBufferLength-startIdx[0]:] = np.zeros(len(byteBuffer[byteBufferLength-
startIdx[0]:]),dtype = 'uint8')
                byteBufferLength = byteBufferLength - startIdx[0]

            # Check that there have no errors with the byte buffer length
            if byteBufferLength < 0:
```

```
            byteBufferLength = 0

        # Read the total packet length
        totalPacketLen = np.matmul(byteBuffer[12:12+4],word)
        # Check that all the packet has been read
        if (byteBufferLength >= totalPacketLen) and (byteBufferLength != 0):
            magicOK = 1

    # If magicOK is equal to 1 then process the message
    if magicOK:
        # Read the entire buffer
        readNumBytes = byteBufferLength
        if(DEBUG):
            print("readNumBytes: ", readNumBytes)
        allBinData = byteBuffer
        if(DEBUG):
            print("allBinData: ", allBinData[0], allBinData[1], allBinData[2], allBinData[3])

        # init local variables
        totalBytesParsed = 0;
        numFramesParsed = 0;

        # parser_one_mmw_demo_output_packet extracts only one complete frame at a time
        # so call this in a loop till end of file
        #
        # parser_one_mmw_demo_output_packet function already prints the
        # parsed data to stdio. So showcasing only saving the data to arrays
        # here for further custom processing
        parser_result, \
        headerStartIndex,  \
        totalPacketNumBytes, \
        numDetObj,  \
        numTlv,  \
        subFrameNumber,  \
        detectedX_array,  \
        detectedY_array,  \
        detectedZ_array,  \
        detectedV_array,  \
        detectedRange_array,  \
        detectedAzimuth_array,  \
        detectedElevation_array,  \
        detectedSNR_array,  \
        detectedNoise_array = parser_one_mmw_demo_output_packet(allBinData[totalBytesParsed::1],
readNumBytes-totalBytesParsed,DEBUG)

        # Check the parser result
        if(DEBUG):
            print ("Parser result: ", parser_result)
        if (parser_result == 0):
            totalBytesParsed += (headerStartIndex+totalPacketNumBytes)
            numFramesParsed+=1
            if(DEBUG):
                print("totalBytesParsed: ", totalBytesParsed)

##################################################################################
```

```
            # TODO: use the arrays returned by above parser as needed.
            # For array dimensions, see help(parser_one_mmw_demo_output_packet)
            # help(parser_one_mmw_demo_output_packet)

##########################################################################################


            # For example, dump all S/W objects to a csv file

            import csv
            if (numFramesParsed == 1):
                democsvfile = open('mmw_demo_output.csv', 'w', newline='')
                demoOutputWriter = csv.writer(democsvfile, delimiter=',', quoting=csv.QUOTE_NONE)
                demoOutputWriter.writerow(["frame","DetObj#","x","y","z","v","snr","noise"])

            for obj in range(numDetObj):
                demoOutputWriter.writerow([numFramesParsed-1, obj, detectedX_array[obj],\
                                           detectedY_array[obj],\
                                           detectedZ_array[obj],\
                                           detectedV_array[obj],\
                                           detectedSNR_array[obj],\
                                           detectedNoise_array[obj]])

            detObj = {"numObj": numDetObj, "range": detectedRange_array, \
                      "x": detectedX_array, "y": detectedY_array, "z": detectedZ_array, "v":
detectedV_array}
            dataOK = 1
        else:
            # error in parsing; exit the loop
            print("error in parsing this frame; continue")

        shiftSize = totalPacketNumBytes
        byteBuffer[:byteBufferLength - shiftSize] = byteBuffer[shiftSize:byteBufferLength]
        byteBuffer[byteBufferLength - shiftSize:] = np.zeros(len(byteBuffer[byteBufferLength -
shiftSize:]),dtype = 'uint8')
        byteBufferLength = byteBufferLength - shiftSize

        # Check that there are no errors with the buffer length
        if byteBufferLength < 0:
            byteBufferLength = 0
        # All processing done; Exit
        if(DEBUG):
            print("numFramesParsed: ", numFramesParsed)

    return dataOK, frameNumber, detObj
class MyWidget(pg.GraphicsView):

    def __init__(self, parent=None):
        super().__init__(parent=parent)

        self.mainLayout = QtWidgets.QVBoxLayout(self)

        # First Plot Layout (for x-y plot)
        self.plotView1 = pg.GraphicsView(self)
        self.setLayout(self.mainLayout)
```

```
        self.noDataLabel = QtWidgets.QLabel("No Data to Plot", self)
        self.noDataLabel.setAlignment(QtCore.Qt.AlignCenter)
        self.noDataLabel.setVisible(False)  # Initially hide the label
        self.mainLayout.addWidget(self.noDataLabel)  # Add the label to the layout

        self.timer = QtCore.QTimer(self)
        self.timer.setInterval(100) # in milliseconds
        self.timer.start()
        self.timer.timeout.connect(self.onNewData)
        self.plotItem1 = pg.PlotItem(title="Radar Points (XY)")
        self.plotView1.setScene(pg.GraphicsScene())  # Set a new scene for the first plot
        self.plotView1.scene().addItem(self.plotItem1)
        self.plotView1.setBackground('w')  # Set background color of the GraphicsView

        self.plotView1.setMinimumSize(400, 300)

        self.mainLayout.addWidget(self.plotView1)
        self.plotItem1.setRange(QtCore.QRectF(-10, -10, 40, 40))  # Adjust plot range for the first plot

        # Create data item for first plot
        self.plotDataItem1 = self.plotItem1.plot([], pen=None, symbolBrush=(25, 125, 0),
symbolSize=5, symbolPen=None)

        self.plotView2 = pg.GraphicsView(self)  # Create a GraphicsView for the second plot
        self.plotItem2 = pg.PlotItem(title="Velocity vs Range")
        self.plotView2.setScene(pg.GraphicsScene())  # Set a new scene for the second plot
        self.plotView2.scene().addItem(self.plotItem2)
        self.plotView2.setBackground('w')  # Set background color of the GraphicsView
        self.plotView2.setMinimumSize(400, 300)

        self.mainLayout.addWidget(self.plotView2)
        self.plotItem2.setRange(QtCore.QRectF(0, 0, 100, 50))  # Adjust plot range for the second plot

        # Create data item for second plot
        self.plotDataItem2 = self.plotItem2.plot([], pen=None, symbolBrush=(0, 0, 255), symbolSize=5,
symbolPen=None)
    def setData(self, x, y):
        self.plotDataItem1.setData(x, y)

    def setData2(self, velocity, range_):
        self.plotDataItem2.setData(range_, velocity)
    # Funtion to update the data and display in the plot
    def update(self):
        dataOk = 0
        global detObj
        x = []
        y = []
        velocity = []
        range_ = []
        # Read and parse the received data
        dataOk, frameNumber, detObj = readAndParseData14xx(Dataport, configParameters)
        if dataOk:
            print("detObj:", detObj)  # Debugging line
        else:
```

```
            print("No data to parse.")  # Debugging line
        if dataOk and len(detObj["x"]) > 0:
            print(detObj)
            x = detObj["x"]
            y = detObj["y"]
            velocity = detObj["v"]
            range_ = detObj["range"]

        return dataOk, x, y, velocity, range_

    def onNewData(self):

        # Update the data and check if the data is okay
        dataOk,newx,newy,velocity,range_  = self.update()

        #if dataOk:
            # Store the current frame into frameData
            #frameData[currentIndex] = detObj
            #currentIndex += 1

        if dataOk and len(newx) > 0 and len(newy) > 0:
            # If data is valid and not empty, update the plot with new data
            self.setData(newx, newy)
            self.noDataLabel.setVisible(False)  # Hide the "No Data" label if data is available
        else:
            # If no data or invalid data, clear the plot and show "No Data to Plot"
            self.setData([], [])  # Clear the plot by passing empty lists
            self.noDataLabel.setVisible(True)  # Show the "No Data" label

        if dataOk and len(velocity) > 0 and len(range_) > 0:
            # Update the second plot with velocity vs range data
            self.setData2(velocity, range_)

def main():
    # Configurate the serial port
    CLIport, Dataport = serialConfig(configFileName)

    # Get the configuration parameters from the configuration file
    global configParameters
    configParameters = parseConfigFile(configFileName)

    app = QtWidgets.QApplication([])

    pg.setConfigOptions(antialias=False) # True seems to work as well
    win = MyWidget()
    win.show()
    win.resize(1500,1000)
    win.raise_()
    app.exec_()
    CLIport.write(('sensorStop\n').encode())
    CLIport.close()
    Dataport.close()

if __name__ == "__main__":
    main()
```

Appendix E - Programming Code on Using the BSA Algorithm on the Raspberry Pi

```
# Generated using the assistance of an AI model

import cv2
import numpy as np

class KalmanBox:
    def __init__(self, x, y, w, h):
        self.kf = cv2.KalmanFilter(8, 4)
        self.kf.measurementMatrix = np.eye(4, 8, dtype=np.float32)
        self.kf.transitionMatrix = np.array([
            [1, 0, 0, 0, 1, 0, 0, 0],
            [0, 1, 0, 0, 0, 1, 0, 0],
            [0, 0, 1, 0, 0, 0, 1, 0],
            [0, 0, 0, 1, 0, 0, 0, 1],
            [0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0, 0, 1]
        ], dtype=np.float32)
        self.kf.processNoiseCov = np.eye(8, dtype=np.float32) * 0.01
        self.kf.measurementNoiseCov = np.eye(4, dtype=np.float32) * 0.1
        self.kf.statePre = np.array([[x], [y], [w], [h], [0], [0], [0], [0]], dtype=np.float32)

    def update(self, x, y, w, h):
        measurement = np.array([[x], [y], [w], [h]], dtype=np.float32)
        self.kf.correct(measurement)
        pred = self.kf.predict()
        px, py, pw, ph = pred[0], pred[1], pred[2], pred[3]
        return int(px), int(py), int(pw), int(ph)

# GStreamer camera input (edit if needed)
gst = (
    "libcamerasrc ! "
    "video/x-raw,format=NV12,width=1280,height=720,framerate=30/1 ! "
    "videoconvert ! "
    "video/x-raw,format=BGR ! "
    "appsink drop=true max-buffers=1 sync=false"
)

cap = cv2.VideoCapture(gst, cv2.CAP_GSTREAMER)
fgbg = cv2.createBackgroundSubtractorMOG2()
tracker = None

while True:
    ret, frame = cap.read()
    if not ret:
        continue

    # Preprocessing
    blurred = cv2.GaussianBlur(frame, (5, 5), 1.0)
    fgmask = fgbg.apply(blurred)
```

```
    kernel = np.ones((15, 15), np.uint8)
    mask = cv2.morphologyEx(fgmask, cv2.MORPH_CLOSE, kernel)
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, np.ones((5, 5), np.uint8))

    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Get one box that covers all movement
    all_x, all_y, all_x2, all_y2 = None, None, None, None
    for cnt in contours:
        if cv2.contourArea(cnt) > 1500:
            x, y, w, h = cv2.boundingRect(cnt)
            if all_x is None:
                all_x, all_y = x, y
                all_x2, all_y2 = x + w, y + h
            else:
                all_x = min(all_x, x)
                all_y = min(all_y, y)
                all_x2 = max(all_x2, x + w)
                all_y2 = max(all_y2, y + h)

    if all_x is not None:
        x, y, w, h = all_x, all_y, all_x2 - all_x, all_y2 - all_y

        if tracker is None:
            tracker = KalmanBox(x, y, w, h)

        sx, sy, sw, sh = tracker.update(x, y, w, h)

        # Draw smoothed box on both windows
        cv2.rectangle(frame, (sx, sy), (sx + sw, sy + sh), (0, 255, 0), 2)
        cv2.putText(frame, f"x:{sx} y:{sy}", (sx, max(15, sy - 10)),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)

        cv2.rectangle(mask, (sx, sy), (sx + sw, sy + sh), (255), 2)
        cv2.putText(mask, f"x:{sx} y:{sy}", (sx, max(15, sy - 10)),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255), 1)

    # Show both full windows
    cv2.imshow("Camera Frame", frame)
    cv2.imshow("Motion Mask", mask)

    if cv2.waitKey(1) & 0xFF == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

Appendix F – Object Detection Using FOMO Report

Report Title:

Object detection using FOMO (faster objects, more objects)

Report Objectives:

- Understanding how to use FOMO in object detection
- Implementing FOMO on a collected dataset

Executive Summary:

In this report we aim to cover what FOMO is and what's are the steps needed to implement it for our project for object detection. We also go through the documentation of gathering the dataset and testing it. The accuracy was 0%, the possible issues are studied. (1) the data set collected is hard for the model to learn from, the cars and persons were relatively close. (2) it could be tied to the issue we faced initially, where we were unable to connect our camera directly to our edge impulse account due to its inability to detect our camera even though the camera was connected to our Raspberry Pi.

Work Contribution:

This task was implemented and documented by Engy Farouq

Introduction

Object detection takes an image as input and returns information on the class and number of objects, as well as their position, size, and other characteristics, whereas image classification takes an image as input and outputs what kind of object is in the image. But compared to object classification models, object detection models make more complicated decisions.

They are also frequently larger (in terms of parameters) and need more data to train, which makes them challenge to implement on microcontrollers. The objective of FOMO's design was to combine the best features of object detection, such as location and object count, with the processing capacity needed for basic image classification.

FOMO uses the same architecture as object classification but cuts off the last layers of a standard image classification model and replaces this layer with a per-region class probability map. It then has a custom loss function which forces the network to fully preserve the locality in the final layer. This essentially gives you a heatmap of where the objects are. The FOMO (Fast Object Detection Model) framework, compatible with MobileNetV2, enables efficient real-time object detection using up to 30x less processing power and memory than models like MobileNet SSD or YOLOv5.

It works by dividing the image into a grid, with each cell detecting a specific object class (e.g., screw, nail, bolt, background). However, this can cause issues when objects are too close together, as their centroids may fall within the same cell, but increasing the image resolution can help. The system runs efficiently on low-power devices, processing a 64x64 pixel image in under a second at 80MHz, with the potential for higher frame rates if tensor acceleration is available.

Methodology

This section contains the steps needed to train and test our model to use FOMO object detection.

1-First of all, we need to install Edge Impulse and its dependencies on the Raspberry Pi.

2- Run Edge Impulse.

3-Login to your account, choose a project and select a microphone and camera to connect to the project.

4- Import a dataset into an Edge Impulse account- We chose to collect our own data set with our camera so it will train the model based on our camera specifications. In order to collect our dataset, we connected our raspberry pi to our laptop by using SSH to be able to move around freely. As seen in figure one our dataset consisted of 23 images. After importing the dataset, we need to label the images; at this stage it was only needed to be able to detect and differentiate between a person and a car.

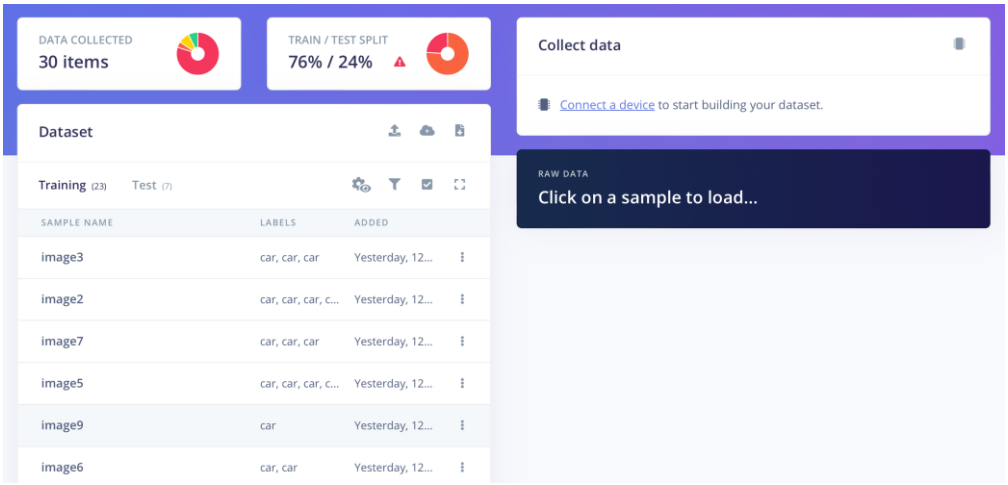


Figure 1. Imported and labeled dataset

5-Build the model-In order to build the model we have to create an impulse (get raw data and normalize it).

- Create impulse> Add a processing block>Image>Add
- Add a learning block>object detection (Images)>Add
- Save impulse

Note: make sure to select fit shortest axis in the image data block.

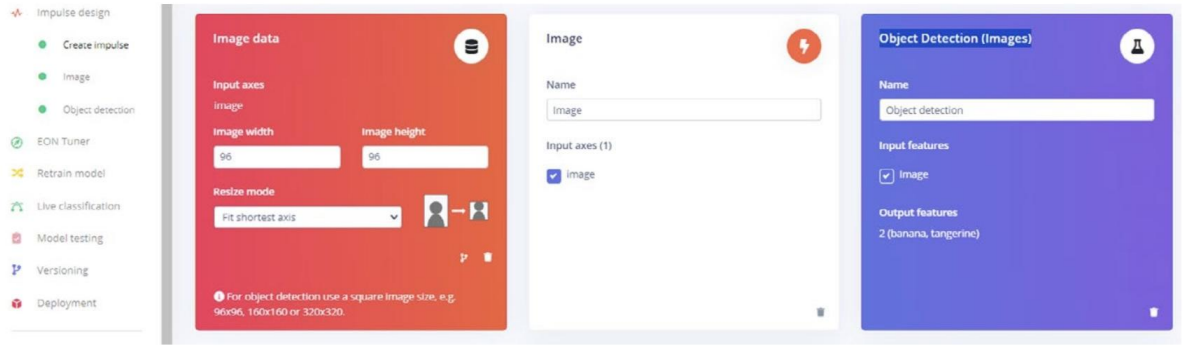


Figure 2. Create impulse window

- Generate features
- Impulse design> Images> save parameters> Generate features

This shows you how well your data separates.

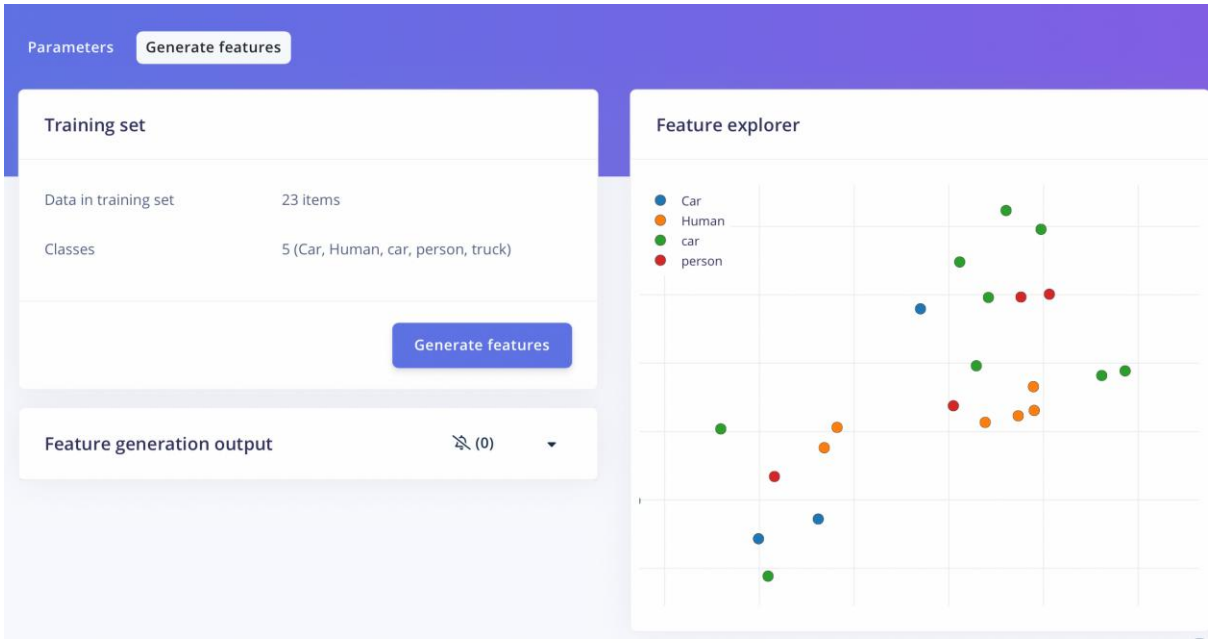


Figure 3. Feature explorer graph

6-Train the model- this includes teaching a machine learning algorithm to make predictions or decisions by feeding it **data**.

- Impulse design>Object detection>Start training

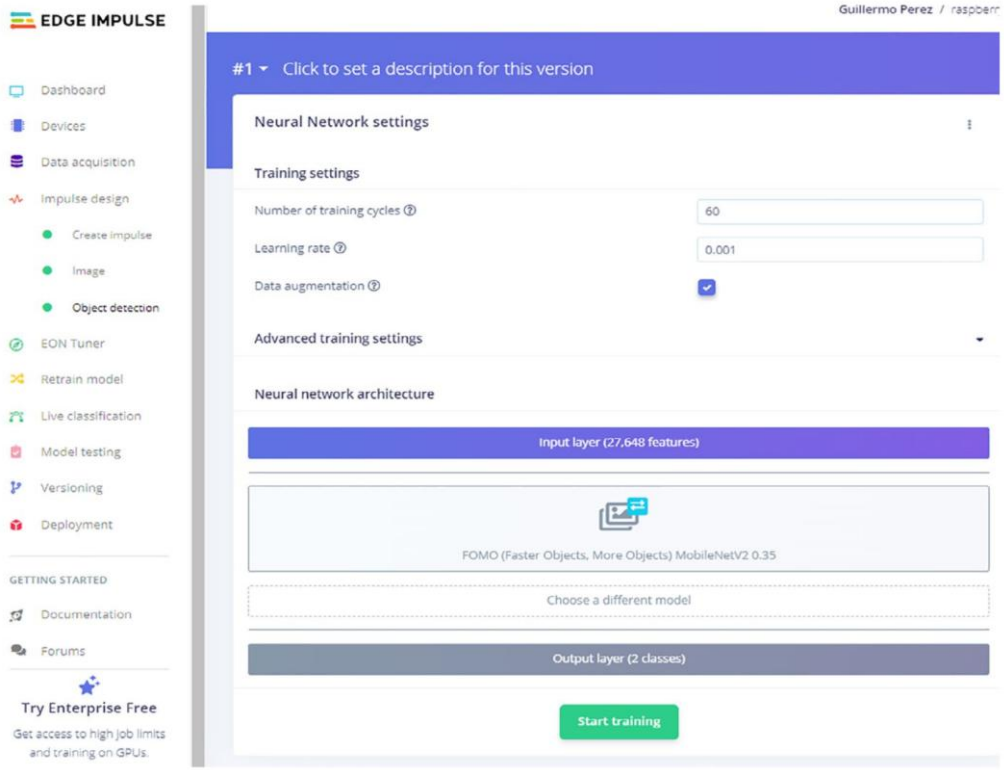


Figure 4: Training settings

7-Testing the model- This step helps validate whether the model works well. At this stage we took 7 pictures to test the system; Hoewever, as seen in figure ,the system was unable to detect any object and we got the accuracy of 0%.

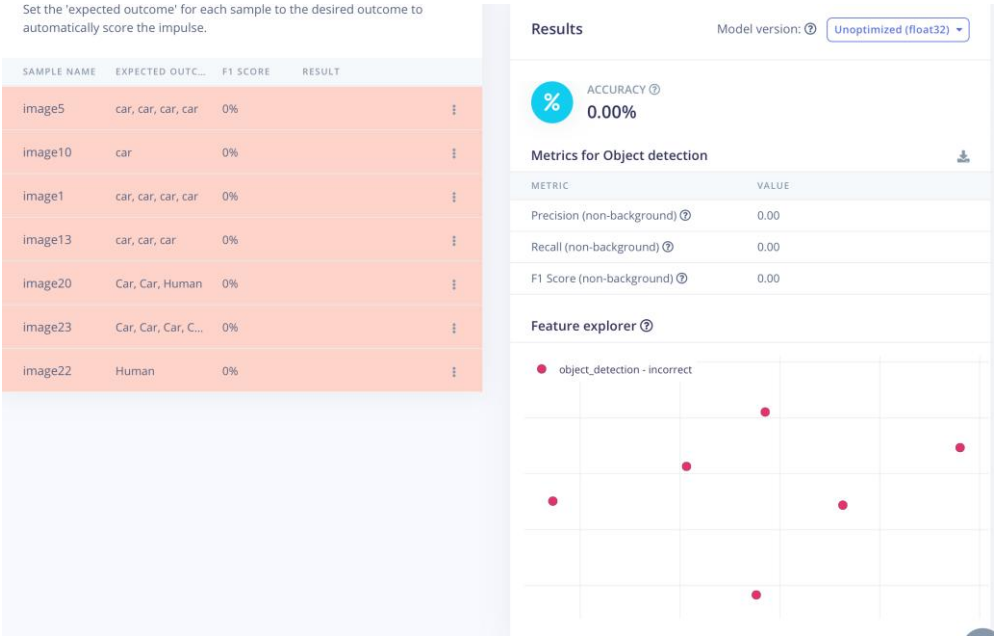


Figure 5: Testing results

8-Typically, the next step would be running the model on our device which can be done by the following steps.

- go to the Terminal window and enter the command: edge-impulse-linux-runner
- Use the URL from the terminal window and paste that into a browser to get the results seen in figure 6

Figure 6: Expected results for live object detection

Unfortunately, since we got the accuracy of 0%we were where unable to perform this step

Difficulties Faced

We were unable to connect our camera directly to our edge impulse account due to its inability to detect our camera even though the camera was connected to our raspberry pi. Furthermore, collecting the dataset also proved difficult due to time concerns and our inability to see the pictures we were taking until we are back to using the monitor.

Future Steps

Try to collect a new data set where the cars are not overlapping and retrain the model to find the new accuracy of the model.

References

[1]. “FOMO: Object Detection for Constrained Devices - Edge Impulse Documentation.” *Edgeimpulse.com*, 2023, docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/object-detection/fomo-object-detection-for-constrained-devices.

[2]. Guillen, Guillermo. *Sensor Projects with Raspberry Pi*. Apress, 17 Dec. 2019.

[3]. Edge Impulse. “Build Your Own Object Detection System with Machine Learning.” *YouTube*, 12 Apr. 2021, www.youtube.com/watch?v=dY3OSiJyne0&list=TLPQMjMwMjIwMjV1SNrmaoAuww&index=3. Accessed 23 Feb. 2025.

[4]. “FOMO: Object Detection for Constrained Devices - Edge Impulse. Documentation.” *Edgeimpulse.com*, 2023, docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/object-detection/fomo-object-detection-for-constrained-devices.

[5]. Edgeimpulse. “Linux-Sdk-Python/Examples/Image/Classify-Video.py at Master Edgeimpulse/Linux-Sdk-Python.” *GitHub*, 2021, github.com/edgeimpulse/linux-sdk-python/blob/master/examples/image/classify-video.py. Accessed 20 Mar. 2025.

Appendix

Code to install Edge impulse:

```
sudo apt update
sudo apt install python3-pip
pip3 install edge-impulse-linux
```

Code to run the model:

```
edge-impulse-linux-runner
```

Code to capture images:

```
libcamera-jpeg -o image.jpg
```

Appendix G – Object Detection Using YOLO Report

Report Title:

TPU, YOLOv11, and Object Detection on the Raspberry Pi and the HQ – IR Cut Camera

Report Objectives:

- To study the YOLO object detection method and the need for the TPU
- To simulate the method using a testing video while using the coral edge tpu

Task Description and Executive Summary

In this report, we study the YOLO object detection method and the need for the TPU when using it on the Raspberry Pi. We detail the steps required for the method simulation on a testing video. The usage of the method on the live camera and the potential issues are being studied.

Work Contribution:

Hind AlBastaki 100060327

Introduction

YOLOvxn which stands for You Only Look Once version x nano is an object detecting algorithm where an image is divided by grids of the matrix (SxS), where they are then processed, and each object is outline by a bounding box.

Furthermore the ‘n’ part stands for the model size and capacity with ‘n’ referring to nano which is the smallest and fastest model, with other letters indicating to other sizes such as ‘l’ for large which is a bigger model but has a higher accuracy than nano for example.

We previously worked on YOLOv8 and have had great results but were inclined to delve into a more advanced YOLO model which is the YOLO 11 as it has better accuracy, is easier to use and interface, in addition it also has a more optimize scalability and multi-tasking versatility [1].

In this project we will be Utilizing the Ultralytics library which includes the YOLO models including YOLOv11, as well as the COCO model which will be used in data segmentation, this library has an exponentially great results in tracking, instance segmentation, object detection, pose estimation and image classification utilizing AI, giving us an quick and easy results with high accuracy [2].

Furthermore, we will be delving into the YOLOv11n model due it’s compatibility with the Coral edge TPU, and it’s utilization of the computing power assisted for faster results than just relying on the raspberry pi’s CPU, the coral edge TPU will be using the TensorFlow Lite libraries to interface it with our models, the speed of our TPU will aid our progress drastically as it is able to preform 4×10^{12} operations per second [3].

Further research

Coral Edge TPU

The importance of coral edge TPU lies in the powerful computation capabilities offered by the TPU that helps in deep learning and neural networks by speeding up the tenso calculations significantly [4].

The process of deep learning includes a nueral network where the nueral nodes are connected in a way that is called the topology of the network, where each node of this network consists of three components which are 1-multiplier (synapse), 2-adder, 3-activation function, each

input signal goes through the three stages giving us the output signal with the given formula $y_i = \emptyset \sum_{i=1}^{n} w_{ij} \cdot x_i$that can also be viewed in figure [1] [4].

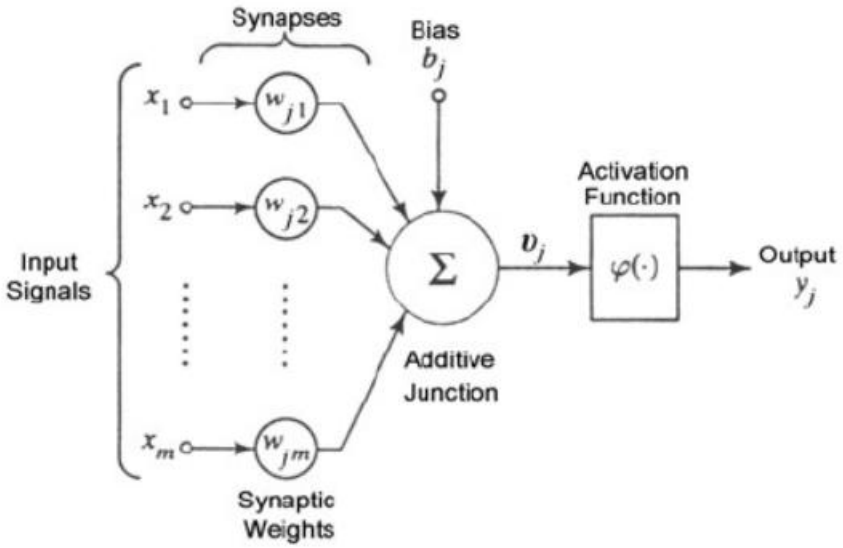


Figure 1 One Node Components

This multiplication is one of millions done in deep learning, which takes significant time, one solution to deal with this is to use the TPU, which was specifically designed to aid with this process in which the compiler transfers the TensorFlow Lite file into partitioned packages that are more suitable for the TPU, the TPU the accelerates the multiply-accumulate operations mentioned above , using Efficient Net[4].

Requirements

In order to be able to use our data in the process of data fusion we must produce the following parameters:

Object Detected

Each object detected falls into a certain class within the COCO Library, in which we must show this class in the simulated video and print it the result as an output, in which we printed it in the command box.

Object Bounding Box

The object bounding box should be displayed in the simulated video, where the bounding box will help us localize and identify each object in each frame.

Object Centroid

The centroid is related to the bounding box in terms of helping us locating the object in our frame, where the coordinates of the centroid will be utilized and printed out as an out put as we'll see in the results.

Confidence

The score of the confidence indicates how confident the model is in the object it is detecting and the score ranges from 0 which is the lack of confidence to 1 which is full confidence, this will help us understand the accuracy of our results, and gives the potential of setting a threshold for confidence to ignore the objects that the model is less confident in.[5]

Speed

Is the time each frame takes to process, while utilizing our coral edge TPU, which speeds up the frame processing significantly, we will also print the FPS rate on the simulated video's window.

Methodology

In the process of downloading the necessary file for YOLOv11 we will downloading libraries, models and python files through a sequence of commands and manual downloading.

Steps:

1. Download the zip file called "" or the contents found in the google document in this link … and extract it to it's designates location
2. Setup the required version of python "Python 3.9.12" found in appendix 1
3. Setup your file and virtual environment using the codes provided in appendix 2
4. Install the coral edge tpu requirements found in appendix 3
5. Reboot your raspberry pi
6. Continue the coral edge tpu setup by accessing you virtual environment and file then bashing in the commands found in appendix 4 in the command window
7. Reboot your raspberry pi
8. Then open the google colab link in appendix 5.1, then open a notebook as seen in appendix 5.2 and click on upload then browse then choose the file 'yolo11edgetpu.ipynb' and click on open as seen in appendix

9. Then execute the cells by pressing on the play button as in figure 5.3, there will be two cells
10. After successfully executing both of the cells make sure that you see the green tick on both of them as circled in the figure in appendix 5.4 then open the file then content then yolo file as seen in appendix 5.4 then choose the file that includes the name yolo11n_full_integer_quant_edgetpu.tflite and copy it's name.
11. Close the previous tabs and go to your main python file and right click on it and choose open with Thony
12. From the to tab click on run and choose 'Configure interpreter'
13. On the first scroll down menu choose Local Python 3, then on the second scroll down click on the three dots and choose you virtual environment then click 'ok'
14. Now from the top tab choose Tools then manage packages and search for 'cvzone' on PyPl and click on cvzone and install it
15. After completing the install close the tab and go back to your python file, go to the line that says in appendix 6.1 and change the name to the name of the file that we downloaded in step 10
16. Change the line in appendix 6.2 to include the path to the video that you want to simulate or 0 or 1 for your camera location then save the file
17. Go in your virtual environment and run your python file by hashing the code in appendix 7

Results

We can see that while utilizing the coral edge TPU with our YOLO model we noticed a faster frame processing time with an interval of around 40 ms , a preprocessing time of 1.3 ms, an interface time of 40 ms and a post processing time of 1.3 ms , and a frame processing rate of 61 FPS shown in the video window on the top left corner, all while maintaining a good confidence score of 0.859 for the highest confidence detected for the laptop and the lowest being for the bottle at 0.269 with a good detection for people with results being 0.768 and 0.733.

Other features were added for the centroid which are displayed on the screen being (490,430) for the laptop for example, which helps us identify the location of the object within the frame. Noting that the model adjusted for the size of the input video of 256x256.

The results indicate a successful utilization of the YOLO model and Interfacing of the coral edge TPU with our raspberry pi, while satisfying the necessary parameters which can be seen as printed outputs in the command window, or displayed directly on the simulated video.

Figure 3 Test video with results

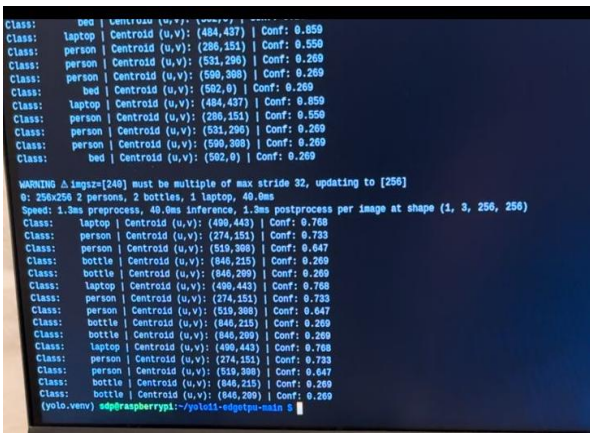


Figure 2 Output of the test video

Challenges

While running our model on the raspberry pi 5 with the bookworm Operating system, we faced multiple hurdles as the coral edge TPU was designed for the bulls eye operating system which forced us to download all of it's libraries in a virtual environment rather than system wide, which worked out but caused another problem which is with interfacing the camera as the libraries for the camera's whether they are 'picam' or 'libcam' had to be operated on system wide and when downloaded in the virtual environment did not operate properly as they needed other system wide libraries, which in turn drove us to download the system wide libraries inside our virtual environment which led our models to crash, making the interfacing between the new OS and camera with the old TPU not possible in our time frame, perhaps in the future, more precaution should be taken when deciding the components used.

References

[1] G. Jocher and J. Qiu, YOLO11 vs YOLOv8: Detailed Comparison, Ultralytics, 27 Sep. 2024. [Online]. Available: https://docs.ultralytics.com/compare/yolo11-vs-yolov8/#ultralytics-yolo11.

[2] Ultralytics, "Ultralytics YOLOv8," GitHub repository, https://github.com/ultralytics/ultralytics (accessed Apr. 22, 2025).

[3] Google Coral, "Edge TPU Benchmarks," *Coral.ai*, https://coral.ai/docs/edgetpu/benchmarks/ (accessed Apr. 23, 2025).

[4] Q-engineering, "Google Coral Edge TPU explained in depth," *Q-engineering*, https://qengineering.eu/google-corals-tpu-explained.html (accessed Apr. 23, 2025).

[5] Ultralytics, "YOLOv11: A foundational model for real-time object detection," 2024. [Online]. Available: https://docs.ultralytics.com/models/yolov11/

Appendices:

Appendix A

1	sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED	this line will give us access to install and upgrade python to the needed version freely, while not modifying the system Python packages tp prevent the system from crashing
2	pip3 install –upgrade thonny	Installs/upgrades Thonny
3	sudo apt-get update	Makes the latest list of available package updated in the local package database
4	sudo apt-get upgrade	Upgrades all of the installed packages obtaind by the updated list
5	curl https://pyenv.run | bash	Downloads and runs pyenv which is used to make be able to download multiple pyhton versions and switch between them
6	echo ‘export PATH=”$HOME/.pyenv/bin:$PATH”’ >> ~/.bashrc echo ‘eval “$(pyenv init –path)”’ >> ~/.bashrc echo ‘eval “$(pyenv virtualenv-init -)”’ >> ~/.bashrc exec “$SHELL”	Makes pynev recognized by configuring our shell
7	sudo apt-get install –yes libssl-dev zlib1g-dev libbz2-dev libreadline-dev libsqlite3-dev llvm libncurses5-dev libncursesw5-dev xz-utils tk-dev libgdbm-dev lzma lzma-dev tcl-dev libxml2-dev libxmlsec1-dev libffi-dev liblzma-dev wget curl make build-essential openssl	Makes python be able to be compiled from source by installing all required build dependencies
8	pyenv install 3.9.12	Installs and compiles python 3.9.12 using pyenv
9	pyenv local 3.9.12	Activates python version 3.9.12 for the time being
10	python –version	Tells us the python version to make sure our process was correct

Appendix B

1	python3 -m venv yolo.venv	Creates a virtual environment named “yolo.venv”
2	source yolo.venv/bin/activate	Activates our “yolo.venv” virtual enviroment
3	cd file_name	Access our file downloaded in the first step

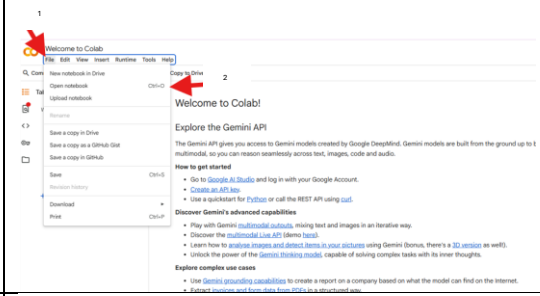
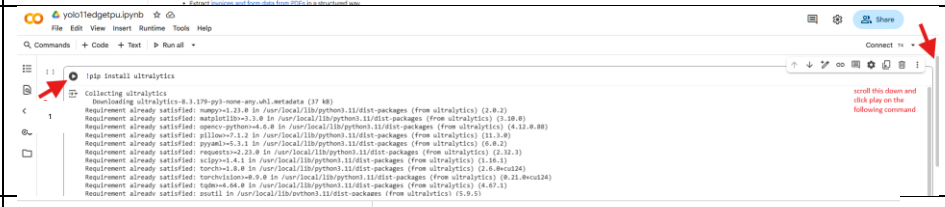
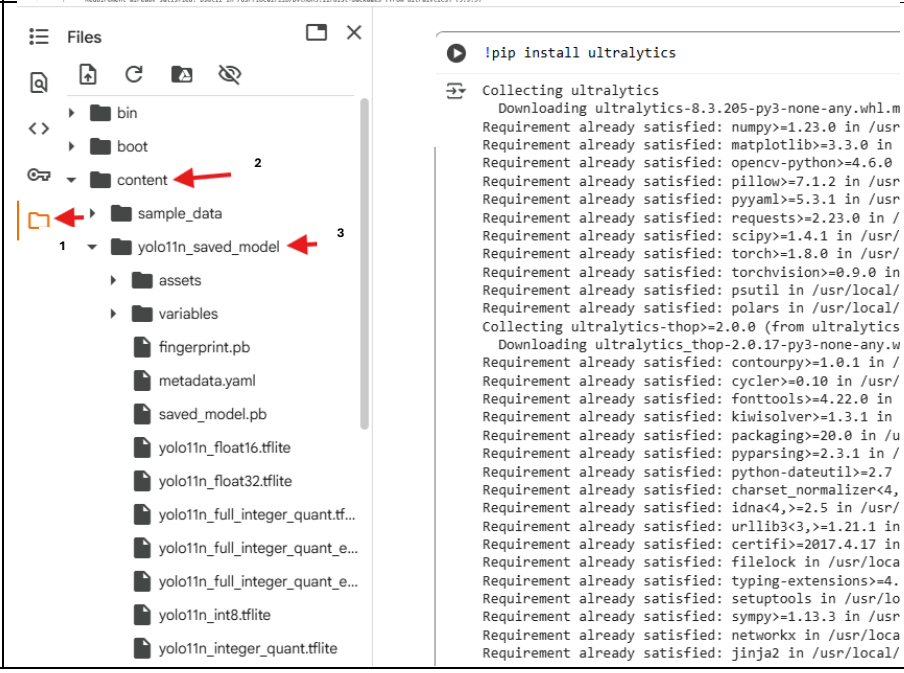
Appendix C

1	pip install edge-tpu-silva	Installs the Silva TPU Python package
2	pip install numpy==1.26.4	Ensures compatibility with the edge-tpu-silva package and TFLite by install this certain package of NymPy which is the 1.26.4 version

Appendix D

1	silvatpu-linux-setup	Configure our system for the ege-tpu by Running the Silva TPU setup script

Appendix E

1	https://colab.research.google.com/	
2		
3		
4		

Appendix F

1	model = YOLO (‘yolo11n_full_integer_quant_edgetpu.tflite’)	
2	cap = cv2.VideoCapture(‘video path’) or (0)	

Appendix G

1	source yolo.venv/bin/activate	
2	cd file_name	
3	python3 python_file_name.py	

Appendix H – Multithreading Python Code

```
# Generated using the assistance of an AI model

import threading, time, csv
from pathlib import Path
from picamera2 import Picamera2
from radar_alone import serialConfig, parseConfigFile, readAndParseData14xx
from picamera2.encoders import H264Encoder
from picamera2.outputs import FileOutput
import time, csv

# ===== Paths / Config =====
LOG_DIR    = Path.home() / "Desktop"
CAMERA_CSV = LOG_DIR / "camera_log.csv"
RADAR_CSV  = LOG_DIR / "radar_log.csv"
VIDEO_FILE = LOG_DIR / "camera_video.h264"
stop_event = threading.Event()
RADAR_CFG  = "/home/Shayma/Downloads/Config.cfg"   # <-- your confirmed cfg path
CAMERA_FPS = 30.0  # camera rate (rows/sec)
HEADER_CAM = ["source","frame_id","t_frame","t_processed","u","v"]
HEADER_RAD = ["source","frame_id","t_frame","t_processed","obj_id","x","y","z","vr"]

stop_event = threading.Event()

def now_epoch() -> float:
    return time.time()

def sleep_until(target_s: float):
    while True:
        dt = target_s - time.monotonic()
        if dt <= 0:
            break
        time.sleep(min(dt, 0.002))
# ================== Camera @ 30 FPS (continuous) ==================
def camera_thread():
    LOG_DIR.mkdir(parents=True, exist_ok=True)
    cam = Picamera2()
    try:
        cam.configure(cam.create_video_configuration(main={"size": (640, 480)}))
    except Exception:
        pass
    cam.start()
    time.sleep(0.5)
    # Start video recording
    cam.start_recording(FileOutput(str(VIDEO_FILE)), encoder=H264Encoder())
    print(f"[Camera] Recording video to {VIDEO_FILE}")
    with open(CAMERA_CSV, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(HEADER_CAM)
        f.flush()
        period = 1.0 / CAMERA_FPS
        now = time.monotonic()
```

```
        t0 = now + (period - (now % period))
        frame_id = 0
        while not stop_event.is_set():
            target = t0 + frame_id * period
            sleep_until(target)
            try:
                frame = cam.capture_array()
            except Exception as e:
                print("[Camera][ERR] capture:", e)
                time.sleep(0.01)
                continue
            t_frame = now_epoch()
            h, wpx = frame.shape[:2]
            u, v = wpx // 2, h // 2  # placeholder coordinates
            w.writerow(["camera", frame_id, t_frame, "", u, v])
            if frame_id % 30 == 0:
                print(f"[Camera] Frame {frame_id} logged")
            f.flush()
            frame_id += 1
    # Stop recording and camera
    try:
        cam.stop_recording()
        print("[Camera] Video recording stopped")
    except Exception:
        pass
    try:
        cam.stop()
    except Exception:
        pass
    print("[Camera] Thread stopped")
# ============== Radar (~10 FPS fresh packets only; continuous) ==============
def radar_thread():
    LOG_DIR.mkdir(parents=True, exist_ok=True)

    try:
        CLIport, Dataport = serialConfig(RADAR_CFG)
        params = parseConfigFile(RADAR_CFG)
        print("[Radar] serial/config OK")
    except Exception as e:
        print("[Radar][FATAL] init failed:", e)
        stop_event.set()
        return
    with open(RADAR_CSV, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(HEADER_RAD); f.flush()
        last_seq = None
        last_sig = None
        frame_id = 0
        while not stop_event.is_set():
            # Poll radar; only log when a NEW packet arrives
            try:
                dataOK, _, detObj = readAndParseData14xx(Dataport, params)
            except Exception as e:
                # transient read/parse issue; keep polling
                time.sleep(0.01)
```

```
    continue
if not dataOK:
    # no new packet yet
    time.sleep(0.003)
    continue
# Try to get a sequence/frame number from detObj (common keys)
this_seq = None
for k in ("frameNum", "frameNumber", "frame", "seq", "sequence"):
    if isinstance(detObj, dict) and k in detObj:
        this_seq = detObj[k]
        break
# Fallback: content signature (num objects + sums) to detect change
this_sig = None
if this_seq is None:
    try:
        n  = detObj.get("numObj", 0) or 0
        sx = sum(detObj.get("x", [])[:n]) if n else 0.0
        sy = sum(detObj.get("y", [])[:n]) if n else 0.0
        sz = sum(detObj.get("z", [])[:n]) if n else 0.0
        sv = sum(detObj.get("v", [])[:n]) if n else 0.0
        this_sig = (n, round(sx,4), round(sy,4), round(sz,4), round(sv,4))
    except Exception:
        # if we can't compute a signature, treat this as fresh
        this_sig = (time.time(),)
# Deduplicate: only log when sequence or signature changes
is_fresh = False
if this_seq is not None:
    is_fresh = (this_seq != last_seq)
else:
    is_fresh = (this_sig != last_sig)

if not is_fresh:
    # Same packet as last log â†' skip (prevents repeats)
    time.sleep(0.001)
    continue
# Mark as seen and log exactly once
if this_seq is not None:
    last_seq = this_seq
else:
    last_sig = this_sig
t_frame = now_epoch()
t_proc  = now_epoch()

n = detObj.get("numObj", 0) or 0
if n > 0:
    for i in range(n):
        w.writerow([
            "radar", frame_id, t_frame, t_proc, i,
            detObj["x"][i], detObj["y"][i], detObj["z"][i], detObj["v"][i]
        ])
    print(f"[Radar] Frame {frame_id}: {n} objects")
else:
    # fresh packet but no detections: still create one row
    w.writerow(["radar", frame_id, t_frame, t_proc, "", "", "", "", ""])
    print(f"[Radar] Frame {frame_id}: no detections")
```

```
            f.flush()
            frame_id += 1
    # graceful shutdown
    try:
        CLIport.write(('sensorStop\n').encode())
    except Exception:
        pass
    for p in (CLIport, Dataport):
        try: p.close()
        except Exception: pass
    print("[Radar] stopped.")
# ============================ Main ============================
if __name__ == "__main__":
    LOG_DIR.mkdir(parents=True, exist_ok=True)
    cam_t = threading.Thread(target=camera_thread, daemon=True)
    rad_t = threading.Thread(target=radar_thread,  daemon=True)
    cam_t.start(); rad_t.start()
    try:
        while cam_t.is_alive() or rad_t.is_alive():
            time.sleep(0.1)
    except KeyboardInterrupt:
        print("\n[INFO] Ctrl+C received, stopping...")
    finally:
        stop_event.set()
        # allow threads to exit cleanly
        time.sleep(0.2)
```

Appendix I – MATLAB Code to Manually Select the (u, v) pixel points

```
% The code for UV pixel selection was developed with assistance from  %ChatGPT
(OpenAI, 2023).
clc;clear;

image= "camn2.jpg";

% set the number of reflectors or targets in the image

num_reflector = input("Number of Reflectors: ");

while(num_reflector == [])

    % wait

end

imshow(image ); % displays the image

for k = 1:num_reflector

    [x, y] = ginput(1); % manually select the target
    x= round(x); y = round(y);
    hold on
    xi(k) = x;
    yi(k) = y;

    plot(xi(k), yi(k), 'x', 'MarkerSize', 10, 'LineWidth', 1);

    str = sprintf('u = %d\nv = %d', xi(k), yi(k));

    % Adds a text box with the u, and v values
    text(xi(k) + 50, yi(k) -40, str, ...
        'Color', 'white', ...
        'FontSize', 10, ...
        'BackgroundColor', [3/255, 102/2555, 252/255], ...
        'EdgeColor', 'white', ...
        'Margin', 4, ...
        'FontWeight', 'bold');
    hold on
    if (k == 1)
    fprintf("The pixel value corresponding to the %d-st point is: (%d, %d) \n", k,
xi(k), yi(k))

    elseif (k ==2)
        fprintf("The pixel value corresponding to the %d-nd point is: (%d, %d)
\n", k, xi(k), yi(k))

    elseif (k == 3)
        fprintf("The pixel value corresponding to the %d-rd point is: (%d, %d)
\n", k, xi(k), yi(k))

    else
        fprintf("The pixel value corresponding to the %d-th point is : (%d3, %d)
\n", k, xi(k), yi(k))
    end
end
```

Appendix J – Fog Intensity Classification Training Python Code

```
# ============================================
# Generated using the assistance of an AI model
# Fog Intensity Classification (BINARY VERSION)
# Dense Fog vs No Fog Only
# ============================================

import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from joblib import dump, load
from scipy.io import savemat

from sklearn.model_selection import (
    train_test_split, StratifiedKFold, RandomizedSearchCV,
    learning_curve
)
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import FunctionTransformer, MinMaxScaler
from sklearn.feature_selection import VarianceThreshold
from sklearn.ensemble import RandomForestClassifier

# --- Fix encoding for Windows ---
try:
    sys.stdout.reconfigure(encoding="utf-8", errors="replace")
except:
    pass

# ============================================
# 1) CONFIGURATION
# ============================================
FILES = {
    "No Fog":
r"C:\Users\User\Desktop\python\foggy_cityscapes\foggy_cityscapes\all_data\No_F
og_features.xlsx",
    "Dense":  r"C:\Users\User\Desktop\python\foggy_cityscapes\foggy_cityscapes
\all_data\Dense_Fog_features.xlsx"
}

LABEL_COL = "fog_label"
RANDOM_STATE = 42
TEST_SIZE = 0.10
VAL_SIZE = 0.20
CORR_THRESHOLD = 0.95
SAVE_MODEL_AS = "fog_rf_binary_deoverfit.joblib"
SAVE_MATLAB_AS = "fog_rf_binary_model.mat"
FAST_MODE = True
# ============================================
# 2) LOAD DATA
# ============================================
```

```
def load_and_label_data(files_map):
    parts = []
    for label, path in files_map.items():
        print(f"Loading data: {path}")
        df = pd.read_excel(path, engine="openpyxl")
        df[LABEL_COL] = label
        print(f" -> {label}: {df.shape[0]} rows")
        parts.append(df)
    df_all = pd.concat(parts, ignore_index=True)
    print(f"\nTOTAL loaded: {df_all.shape[0]} rows")
    return df_all
# ================================================
# 3) PREPARE FEATURES
# ================================================
def prepare_X_y(df):
    y = df[LABEL_COL].astype(str)
    X = df.drop(columns=[LABEL_COL], errors="ignore")
    X = X.select_dtypes(include=[np.number])
    return X, y
# ================================================
# 4) CORRELATION DROPPER
# ================================================
class CorrDropper(FunctionTransformer):
    def __init__(self, threshold=0.95):
        super().__init__(validate=False, feature_names_out="one-to-one")
        self.threshold = threshold
        self.keep_cols_ = None

    def fit(self, X, y=None):
        df = X if isinstance(X, pd.DataFrame) else pd.DataFrame(X)
        corr = df.corr().abs()
        upper = corr.where(np.triu(np.ones(corr.shape), k=1).astype(bool))
        drop_cols = [c for c in upper.columns if any(upper[c] >
self.threshold)]
        self.keep_cols_ = [c for c in df.columns if c not in drop_cols]
        return self

    def transform(self, X):
        df = X if isinstance(X, pd.DataFrame) else pd.DataFrame(X)
        return df[self.keep_cols_]
# ================================================
# 5) PIPELINE
# ================================================
def build_pipeline():
    return Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        ("low_var", VarianceThreshold(1e-5)),
        ("corr_dropper", CorrDropper(threshold=CORR_THRESHOLD)),
        ("rf", RandomForestClassifier(
            random_state=RANDOM_STATE,
            class_weight="balanced",
            max_depth=10,
            min_samples_leaf=2,
            max_features="sqrt",
            bootstrap=True,
            oob_score=True,
```

```
            max_samples=0.7
        ))
    ])
# ================================================
# 6) TRAIN + VALIDATION
# ================================================
from scipy.stats import randint, uniform

def train_and_validate(X_trainval, y_trainval):

    X_tr, X_val, y_tr, y_val = train_test_split(
        X_trainval, y_trainval, test_size=VAL_SIZE, random_state=RANDOM_STATE,
stratify=y_trainval
    )
    pipe = build_pipeline()

    param_dist = {
        "rf__n_estimators": randint(150, 400),
        "rf__max_depth": randint(7, 12),
        "rf__min_samples_leaf": randint(1, 4),
        "rf__min_samples_split": randint(2, 6),
        "rf__max_features": uniform(0.3, 0.5),
        "rf__max_samples": uniform(0.55, 0.3),
        "rf__ccp_alpha": uniform(0.0, 0.006)
    }
    search = RandomizedSearchCV(
        pipe, param_dist, n_iter=24,
        cv=StratifiedKFold(3, shuffle=True, random_state=RANDOM_STATE),
        scoring="accuracy",
        n_jobs=-1,
        verbose=1
    )
    search.fit(X_tr, y_tr)

    print("\nBest Parameters:", search.best_params_)

    best = search.best_estimator_

    preds = best.predict(X_val)
    print("\n--- VALIDATION RESULTS ---")
    print(classification_report(y_val, preds))
    print("Validation Accuracy:", accuracy_score(y_val, preds))
    print("Confusion Matrix:\n", confusion_matrix(y_val, preds))

    return best, (X_tr, y_tr, X_val, y_val)
# ================================================
# 7) TESTING
# ================================================
def test_model(model, X_test, y_test):
    preds = model.predict(X_test)
    print("\n--- TEST RESULTS (10%) ---")
    print(classification_report(y_test, preds))
    print("Test Accuracy:", accuracy_score(y_test, preds))
    print("Confusion Matrix:\n", confusion_matrix(y_test, preds))
# ================================================
# 8) EXPORT MODEL TO MATLAB
```

```
# ================================================
def export_to_matlab(model, X_train):
    rf = model.named_steps["rf"]
    imputer = model.named_steps["imputer"]

    mat_dict = {
        "trees": rf.n_estimators,
        "max_depth": rf.max_depth,
        "min_samples_leaf": rf.min_samples_leaf,
        "min_samples_split": rf.min_samples_split,
        "max_features": rf.max_features,
        "median_values": imputer.statistics_,
        "feature_names": np.array(list(X_train.columns), dtype=object),
        "classes": np.array(rf.classes_, dtype=object)
    }
    savemat(SAVE_MATLAB_AS, mat_dict)
    print(f"\n MATLAB file saved as {SAVE_MATLAB_AS}")
# ================================================
# 9) MAIN
# ================================================
if __name__ == "__main__":

    df = load_and_label_data(FILES)
    X_all, y_all = prepare_X_y(df)

    X_trainval, X_test, y_trainval, y_test = train_test_split(
        X_all, y_all, test_size=TEST_SIZE,
        random_state=RANDOM_STATE, stratify=y_all
    )
    model, (X_tr, y_tr, X_val, y_val) = train_and_validate(X_trainval,
y_trainval)

    test_model(model, X_test, y_test)

    dump(model, SAVE_MODEL_AS)
    print(f"\n Model saved as {SAVE_MODEL_AS}")

    export_to_matlab(model, X_tr)
```

Appendix K – Radar-Camera Data Fusion MATLAB Code

Main MATLAB Code

```
clc;clear
% Loading the datasets
camerasynthetic= readmatrix('yolo_detections.xlsx');
radarsynthetic = readmatrix('radar_log.xlsx');
gt_data = readmatrix('camera_with_markers.xlsx');

%% Parametrs %%

Z_projection_YOLO = 0.019;

R_new = [    0.9994   -0.0043   -0.0341;
   -0.0342   -0.0068   -0.9994;
    0.0040    1.0000   -0.0069];

T_new = [-0.1509;    0.0360  ;  0.0705];

K_intrinsic = [4053.60854628705  -12.6845378485635   1979.89094280910;
                0   4046.29446186445    1577.19501797494;
                0   0      1 ];

cam_meas = camerasynthetic(:,10:11);

YOLO_W = 640;YOLO_L = 416;

pix_error = 54.316;

% Radar parametrs
range_res = 0.293; %(m)
rvel_res = 0.31;%(m/s)
azimuth_res = pi/6; %(rad)

% Acceleration std
std_ax = sqrt(2); std_ay = sqrt(2);

% tuned parametr, 7.815 is based on the chi square distribution (alpha = 0.05 % ,
n =3)
G_cam = 5.991;
G_rad = 7.815*(1+0.107);

% Kalman Filter Initilization
INIT_COV= 2*diag([4 4 5.65 5.65]);INIT_VELOCITY =0.6;

last_tracker_update = 0;del_threshold = 70;nStates = 4;
Tracks(1).states_priori = [];
Tracks(1).states_posteriori = [0.883295714855194;   3.95602703094482; -0.5; -
0.7];
Tracks(1).P_priori = [];
Tracks(1).P_posteriori = 2*eye(4);
Tracks(1).H = [];
Tracks(1).R = [];
Tracks(1).assigned_meas = [];
Tracks(1).status = 'active';

Tracks(1).deletion_increment =0;
```

```
Tracks(1).assignment_count = 0;

% History to save track's states every frame
maxSteps = size(radarsynthetic,1);
maxTracks = 50;
state_history = zeros(maxSteps, maxTracks, nStates);



%% Radar data procassing

% Determines indices corresponding to
% the statis radar measument, to be removed

det = size(radarsynthetic,1);
static_meas = [];
K_cam = [];

for y = 1:det
    if (radarsynthetic(y,9)==0)
        static_meas = [static_meas; y];
    end
end


% Keeps the frame count while replacing the static measurments with NaN
% such that they are no procassed

for k = 1:length(static_meas)

    radarsynthetic(static_meas(k),5) = NaN;
    radarsynthetic(static_meas(k),6) = NaN;
    radarsynthetic(static_meas(k),8) = NaN;
end


%%%%%%%%%%%%%%%%%%%%%%% Main Structure %%%%%%%%%%%%%%%%%%%%


% The function outputs the cordinate transformed YOLO dataset and the GT
% data set
[x_y,Rcam, gt_data] = Coordinate_Transformation_YOLO_GT(cam_meas,gt_data,
Z_projection_YOLO, K_intrinsic,YOLO_W,YOLO_L, R_new, T_new, pix_error);


it = 1;

% the function process and sorts the data in terms of arrival, flagging
% measuments as either to be combined (camera and radar), or radar-only, or
% camera-only in updating the EKF states estimations

[data_sorted] = sensor_data_process(camerasynthetic,radarsynthetic);


frame_set = unique(data_sorted(:,4));

for k = 1:length(frame_set)
```

```
    % 4th row lists the blocks of radar and/or camera data considered every EKF
predict / update iteration

    indices = find(data_sorted(:,4) == frame_set(k));
    flag = data_sorted(indices(1),5); % 3 for combine, 2 for cam, 1 for radar

    if (flag == 4)
        % if no measurment is recieved, increment the deletion count and contnue,
        % and the prediction and update steps are skipped no

        tracks_num = length(Tracks);
        for j = 1:tracks_num
            if (Tracks(j).deletion_increment >=del_threshold)
                Tracks(j).status = 'inactive';
            else
                Tracks(j).deletion_increment = Tracks(j).deletion_increment + 1;
                Tracks(j).status = 'partially inactive';
            end
        end
        continue
    else

        % this function updates the measument strust with the currently
        % considered radar and/or camera measuments for iteration k,
        % and the each measurment's noise covariance matrix for radar data

        [measurment, Rmat] = measument_struct_n_R_computing(data_sorted,indices,k,
range_res, rvel_res,azimuth_res, flag);

        % the function computes delta_t, F, Q, and computes the P_priori matrix
and
        % states_prriori vector
        [last_tracker_update,Tracks, xk_] = PredictEKF(Tracks,
indices,data_sorted,last_tracker_update, std_ax, std_ay);

        % calculates the jacobian of the H matrix, the
        % measuments recieved are x, y, and vr for radar, states
        % are x, y, vx, and vy, respictievly

        [Tracks] = Hrad(Tracks, measurment, k);
        Hcam = [1 0 0 0; 0 1 0 0];

        % measument prediction step as part of the target tracking logic

        [Tracks] = measPredict(Tracks,flag, Hcam);

        % radar cost funuction

        if (flag == 2) || (flag == 3)

            [Cost_function_rad, active_tracks, row1] =
RadCostFunc(Tracks,measurment, k, G_rad, Rmat, flag);

        end

        % camera cost funuction

        if (flag == 1) || (flag == 3)  % camera only flag or combine flag
```

```
            [Cost_function_cam, active_tracks2, row2] =
CameraCostFunc(Tracks,measurment, k, G_cam, Rcam, flag, Hcam);

        end

        % Tracks not assigned with empty ellipsoids are removed from the cost
function,
        % such that the GNN does not mistakenly assign it a measument outside its
ellipsoid


        if  (flag == 1)

            [Cost_function_cam,Tracks, active_tracks2] =
UnassignedTracksMangment(Cost_function_cam,Tracks, row2, active_tracks2);

        elseif (flag == 3)

            [Cost_function_cam,Tracks, active_tracks2] =
UnassignedTracksMangment(Cost_function_cam,Tracks, row2, active_tracks2);

            [Cost_function_rad,Tracks, active_tracks] =
UnassignedTracksMangment(Cost_function_rad,Tracks, row1, active_tracks);

        else

            [Cost_function_rad,Tracks, active_tracks] =
UnassignedTracksMangment(Cost_function_rad,Tracks, row1, active_tracks);

        end


% Hungarian Algorithm and Assignment Problem

            if (flag == 3)

                [Tracks, matching_rad, matching_cam] =
CombinedMeasToTrackAssignment(Tracks,active_tracks, measurment, k,Hcam, Rcam,
Cost_function_cam, Cost_function_rad, Rmat);

            elseif (flag == 1)

                [Tracks, matching_cam] =
CamMeasToTrackAssignment(Cost_function_cam,active_tracks2, Tracks,measurment, k,
Rcam, Hcam);
            else

                 [Tracks, matching_rad] =
RadarMeasToTrackAssignment(Cost_function_rad, Tracks,Rmat, k, measurment,
active_tracks);
            end


        % intilization for unassigned measuments

        if (flag == 3)

           [Tracks] = CameraUnassignedMeasInitialization(Tracks,measurment,
matching_cam, k,INIT_VELOCITY, INIT_COV, Rcam, Hcam);
```

```
            [Tracks] = CameraUnassignedMeasInitialization(Tracks,measurment,
matching_cam, k,INIT_VELOCITY, INIT_COV, Rcam, Hcam);

        elseif (flag == 1)

            [Tracks] = CameraUnassignedMeasInitialization(Tracks,measurment,
matching_cam, k,INIT_VELOCITY, INIT_COV, Rcam, Hcam);

        else
            [Tracks] = CameraUnassignedMeasInitialization(Tracks,measurment,
matching_cam, k,INIT_VELOCITY, INIT_COV, Rcam, Hcam);

        end

        %% Kalman Filter Update Module

        nTracks = length(Tracks);

        if (data_sorted(indices(1), 10) == 0)

            [Tracks,state_history] = EKFUpdate(data_sorted, Tracks,state_history,
k);
        else
            [state_history,Tracks] = OOSM_EKF_Update(Tracks, Hcam, state_history,
xk_ , k);
        end
        Rmat = []; to_be_deleted=[];
    end
end
%% === Plotting: this module was developed using the assistance of AI ===

video_filename = 'tracker_radar_with_YOLO.mp4';
trackLineWidthBase = 1.5;
trackLineWidthMax  = 4;
fadeWindow         = 15;
fps                = 15;
min_assignments    = 30;
xyLim              = [-1 6];

%% === Load Data ===
radar_data = readmatrix('radar_log.xlsx');
frames     = unique(radar_data(:,2));
xCol = 6; yCol = 7; zCol = 8; vrCol = 9;

% Assuming gt_data is in workspace with columns:
%  col3 = timestamp, col4 = ID, col5 = x, col6 = y
% YOLO video
yoloVid = VideoReader('yolo_output.mp4');

%% === Preprocess GT (per-ID) ===
% Extract GT columns
gt_times_all = gt_data(:,3);
gt_ids_all   = gt_data(:,4);
gt_x_all     = gt_data(:,5);
gt_y_all     = gt_data(:,6);

unique_gt_ids = unique(gt_ids_all);
num_gt_ids = numel(unique_gt_ids);
```

```
% Build per-id cell arrays with ABSOLUTE timestamps (do NOT normalize)
gt_id_time = cell(num_gt_ids,1);
gt_id_x    = cell(num_gt_ids,1);
gt_id_y    = cell(num_gt_ids,1);
gt_start_times = zeros(num_gt_ids,1);
gt_end_times = zeros(num_gt_ids,1);

for i=1:num_gt_ids
    id = unique_gt_ids(i);
    mask = gt_ids_all == id;
    % sort by absolute time for safety
    [t_sorted, idxs] = sort(gt_times_all(mask));
    gt_id_time{i} = t_sorted;           % keep absolute times
    gtx = gt_x_all(mask);
    gty = gt_y_all(mask);
    gt_id_x{i} = gtx(idxs);
    gt_id_y{i} = gty(idxs);
    % record start/end in same absolute time base
    if ~isempty(t_sorted)
        gt_start_times(i) = t_sorted(1);
        gt_end_times(i)   = t_sorted(end);
    else
        gt_start_times(i) = Inf;
        gt_end_times(i)   = -Inf;
    end
end

%% === Setup figure and axes ===
fig_width = 1500; fig_height = 1000;
fig = figure('Color','w','Position',[100 100 fig_width fig_height]);
tiledlayout(2,3,'Padding','compact','TileSpacing','compact');

axFusion  = nexttile(1);
axRadarXY = nexttile(2);
axRadarRV = nexttile(3);
axYOLO    = nexttile(4);
axRMSE    = nexttile(5);
axGT      = nexttile(6);

%% === Video Writer ===
v = VideoWriter(video_filename, 'MPEG-4');
v.FrameRate = fps;
open(v);

% Variables from tracker logic
num_steps = size(state_history,1);
T_final   = length(Tracks);
rmse_values = nan(num_steps,1);

%% === Fix all axes initial settings ===
xlim(axFusion,xyLim); ylim(axFusion,xyLim); axis(axFusion,'equal');
xlim(axRadarXY,xyLim); ylim(axRadarXY,xyLim); axis(axRadarXY,'equal');
xlim(axRMSE,[1 num_steps]); ylim(axRMSE,[0 2]); % initial RMSE limits

%% === YOLO Frame Count & timing ===
numYoloFrames = floor(yoloVid.Duration * yoloVid.FrameRate);
yolo_time = (0:numYoloFrames-1)/yoloVid.FrameRate;
```

```
%% === Time vectors and master end time (shortest of radar, GT, YOLO) ===
radar_time = (frames - 1)/fps;  % radar timestamps in seconds

% Use max GT end time among IDs (absolute)
gt_max_time = max(gt_end_times);

% determine earliest final time among datasets (Q1 = C)
end_time = min([radar_time(end), gt_max_time, yolo_time(end)]);

% Build time_vector from radar_time but clipped to end_time (radar driving)
time_vector = radar_time(radar_time <= end_time);
if isempty(time_vector)
    error('No overlapping time region between radar, GT, and YOLO. Check
timestamps.');
end
nSteps = min(num_steps, numel(time_vector));

%% === Base colors for tracks and GT IDs ===
base_colors = lines(T_final);
gt_colors = lines(max(num_gt_ids,3)); % ensure at least a few colors

%% === Main animation loop (radar-driven, stops at shortest dataset end_time) ===
for tidx = 1:nSteps
    current_time = time_vector(tidx);

    % --- Current GT positions per ID (interpolated to current_time) ---
    gt_curr_pos = nan(num_gt_ids,2); % rows: [x, y]
    for i=1:num_gt_ids
        tvec = gt_id_time{i};
        if isempty(tvec)
            continue;
        end

        % Only define a current position if the GT has started
        if current_time < gt_start_times(i)
            % not started yet -> leave NaN
            gt_curr_pos(i,:) = [NaN, NaN];
        else
            % if within ID's recorded range -> interpolate
            if current_time <= gt_end_times(i)
                gx = interp1(tvec, gt_id_x{i}, current_time, 'linear');
                gy = interp1(tvec, gt_id_y{i}, current_time, 'linear');
                gt_curr_pos(i,:) = [gx, gy];
            else
                % after ID's last time -> use last known position (keep it
visible)
                gt_curr_pos(i,:) = [gt_id_x{i}(end), gt_id_y{i}(end)];
            end
        end
    end

    %% --- Fusion / Tracker subplot ---
    cla(axFusion); hold(axFusion,'on');

    rmse_sum = 0; rmse_count = 0;

    for p = 1:T_final
        if ~isfield(Tracks(p),'assignment_count') || Tracks(p).assignment_count <
min_assignments
```

```
            continue;
        end

        % Extract states up to this radar-driven step
        xs = squeeze(state_history(1:tidx,p,1));
        ys = squeeze(state_history(1:tidx,p,2));

        validIdx = find(~isnan(xs) & ~isnan(ys) & ~(xs==0 & ys==0));
        if numel(validIdx) < 2, continue; end

        % --- Fading trail ---
        fade_len = min(fadeWindow, numel(validIdx));
        idx_seg  = validIdx(end-fade_len+1:end);
        fade_alpha = linspace(0.2,1,fade_len);
        base_color = base_colors(p,:);

        for k = 1:fade_len-1
            seg_color = (1-fade_alpha(k))*[0.7 0.7 0.7] +
fade_alpha(k)*base_color;
            plot(axFusion, xs(idx_seg(k:k+1)), ys(idx_seg(k:k+1)), ...
                '-', 'Color', seg_color, ...
                'LineWidth', trackLineWidthBase +
fade_alpha(k)*(trackLineWidthMax-trackLineWidthBase));
        end

        % Current tracker point (use most recent valid)
        cur_x = xs(idx_seg(end));
        cur_y = ys(idx_seg(end));
        scatter(axFusion, cur_x, cur_y, 60, base_color, 'filled');

        % RMSE: distance to nearest available GT current point
        if ~all(isnan(gt_curr_pos(:)))
            dists = hypot(gt_curr_pos(:,1) - cur_x, gt_curr_pos(:,2) - cur_y);
            dists(isnan(dists)) = [];  % remove NaNs
            if ~isempty(dists)
                rmse_sum = rmse_sum + dists(1)^2; % nearest GT squared distance
                rmse_count = rmse_count + 1;
            end
        end
    end

    hold(axFusion,'off');
    xlim(axFusion, xyLim); ylim(axFusion, xyLim); axis(axFusion,'equal');
grid(axFusion,'on');
    title(axFusion,sprintf('Tracker Output [Time %.2f s (step %d/%d)]',
current_time, tidx, nSteps));

    %% --- Radar plots ---
    cla(axRadarXY); cla(axRadarRV);
    hold(axRadarXY,'on'); hold(axRadarRV,'on');

    pts_idx = max(1, min(tidx, numel(frames)));  % safe mapping to available
frames index
    points = radar_data(radar_data(:,2)==frames(pts_idx),:);
    if ~isempty(points)
        x = points(:,xCol); y = points(:,yCol); vr = points(:,vrCol);
        rng = sqrt(x.^2 + y.^2 + points(:,zCol).^2);

        scatter(axRadarXY, x, y, 50, vr, 'filled');
```

```
        xlabel(axRadarXY,'X [m]'); ylabel(axRadarXY,'Y [m]');
grid(axRadarXY,'on');
        title(axRadarXY,sprintf('Radar Top-Down (Frame %d)',frames(pts_idx)));

        scatter(axRadarRV, rng, vr, 50, 'filled');
        xlabel(axRadarRV,'Range [m]'); ylabel(axRadarRV,'Radial Velocity [m/s]');
grid(axRadarRV,'on');
        title(axRadarRV,'Range vs Velocity');

        % keep RV axis stable
        xlim(axRadarRV, [0 max(rng)*1.1]);
    end
    hold(axRadarXY,'off'); hold(axRadarRV,'off');
    xlim(axRadarXY, xyLim); ylim(axRadarXY, xyLim);

    %% --- YOLO subplot ---
    cla(axYOLO);
    % Compute YOLO frame index safely and clamp
    yolo_idx = round(current_time * yoloVid.FrameRate) + 1;
    yolo_idx = max(1, min(yolo_idx, numYoloFrames));
    yoloFrame = read(yoloVid, yolo_idx);
    imshow(yoloFrame, 'Parent', axYOLO);
    title(axYOLO, sprintf('YOLO Detection (frame %d/%d)', yolo_idx,
numYoloFrames));

    %% --- RMSE subplot ---
    if rmse_count > 0
        rmse_values(tidx) = sqrt(rmse_sum / rmse_count);
    else
        rmse_values(tidx) = NaN;
    end

    cla(axRMSE); hold(axRMSE,'on');
    plot(axRMSE, 1:tidx, rmse_values(1:tidx), 'r-', 'LineWidth',1.5);
    scatter(axRMSE, tidx, rmse_values(tidx), 40, 'r', 'filled');
    xlabel(axRMSE,'Step'); ylabel(axRMSE,'RMSE'); grid(axRMSE,'on');
    xlim(axRMSE,[1 nSteps]);
    rmse_max = max(rmse_values(1:tidx), [], 'omitnan');
    if isempty(rmse_max) || isnan(rmse_max) || rmse_max==0
        rmse_max = 1;
    end
    ylim(axRMSE, [0 rmse_max*1.1]);
    title(axRMSE,'RMSE vs Time');
    hold(axRMSE,'off');

    %% --- GT subplot (per-ID colors) ---
    cla(axGT); hold(axGT,'on');
    for i=1:num_gt_ids
        tvec = gt_id_time{i};
        if isempty(tvec)
            continue;
        end

        % Only plot history if this GT has started
        if current_time >= gt_start_times(i)
            past_mask = tvec <= current_time;
            if any(past_mask)
                xhist = gt_id_x{i}(past_mask);
                yhist = gt_id_y{i}(past_mask);
```

```
                plot(axGT, xhist, yhist, '--', 'LineWidth', 1.2, 'Color',
gt_colors(i,:));
            end
        end

        % Plot current point if defined (we set NaN if not started)
        if ~isnan(gt_curr_pos(i,1))
            scatter(axGT, gt_curr_pos(i,1), gt_curr_pos(i,2), 60, gt_colors(i,:),
'filled');
            text(axGT, gt_curr_pos(i,1), gt_curr_pos(i,2), sprintf(' ID %d',
unique_gt_ids(i)), 'VerticalAlignment','bottom');
        end
    end
    xlabel(axGT,'GT X [m]'); ylabel(axGT,'GT Y [m]');
    title(axGT,'Ground Truth Paths (by ID)');
    xlim(axGT, xyLim); ylim(axGT, xyLim); axis(axGT,'equal'); grid(axGT,'on');
    hold(axGT,'off');

    %% --- Render and save frame ---
    drawnow;                        % allow event processing and ensure figure
updates
    frame = getframe(fig);
    writeVideo(v, frame);
end

%% --- Close video safely ---
close(v);
disp(['Video saved to ', video_filename]);
```

Functions

The Coordinate_Transformation_YOLO_GT Function

```
function [d_x_y,Rcam, gt_data] =
Coordinate_Transformation_YOLO_GT(cam_meas,gt_data, Z, K_intrinsic,YOLO_W,YOLO_L,
R_new, T_new, pix_error )

Ext = [R_new T_new]; % Extrnsic calibration matric [R|t]

P_projective = K_intrinsic*Ext; % Projective transformation matrix

% Cmaera image  resolution
Orig_W = 4056;Orig_L = 3040;

% Undoing the image resizing done during YOLO procassing
cam_meas(:,1) = cam_meas(:,1)*(Orig_W/YOLO_W);
cam_meas(:,2) = cam_meas(:,2)*(Orig_L/YOLO_L);



% H 3x3 homography matrix
H = [P_projective(:,1) P_projective(:,2) P_projective(:,3)*Z+P_projective(:,4) ];


% 3 x n ( number of rows in the YOLO dataset) matrix
pixels = [cam_meas(:,1)';
          cam_meas(:,2)';
          ones(1, size(cam_meas,1))];
```

```
% solving for the [kx ; ky;k] vector
x_y_w = H\pixels;
% normalizing the vector by dividing by the 3rd element k
x_y = (x_y_w(1:2,:) ./ x_y_w(3,:))';

% Estimating the camera noise covariance matrix using the extrnsic
% calibration reprojection error

d_uv = [pix_error; pix_error; 1];

dx_y_1 = inv(H)*d_uv;

d_x_y = dx_y_1(1:2,:)/dx_y_1(3,:);

Rcam = [d_x_y(1)^2 0 ; 0 d_x_y(2)^2];

% Coordinate transforming the ground truth using the rotation atrix R, and
% the translational vector T

% extracting the GT - x and y coordinate measurments in the camera coordinate
% frame
u_v_gt = gt_data(:, 5:7)';

% Obtaining the R camera -> radar matrix

Trc = -R_new*T_new ;

% Obtaining the T camera -> radar vector
Rrc = R_new';

% obtaining the GT measurments in the radar coordinate frame
x_y_gt_nor = (Rrc*u_v_gt + Trc)';

gt_data(:, 5:7) = x_y_gt_nor;

end
```

The sensor_data_process Function

```
function [data_sorted] = sensor_data_process(camerasynthetic,radarsynthetic)

    t_cam_meas_frame = camerasynthetic(:,2); % meas frame timestampe
    cam_pi_clock = camerasynthetic(:,3); % timestampe corresponding to when the
Rassberry Pi recieves the measurment, after the YOLO procassing
    cam_meas = camerasynthetic(:,10:11); % BBoxes centroids of the YOLO detections

    t_rad_meas_frame = radarsynthetic(:,3); % meas frame timestampe
    rad_meas = radarsynthetic(:,[6:7, 9]); % radar x, y, and vr measurments
    rad_pi_clock = radarsynthetic(:,4); % timestampe corresponding to when the
Rassberry Pi recieves the measurment, after the data parsing step
    rad_meas(:,1:2) = rad_meas(:,1:2);

    % to produce discrete frames timestamps
    FPS = 1000;
    radar_dt = 1/FPS;
    camera_dt = 1/FPS;
    t_rad_meas_frame = round(t_rad_meas_frame/radar_dt)*radar_dt;
    t_cam_meas_frame = round(t_cam_meas_frame/camera_dt)*camera_dt;
```

```
    % row 1 = meas frame, row 2 = frame meas timestamp after data procassing,
    % row 3 = sensor flag (0 for camera, 1 for radar), row 4 = procassed blocks
num, row 5
    % = EKF update flag, row 6 = x meas, row 7 = y meas, row 8 = vr meas
    data = [ t_rad_meas_frame rad_pi_clock ones(length(t_rad_meas_frame),1)
zeros(length(t_rad_meas_frame),1) zeros(length(t_rad_meas_frame),1) rad_meas;
        t_cam_meas_frame cam_pi_clock zeros(length(t_cam_meas_frame),1)
zeros(length(t_cam_meas_frame),1) zeros(length(t_cam_meas_frame),1) x_y
zeros(length(t_cam_meas_frame), 1)];

    data_sorted = sortrows(data, 2);

    block_num = 1;

    frame_groups = findgroups(data_sorted(:,1));

    % frame group column is 9, frame ID

    data_sorted(:,9) = frame_groups;
    % total number of unique frames stored for a given scene
    set = unique(frame_groups);
    static_target_indices =[];

    for w = 1:length(set)

        % determines the index corresponding to the first measurment data in
        % the data block considered for a given frame
        indeces_orig = find(data_sorted(:, 4) ==set(w));

        % checks if any of the measurments consdired is static (static meas
        % were replaced by nan at an earlier step)
        % the vector is all ones if all measurments are static
        check_static_meas = isnan(data_sorted(indeces_orig, 6));

        % num of meas recieved from camera or radar in a given frame
        num_recived_meas_per_fram = length(check_static_meas);

        % number of static measurments in that frame
        intermediate = sum(check_static_meas(:) == 1);

        % if the measurments recieved are not all static

        if ~(intermediate == num_recived_meas_per_fram)

            % extracts the moving_target_indicces

            indeces = indeces_orig(~check_static_meas);

            static_target_indices = indeces_orig(check_static_meas);

            % for a given frame, we record the earliest time (min) a measurment
was
            % 'recieved' by the rassberry pi for that frame, and the latest time
(max)
            max_pi_timestamp = max(data_sorted(indeces,2));

            min_pi_timestamp = min(data_sorted(indeces,2));
```

```
            % if the difference between arrival of the meas to the
            % Rassberry Pi is less that 0.05, and the measurments is of two
            % different sensors, the EKF update flag is 3, and the sensors
            % data are considered in one block ( updates data_sorted 4th row)

            if (max_pi_timestamp- min_pi_timestamp <0.05) &&
(length(unique(data_sorted(indeces,3)))>1)

                data_sorted(indeces,4) = block_num*ones(length(indeces),1);
                block_num = block_num+1;
                data_sorted(indeces,5) = 3*ones(length(indeces),1); % combine flag

            % else, if the difference between arrival of the meas to the
            % Rassberry Pi is less that 0.05, and the measurments is of one
            % sensor, the EKF update flag is either 1 or 2

            elseif (max_pi_timestamp- min_pi_timestamp <0.05) &&
(length(unique(data_sorted(indeces,3)))<2)

                data_sorted(indeces,4) = block_num*ones(length(indeces),1);
                block_num = block_num+1;

                % the sensor flag is used to determine the EKF Update flag

                if data_sorted(indeces(1),3) == 0
                    data_sorted(indeces,5) = 1*ones(length(indeces),1); % camera
flag
                else
                    data_sorted(indeces,5) = 2*ones(length(indeces),1); % radar
flag
                end
                % if the difference between arrival of the meas to the
                % Rassberry Pi is not less that 0.05, and the measurments is of
two
                % different sensors. This is when one sensor
                % provides data at a faster rate, while the other is
                % slower, leadeing to OOSM. OOSM measurment are flaged
                % after this step, here, block num of grouped data
                % incrments and the EKF update flag is determined based on
                % sensor type flag

            elseif ~(max_pi_timestamp- min_pi_timestamp <0.05) &&
(length(unique(data_sorted(indeces,3)))>1)

                % uses the sensor type flag to determine each sensor's meas
indices
                indeces0 = data_sorted(indeces, 3) == data_sorted(indeces(1), 3);
                sensor1_indeces = indeces(indeces0);

                sensor2_indeces = setdiff(indeces, sensor1_indeces);

                if (data_sorted( indeces(1), 3) == 0)

                    data_sorted(sensor1_indeces,5) =
1*ones(length(sensor1_indeces),1); % camera flag
                    data_sorted(sensor1_indeces,4) =
block_num*ones(length(sensor1_indeces),1);
                    block_num = block_num+1;
```

```
                    data_sorted(sensor2_indeces,5) =
2*ones(length(sensor2_indeces),1); % radar flag
                    data_sorted(sensor2_indeces,4) =
block_num*ones(length(sensor2_indeces),1);
                    block_num = block_num+1;

                else
                    data_sorted(sensor1_indeces,5) =
2*ones(length(sensor1_indeces),1); % radar flag
                    data_sorted(sensor1_indeces,4) =
block_num*ones(length(sensor1_indeces),1);
                    block_num = block_num+1;

                    data_sorted(sensor2_indeces,5) =
1*ones(length(sensor2_indeces),1); % camera flag
                    data_sorted(sensor2_indeces,4) =
block_num*ones(length(sensor2_indeces),1);
                    block_num = block_num+1;
                end
            end
        else
            % if the measument recived is static radar meas, the flag is
            % set as 4
            data_sorted(indeces_orig,5) = 4*ones(length(indeces_orig),1); % no
measument recieved
            data_sorted(indeces_orig,4) = block_num*ones(length(indeces_orig),1);
            block_num = block_num+1;

        end

        if ~isempty(static_target_indices)
            data_sorted(static_target_indices,:) = [];
        end
        static_target_indices =[];
    end

    % OOSM flagging , row 10
    frame_prev = data_sorted(1,9);
    data_sorted(:,10) = zeros(size(data_sorted,1),1);
    block_num_prev =  data_sorted(1,4);

    % when the frame of the meas previous to the currently checked frame is
    % greater than or equal to it, the meas is potentially an OOSM, if the
    % the meas does not have the same grouped meas block num, then the
    % signal is an OOSM. frame_prev and block_num_prev are ot updated

    for t = 2:size(data_sorted,1)
        current_check = data_sorted(t,9);

        block_num_curr =  data_sorted(t,4);

        if (current_check <= frame_prev)

            if (block_num_curr == block_num_prev)
                frame_prev = data_sorted(t, 9);
                block_num_prev =  data_sorted(t,4);
            else
                data_sorted(t, 10) = 1;
```

```
            end
        else
            frame_prev = data_sorted(t, 9);
            block_num_prev =  data_sorted(t,4);
        end
    end
end
```

The measument_struct_n_R_computing Function

```
function [measurment, Rmat] = measument_struct(data_sorted,indices,k, range_res,
rvel_res,azimuth_res, flag)

    if flag == 3

        indeces0 = data_sorted(indices, 3) == 1;% 1 encodes for radar meas.
        radar_indices = indices(indeces0);
        camera_indeces = setdiff(indices, radar_indices);

        measurment(k).y_rad_meas = data_sorted(radar_indices,6:8);

        % produces a 3D tensor, 3 x3 by the number of measurments

        Rmat = Measurement_Covariance(measurment, range_res, rvel_res,azimuth_res,
k);

        measurment(k).y_cam_meas = data_sorted(camera_indeces,6:7);

    elseif flag == 1

        measurment(k).y_rad_meas = [];
        measurment(k).y_cam_meas = data_sorted(indices,6:7);
        Rmat =[];

    elseif flag == 2

        measurment(k).y_rad_meas = data_sorted(indices,6:8);
        Rmat = Measurement_Covariance(measurment, range_res, rvel_res,azimuth_res,
k);
        measurment(k).y_cam_meas = [];
    end
end


function Rmat = Measurement_Covariance(measurment, range_res,
rvel_res,azimuth_res, k)

    y_rad_mea_vect = measurment(k).y_rad_meas;

    [rows, ~] = size(y_rad_mea_vect);


    % tensor 3 x 3 x by the number of radar measurments
    Rmat = zeros(3,3,rows);

   for q = 1:rows

       x = y_rad_mea_vect(q,1);
```

```
        y = y_rad_mea_vect(q,2);

        theta = atan2(y, x);
        range = sqrt(x^2 + y^2);

        var_x = (cos(theta))^2*(range_res^2/4) -
(range*sin(theta))^2*(azimuth_res^2/4);
        var_y = (sin(theta))^2*(range_res^2/4) +
(range*cos(theta))^2*(azimuth_res^2/4);
        doppler_res = (rvel_res^2)/4;

        Rmat(:,:,q) = [var_x 0 0; 0 var_y 0; 0 0 doppler_res];
    end
end
```

The Hrad Function

```
function [Tracks] = Hrad(Tracks, measurment, k)

    if ~isempty(measurment(k).y_rad_meas)
        for g = 1:length(Tracks)

            if strcmp(Tracks(g).status,'inactive')

                continue
            else

                x = Tracks(g).states_priori(1); y = Tracks(g).states_priori(2);
                vx = Tracks(g).states_priori(3); vy = Tracks(g).states_priori(4);
                r2 = x^2 + y^2; r = sqrt(r2);
                if r < 1e-2, r = 1e-2; r2 = r^2; end

                dvr_dx = ( y*(vx*y - vy*x) ) / (r2^(3/2));
                dvr_dy = ( x*(-vx*y + vy*x) ) / (r2^(3/2));
                dvr_dvx = x / r;
                dvr_dvy = y / r;

                Tracks(g).H = [];

                Tracks(g).H = [1 0 0 0;
                               0 1 0 0;
                               dvr_dx dvr_dy dvr_dvx dvr_dvy];

            end
        end
    end
end
```

The measPredict Function

```
function [Tracks] = measPredict(Tracks,flag, Hcam)

    for r = 1:length(Tracks)

        % skip the meas prediction step for inactive tracks
```

```
        if strcmp(Tracks(r).status,'inactive')
            continue
        else

            if (flag == 1) % camera
                Tracks(r).y_predicted = [];
                Tracks(r).H = Hcam;

                Tracks(r).y_predicted = Hcam*Tracks(r).states_priori;


            elseif (flag == 3) %combine

                Tracks(r).y_combined_predicted =[];
                Tracks(r).y_combined_predicted =  ([Tracks(r).H;
Hcam])*(Tracks(r).states_priori);

            else % radar
                Tracks(r).y_predicted =[];
                Tracks(r).y_predicted = (Tracks(r).H)*(Tracks(r).states_priori);

            end
        end
    end
end
```

The RadCostFunc Function

```
function [Cost_function_rad, active_tracks, row1] = RadCostFunc(Tracks,measurment,
k, G_rad, Rmat, flag)

    nTracks = length(Tracks);
    y_rad_vect = measurment(k).y_rad_meas;
    nMeas = size(y_rad_vect,1);
    % pre-allocates a cost function for active and partially active tracks
    % only
    Cost_function_rad = 1000 * ones( sum(~strcmp({Tracks.status},'inactive')),
nMeas );

    row1 = 0; active_tracks = [];
    for n = 1:nTracks

        if strcmp(Tracks(n).status, 'inactive')
            continue
        end
        row1 = row1+1;
        % saves the cost function row index and the corresponding track
        % index
        active_tracks =[active_tracks;row1 n];

        S_rad = Tracks(n).H * Tracks(n).P_priori * Tracks(n).H';
        for o = 1:nMeas
            S = S_rad + Rmat(:,:, o);
            if flag ==3
                y_pred= Tracks(n).y_combined_predicted(1:3);
            else
                y_pred= Tracks(n).y_predicted;
            end
            residual = measurment(k).y_rad_meas(o,:)' - y_pred;
```

```
            S = (S +S')/2;
            % Mahalanobis distance eq
            d2 = residual'*(S\residual);

            if d2 < G_rad
                Cost_function_rad(row1, o) = d2;
            end
        end
    end
end
```

The CameraCostFunc Function

```
function [Cost_function_cam, active_tracks2, row2] =
CameraCostFunc(Tracks,measurment, k, G_cam, Rcam, flag, Hcam)

    nTracks = length(Tracks);
    y_rad_vect = measurment(k).y_rad_meas;
    nMeas = size(y_rad_vect,1);
    % pre-allocates a cost function for active and partially active tracks
    % only
    Cost_function_cam = 1000 * ones( sum(~strcmp({Tracks.status},'inactive')),
nMeas );

    row2 = 0; active_tracks2 = [];
    for n = 1:nTracks

        if strcmp(Tracks(n).status, 'inactive')
            continue
        end
        row2 = row2+1;
        % saves the cost function row index and the corresponding track
        % index
        active_tracks2 =[active_tracks2;row2 n];

        S_rad = Hcam * Tracks(n).P_priori *Hcam';
        for o = 1:nMeas
            S = S_rad + Rcam;
            if flag ==3
                y_pred= Tracks(n).y_combined_predicted(1:3);
            else
                y_pred= Tracks(n).y_predicted;
            end
            residual = measurment(k).y_rad_meas(o,:)' - y_pred;
            S = (S +S')/2;
            % Mahalanobis distance eq
            d2 = residual'*(S\residual);

            if d2 < G_cam
                Cost_function_cam(row2, o) = d2;
            end
        end
    end
end
```

The UnassignedTracksMangment Function

```
function [Cost_function,Tracks, active_tracks] =
UnassignedTracksMangment(Cost_function,Tracks, row, active_tracks)

    [~, measuments ] = size(Cost_function);
    to_be_deleted = [];

    for a = 1:row
        Summ = sum(Cost_function(a, :)== 1000);
        track_index = active_tracks(a,2);
        if (Summ == measuments)

            if (Tracks(track_index).deletion_increment >=del_threshold)
                Tracks(track_index).status = 'inactive';
            else
                Tracks(track_index).status = 'partially inactive';
                Tracks(track_index).deletion_increment =
Tracks(track_index).deletion_increment +1;
            end
            to_be_deleted = [to_be_deleted;a];
        else
            Tracks(track_index).deletion_increment = 0;
        end
    end
    Cost_function(to_be_deleted,:) = []; active_tracks(to_be_deleted,:) = [];
end
```

The CombinedMeasToTrackAssignment Function

```
function [Tracks, matching_rad, matching_cam] =
CombinedMeasToTrackAssignment(Tracks,active_tracks, measurment, k,Hcam, Rcam,
Cost_function_cam, Cost_function_rad, Rmat)

    [matching_rad, ~] = matchpairs(Cost_function_rad, 10000);
    [matching_cam, ~] = matchpairs(Cost_function_cam, 10000);

    if ~isempty(matching_rad)

        for y = 1:length(matching_rad(:,1))

            rowIdx = matching_rad(y,1); % row in Cost_function_rad
            ind_track = active_tracks(rowIdx,2); % actual Tracks index

            meas1 = measurment(k).y_rad_meas(matching_rad(y,2),:);

            if ~isempty(matching_cam)

                indexxx= find(ind_track == matching_cam(:,1));

                if ~isempty(indexxx) % case 1, a radar measument is matched with a
cam measurment

                    meas2 = measurment(k).y_cam_meas(matching_cam(indexxx,2),:);

                    y_combined = [meas1(:); meas2(:)];
                    Tracks(ind_track).assigned_meas = [];
                    Tracks(ind_track).assigned_meas = y_combined;
                    Tracks(ind_track).sensor = 'combined';
```

```
                    if isempty(Tracks(ind_track).assignment_count)
                        Tracks(ind_track).assignment_count= 1;
                    else
                        Tracks(ind_track).assignment_count =
Tracks(ind_track).assignment_count + 1;
                    end
                    Tracks(ind_track).H = [Tracks(ind_track).H; Hcam];
                    Tracks(ind_track).Rsub = [Rmat(:,:,matching_rad(y, 2))
zeros(3,2) ; zeros(2,3) Rcam];
                    Tracks(ind_track).deletion_increment = 0;
                    Tracks(ind_track).status = 'active';

                else % one measument only (cam or radar) is matched to a track
                    if (Tracks(ind_track).deletion_increment >=del_threshold)
                        Tracks(ind_track).status = 'inactive';
                    else
                        Tracks(ind_track).deletion_increment =
Tracks(ind_track).deletion_increment + 1;

                        Tracks(ind_track).status = 'partially inactive';
                    end
                end
            end
        end
    end
end
```

The CamMeasToTrackAssignment Function

```
function [Tracks, matching_cam] =
CamMeasToTrackAssignment(Cost_function_cam,active_tracks2, Tracks,measurment, k,
Rcam, Hcam)

    [matching_cam, ~] = matchpairs(Cost_function_cam, 10000);

    if ~isempty(matching_cam)

        for y = 1:length(matching_cam(:, 1))

            row_ind = matching_cam(y,1);
            index = active_tracks2(row_ind,2);

            Tracks(index).sensor ='camera';

            Tracks(index).assigned_meas = [];
            Tracks(index).assigned_meas  =
measurment(k).y_cam_meas(matching_cam(y, 2),:);
            Tracks(index).Rsub = [];
            Tracks(index).Rsub = Rcam;
            Tracks(index).H = [];
            Tracks(index).H = Hcam;

            if isempty(Tracks(index).assignment_count)
                Tracks(index).assignment_count= 1;
            else
                Tracks(index).assignment_count = Tracks(index).assignment_count +
1;          end
```

```
            Tracks(index).status = 'active';

        end

        if ~isempty(Cost_function_cam)
            if (size(Cost_function_cam,1)> length(Cost_function_cam(1,:)))

                assigned_tracks = active_tracks(matching_cam(:,1),2);
                unassigned_tracks = setdiff(active_tracks,assigned_tracks);

                for b = 1:length(unassigned_tracks)

                    if (Tracks(unassigned_tracks(b)).deletion_increment
>=del_threshold)
                        Tracks(unassigned_tracks(b)).status = 'inactive';
                    else
                        Tracks(unassigned_tracks(b)).deletion_increment =
Tracks(unassigned_tracks(b)).deletion_increment + 1;

                        Tracks(unassigned_tracks(b)).status = 'partially
inactive';
                    end
                end
            end
        end
    end
end
```

The RadarMeasToTrackAssignment Function

```
function [Tracks, matching_rad] = RadarMeasToTrackAssignment(Cost_function_rad,
Tracks,Rmat, k, measurment, active_tracks)

   [matching_rad, ~] = matchpairs(Cost_function_rad, 10000);

   for y = 1:length(matching_rad(:, 1))

       row_ind = matching_rad(y,1);
       index = active_tracks(row_ind,2);
       Tracks(index).sensor ='radar';
       Tracks(index).assigned_meas = [];
       Tracks(index).assigned_meas = measurment(k).y_rad_meas(matching_rad(y,
2),:);
       if isempty(Tracks(index).assignment_count)
            Tracks(index).assignment_count= 1;
       else
            Tracks(index).assignment_count = Tracks(index).assignment_count + 1;
       end
       Tracks(index).Rsub = Rmat(:,:,matching_rad(y, 2));
       Tracks(index).status = 'active';
   end

   if ~isempty(Cost_function_rad)
       if (size(Cost_function_rad,1)> length(Cost_function_rad(1,:)))
           assigned_tracks = active_tracks(matching_rad(:,1),2);
           unassigned_tracks = setdiff(active_tracks,assigned_tracks);
```

```
            for b = 1:length(unassigned_tracks)
                if (Tracks(unassigned_tracks(b)).deletion_increment
>=del_threshold)
                    Tracks(unassigned_tracks(b)).status = 'inactive';
                else
                     Tracks(unassigned_tracks(b)).deletion_increment =
Tracks(unassigned_tracks(b)).deletion_increment + 1;
                     Tracks(unassigned_tracks(b)).status = 'partially inactive';
                end
            end
        end
    end
end
```

The RadarUnassignedMeasInitialization Function

```
function [Tracks] = RadarUnassignedMeasInitialization(measurment,k, Rmat,Tracks,
matching_rad, INIT_VELOCITY, INIT_COV)

    all_indices = 1:size(measurment(k).y_rad_meas,1);
    assigned_indices = matching_rad(:,2);
    unassigned_indices = setdiff(all_indices, assigned_indices);

    unassigned_rad = measurment(k).y_rad_meas(unassigned_indices,:);
    R_unassigned = Rmat(:,:,unassigned_indices);

    if ~isempty(unassigned_rad)

        curr_tracks = length(Tracks);

        for J = 1:size(unassigned_rad,1)

            Tracks(curr_tracks + J).sensor = 'radar';
            Tracks(curr_tracks + J).assigned_meas = [];
            Tracks(curr_tracks + J).assigned_meas = unassigned_rad(J, :);
            Tracks(curr_tracks + J).assignment_count =1;
            Tracks(curr_tracks + J).Rsub = R_unassigned(:,:,J);
            Tracks(curr_tracks + J).P_priori = [];
            Tracks(curr_tracks + J).P_priori =INIT_COV;

            vr_x = unassigned_rad(J, 1); vr_y = unassigned_rad(J, 2);
            vr_vx = INIT_VELOCITY; vr_vy = INIT_VELOCITY;

            Tracks(curr_tracks + J).H = [];
            Tracks(curr_tracks + J).H =[1 0 0 0; 0 1 0 0; vr_x vr_y vr_vx vr_vy];
            Tracks(curr_tracks + J).states_priori =[];
            Tracks(curr_tracks + J).states_priori =[vr_x vr_y vr_vx vr_vy];
            Tracks(curr_tracks + J).y_predicted = [];
            Tracks(curr_tracks + J).y_predicted =(Tracks(curr_tracks +
J).H)*(Tracks(curr_tracks + J).states_priori');
            Tracks(curr_tracks + J).deletion_increment = 0;
            Tracks(curr_tracks + J).status = 'active';

        end
    end
end
```

The CameraUnassignedMeasInitialization Function

```
function [Tracks] = CameraUnassignedMeasInitialization(Tracks,measurment,
matching_cam, k,INIT_VELOCITY, INIT_COV, Rcam, Hcam)

    all_indices = 1:size(measurment(k).y_cam_meas,1);
    assigned_indices = matching_cam(:,2);
    unassigned_indices = setdiff(all_indices, assigned_indices);
    unassigned_cam = measurment(k).y_cam_meas(unassigned_indices,:);

    if ~isempty(unassigned_indices)
        unassigned_cam = measurment(k).y_cam_meas(unassigned_indices,:);

        curr_tracks = length(Tracks);

        for J = 1:size(unassigned_cam,1)

            Tracks(curr_tracks + J).sensor ='camera';

            Tracks(curr_tracks + J).assigned_meas = [];
            Tracks(curr_tracks + J).assigned_meas = unassigned_cam(J, :);
            Tracks(curr_tracks + J).assignment_count =1;
            Tracks(curr_tracks + J).Rsub = Rcam;
            Tracks(curr_tracks + J).P_priori = [];
            Tracks(curr_tracks + J).P_priori =INIT_COV;
            vr_x = unassigned_cam(J, 1); vr_y = unassigned_cam(J, 2);
            vr_vx = INIT_VELOCITY; vr_vy = INIT_VELOCITY;

            Tracks(curr_tracks + J).H = [];
            Tracks(curr_tracks + J).H =Hcam;
            Tracks(curr_tracks + J).states_priori =[];
            Tracks(curr_tracks + J).states_priori =[vr_x vr_y vr_vx vr_vy];
            Tracks(curr_tracks + J).y_predicted = [];
            Tracks(curr_tracks + J).y_predicted =(Tracks(curr_tracks +
J).H)*(Tracks(curr_tracks + J).states_priori');
            Tracks(curr_tracks + J).deletion_increment = 0;
            Tracks(curr_tracks + J).status = 'active';
        end
    end
end
```

The EKFUpdate Function

```
function [Tracks,state_history] = EKFUpdate(Tracks,state_history, k)

    nTracks = length(Tracks);

    for p = 1:nTracks

        if strcmp(Tracks(p).status,'inactive') ||
strcmp(Tracks(p).status,'partially inactive')
            continue
        else
            if strcmp(Tracks(p).sensor,'combined')

                K = (Tracks(p).P_priori * Tracks(p).H') /(Tracks(p).H *
Tracks(p).P_priori * Tracks(p).H' + Tracks(p).Rsub);
                Tracks(p).P_posteriori = [];
```

```
                Tracks(p).P_posteriori = (eye(nStates) -
K*(Tracks(p).H))*Tracks(p).P_priori*(eye(nStates) -
K*(Tracks(p).H))'+K*Tracks(p).Rsub*K';
                Tracks(p).states_posteriori = [];
                Tracks(p).states_priori = Tracks(p).states_priori(:);
                Tracks(p).states_posteriori = Tracks(p).states_priori +
K*(Tracks(p).assigned_meas - Tracks(p).y_combined_predicted);

                state_history(k, p, :) = Tracks(p).states_posteriori(:);

            elseif strcmp(Tracks(p).sensor,'radar')
||strcmp(Tracks(p).sensor,'camera')


                K = Tracks(p).P_priori * Tracks(p).H' /(Tracks(p).H *
Tracks(p).P_priori * Tracks(p).H' + Tracks(p).Rsub);

                if strcmp(Tracks(p).sensor,'radar')
                    K_rad = K;
                else
                    K_cam = K;
                end

                Tracks(p).P_posteriori = [];
                Tracks(p).P_posteriori = (eye(nStates) -
K*(Tracks(p).H))*Tracks(p).P_priori*(eye(nStates) -
K*(Tracks(p).H))'+K*Tracks(p).Rsub*K';
                Tracks(p).states_posteriori = [];
                Tracks(p).states_priori = Tracks(p).states_priori(:);
                Tracks(p).states_posteriori = Tracks(p).states_priori +
K*(Tracks(p).assigned_meas' - Tracks(p).y_predicted);

                state_history(k, p, :) = Tracks(p).states_posteriori(:);
            end
        end
    end
end
```

The OOSM_EKF_Update Function

```
function [state_history,Tracks] = OOSM_EKF_Update(Tracks, Hcam, state_history, xk_
, k)
    for p = 1:nTracks

        if strcmp(Tracks(p).status,'inactive') ||
strcmp(Tracks(p).status,'partially inactive')
            continue
        else

            if strcmp(Tracks(p).sensor,'combined')

                K = Tracks(p).P_priori * Tracks(p).H' /(Tracks(p).H *
Tracks(p).P_priori * Tracks(p).H' + Tracks(p).Rsub);
                Tracks(p).P_posteriori = (eye(nStates) -
K*(Tracks(p).H))*Tracks(p).P_priori*(eye(nStates) -
K*(Tracks(p).H))'+K*Tracks(p).Rsub*K';
                Tracks(p).states_posteriori = [];
                Tracks(p).states_priori = Tracks(p).states_priori(:);
```

```
                Tracks(p).states_posteriori = xk_ + K*(Tracks(p).assigned_meas -
Tracks(p).y_combined_predicted);

                state_history(k, p, :) = Tracks(p).states_posteriori(:);

            elseif strcmp(Tracks(p).sensor,'radar')
||strcmp(Tracks(p).sensor,'camera')

                if strcmp(Tracks(p).sensor,'radar')
                    K = K_rad;

                else
                    if ~isempty(K_cam)

                        K= K_cam;
                    else
                        Tracks(p).H = Hcam;
                        K = Tracks(p).P_priori * Tracks(p).H' /(Tracks(p).H *
Tracks(p).P_priori * Tracks(p).H' + Tracks(p).Rsub);

                    end

                end

                Tracks(p).P_posteriori = (eye(nStates) -
K*(Tracks(p).H))*Tracks(p).P_priori*(eye(nStates) - K*(Tracks(p).H))'+
K*Tracks(p).Rsub*K';
                Tracks(p).states_priori = Tracks(p).states_priori(:);
                Tracks(p).states_posteriori = xk_ + K*(Tracks(p).assigned_meas' -
Tracks(p).y_predicted);

                state_history(k, p, :) = Tracks(p).states_posteriori(:);
            end
        end
    end
end
```

Appendix L – YOLO Implementation Code on a Laptop

Generated using the assistance of an AI model

```
import cv2
import pandas as pd
import cvzone
from ultralytics import YOLO
# Load video and CSV
cap = cv2.VideoCapture('camera_video3.mp4')
csv_df = pd.read_csv("yolo_detections3.csv")
# COCO class names
class_list = YOLO('yolo11n.pt').names
# Video writer
output_width, output_height = 640, 416
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
out = cv2.VideoWriter("yolo_output3.mp4", fourcc, 30, (output_width, output_height))
frame_count = 0
while True:
    ret, frame = cap.read()
    if not ret:
        break

    frame = cv2.resize(frame, (output_width, output_height))
    frame_count += 1

    # Get detections for this frame
    frame_dets = csv_df[csv_df.frame_id == frame_count]

    for _, det in frame_dets.iterrows():
        x1, y1, x2, y2 = int(det['x1']), int(det['y1']), int(det['x2']), int(det['y2'])
        cls_id = int(det['class_id'])
        conf = det['confidence']
        label = class_list[cls_id] if cls_id < len(class_list) else str(cls_id)
        # Draw bounding box + center + label
        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cx, cy = (x1+x2)//2, (y1+y2)//2
        cv2.circle(frame, (cx, cy), 3, (255, 255, 255), -1)
        cvzone.putTextRect(frame, f'{label} {conf:.2f}', (x1, max(0, y1-10)), 1, 1)
    out.write(frame)
cap.release()
out.release()
print("YOLO output video saved: yolo_output3.mp4")
```

Appendix M – Component Selection and Budget Summary Report

Report Title:

Comparison and Selection of Camera and Radar Modules, Component Identification, and Budget Summary

Report Objectives:

- To compare multiple camera and radar modules and select the one that best suits our project.
- To identify the components needed for the fourth stage of the senior design project (refer to the WBS, linked in the Links to Relevant Documents section).
- To generate a matrix with the components, their costs, links to their sellers, and totals costs.

Executive Summary:

In this report, we analysed and compared several radar and camera modules to determine which modules best suit our application. After careful consideration, we propose using the **TI IWR6853AOPEVM Radar Module** and the **HQ IR-Cut Camera Module**. The reasons for these selections are detailed in the report. We also listed the components required at this stage of the project and calculated the total costs, which is **3668.53 AED**.

Work Contribution:

Hind Al Bastaki	• Identified and compiled a list of candidate radar and camera modules. • Conducted extensive research to gather feature data for various radar and camera modules. • Researched and identified the necessary components for the project. • Prepared a summary budget table. • Conducted the resolutions calculation table.
Shayma Alteneiji	• Evaluated and compared radar and camera modules to finalize the selection. • Authored the final report, summarizing all the key findings.

Task Description

In this report, we compare the features of multiple radar and camera modules to select the ones that best suit our application. We also summarize the components needed for the fourth stage of the project (refer to the WBS, linked in the Links to Relevant Documents section) and provide their costs along with links to the sellers' websites.

Methodology

To complete this task, we reviewed multiple research papers on radar-camera data fusion. We listed the radar and camera modules used in each paper, then searched for their datasheets (collected in the Data Sheets folder under Important Documents, linked in the Links to Relevant Documents section). We compared and contrasted these features to make our selection. Additionally, we listed the required components.

Radar Module Selection

Based on the described approach, we generated a radar candidate module list of four modules, as shown in Table 1. To make the selection, we analysed the following features of the radar modules:

(1) The range, maximum bearing, and Doppler resolutions

Range, maximum bearing, and Doppler resolution are crucial parameters in our application [1]. The more accurate the results are, the easier and the greater is the accuracy of the fused data. We included tuning frequency and TX/RX data to calculate these values, as outlined in Equations 1-3 [1]. Table 2 calculates these measurements for the four modules. PRI is the total chirp time of a single frequency-modulated signal [1] and can be configured by the user. For table 2, PRI is set to 300 µs, and Nd is set to 128 chirps per frame (Although these values may not be optimal for our application, they were chosen to compare resolution across the modules.). Additionally, we included the beamwidth angle column to contrast the coverage area and resolution trad-offs of the modules [2].

(2) Maximum unambiguous range

We have opted for the radar to be selected to have a detection range between 0.5 and 30 m, which is commonly used for advanced driver-assistance system (ADAS) applications [3]. We therefore added the 'maximum unambiguous range' column.

(3) Ease of use

We also considered the ease of interfacing the module with the Raspberry Pi and the computational load it requires on the processing unit. To address this, we added the output data type and interface type columns. Outputting raw data would require more computational load on the Raspberry Pi and more time debugging. Additionally, we added the AOP check column, as using USB antennas would levitate the challenge of the project and would require experienced engineers' assistance in wiring and aligning the antennas.

(4) Price and power supply requirement

Finally, power supply requirements and price are important metrics in the selection process, as they influence integration and project budget.

$$\Delta\rho = \frac{c_o}{2B} \quad (1)$$

$$\Delta\theta = \frac{c_o}{f_c \times d \times N_{TX} \times N_{RX}} \quad (2)$$

$$\Delta v_d = \frac{c_o}{2f_c \times PRI \times N_d} \quad (3)$$

Where,

c_o	: the speed of electromagnetic waves	N_{TX}	: number of transmitters (TX)
B	: the bandwidth of the radar chirp signal	N_{RX}	: number of receivers (RX)
f_c	: centre frequency of linear frequency modulated signal	d	: receiver element spacing, typically carrier signal wavelength (λ)/2
PRI	: the pulse repetition interval	N_d	: number of chirps per radar frame

Table 1. Radar Modules Features

Model Name	Maximum Unambi-guous Range	Tuning Frequ-ency (GHz)	Supply Voltage	Power Cons-umpti-on	TX/ RX	Beamwidth Angle (Horizontal Coverage x Vertical Coverage)	Output Data Type	Antenna on Package (AOP)	Interface Types	Price (AED)
Continental ARS 408-21 [4]	0.2 m – 250 m, depending on the beam steering angle	76 – 77	12 V DC	12 V DC/ 24 V DC	4/12	Horizontal coverage: 2.2° - 17° depending on the beam steering angle	Speed, range, and angle between two objects in sight	Yes	CAN bus interface	~1,500
TI AWR1642 [5]	25 m	76 – 81	5V, 2A	-	4/2	(120° x 30°)	Raw ADC data	No	CAN, CAN-FD, I2C, QSPI, SPI, UART	~1,100
TI IWR6843 [6]		60 – 64		-	4/3		Raw ADC and point cloud data	Yes		~1,100
Omnipresense OPS243-C [7]	1m-60m	24 - 24.25	5V	1.5W	1/1	(20° x 24°)	Speed, direction, and Range of objects in sight	Yes	USB, UART, and RS-232	~900

Table 2. Range, Maximum Bearing, and Doppler Resolutions Calculation

Model Name	$f_c = \frac{f_l + f_u}{2}$ (GHz)	$B = f_u - f_l$ (GHz)	$\lambda = \frac{c_0}{f_c}$ (mm)	$\Delta\rho$ (mm)	$\Delta\theta$ (rad)	Δv_d (mm/s)
Continental ARS 408-21	76.5	1	3.92	150	0.0417	51.1
TI AWR1642	78.5	5	3.85	30	0.248	49.76
TI IWR6843	62	4	4.84	37.5	0.167	63
Omnipresense OPS243-C	24.125	0.25	12.44	600	2	161.92

Since this project involves data fusion at the object level, radar modules with advanced processing capabilities, such as providing high-level data like range, speed, detected objects, and azimuth angle, will both reduce the computational load on the Raspberry Pi and ensure we have accurately processed data to work with for data fusion. While we will still work on understanding the theory and Python code for processing techniques like FFT, CFAR, and clustering, using these advanced radar modules allows us to have reliable data to work with in case things don't go as expected or if time constraints arise. As a result, we have decided to discard the TI IWR1642 module.

After analyzing the resolution measurements in Table 2, we find that the Omnipresense OPS243-C radar module is significantly outperformed by the other modules, leading us to discard it. This leaves us with two contenders: the Continental ARS 408-21 and the TI IWR6843, both of which are within our budget constraints.

The Continental ARS 408-21 offers better angular and Doppler resolution (0.0417 rad vs. 0.167 rad, and 51.1mm/s vs. 63mm/s, respectively), while the TI IWR6843 provides better range resolution (150 mm vs. 30 mm). Additionally, the Continental module requires a higher power supply (12V vs. 5V) but boasts a narrower beamwidth (17° vs. 120°), which contributes to better resolution and accuracy.

Despite the Continental ARS 408-21 outperforming the TI IWR6843 in several important metrics, we selected the TI IWR6843. The primary deciding factor was its easier interface with the Raspberry Pi. Given our team's limited experience, we opted for the TI module to avoid potential complications and time-consuming debugging that could arise from using the more advanced CAN interface in the Continental radar.

Camera Module Selection

Following the same approach as described above, we created a list of potential camera modules, shown in Table 3. In selecting the camera module, we considered factors such as resolution, frame rate, interface type, field of view, and price. Based on [8], stereo cameras introduce complexity in data fusion, leading us to exclude the Arducam Stereo Camera HAT module. Due to budget constraints, the FLIR Chameleon3 with Theia SL183M lens was also ruled out.

The Logitech C920 module was discarded due to its significantly lower resolution (3 MP vs. 12.3 MP – HQ IR Cut camera), as higher resolution enhance the accuracy of object-level data fusion. This left us with the first two modules. While there is only a subtle difference between the two for most features, the significant difference lies in the field of view category. The HQ IR-Cut Camera Module has a narrower field of view, 65°(H), which allows for more accurate capturing of details [9]. On the other hand, the Raspberry Pi Camera Module 3 (Wide) offers a wider field of view, 102°(H), providing greater coverage at the expense of less clarity. Therefore, we propose using the **HQ IR-Cut Camera Module**, since using a camera with a narrower FoV would allow for more accurate object detection, hence enhancing the accuracy of the future radar – camera data fusion system.

Table 3. Camera Modules Features

Model Name	Resolution (Mega Pixels - MP)	Frame rate at 1080 p	Field of View (FoV)	Interface Types	Price (AED)
Raspberry Pi Camera Module 3 (Wide) [10]	11.0	50	120°(D) 102°(H) 67°(V)	CSI (Camera Serial Interface)	~130
HQ IR-Cut Camera Module [11]	12.3	30	65°(H)	2-lane MIPI CSI-2	~280
Arducam Stereo Camera HAT [12]	8	30	66°(D) 54°(H) 41°(V)	CSI	~400
FLIR Chameleon3 with the lens of Theia SL183M [13]	1.3	149	53.6°	USB 3.0	~2,000
Logitech C920 [14]	3	30	78°	USB 2.0	~300

Other Components

Table 4 shows list the components needed for this stage of the project, along with their description and costs.

Table 4. Components, their Description, and Costs

Component	Description	Price (AED)
Raspberry Pi 5 (4GB) Starter KiT	The kit includes Raspberry Pi case, MicroSD Card (64 GB), Heat Sinks and Cooling Fan, Micro – HDMI to HDMI Cable, Power Supply)	~700
GPIO Extension Board	T-type extension board with ribbon cable and breadboard.	~25
Jumper Wires	M-F & M-M wires for connections.	~10
SD Card Reader	For downloading the Raspberry Pi's operating system	~40
Battery Pack	5V power bank with 5A output for portable use.	~100
Accelerator	For acceleration of machine learning (ML) inferencing calculations.	~500

Budget Summary Table

Table 5 presents the budget summary for the project.

Table 5. Budget Summary

Item	Quantity	Unit Cost (AED)	VAT	Link	Delivery Time	Delivery costs (AED)	Total cost (AED)
TI IWR6843AOPEVM	1	547.28	-	1	2-5	44.08	591.26
HQ IR-Cut Camera Module	1	334	-	2	4-12	140	474
Raspberry Pi 5 (4GB) Starter KiT	1	700	35	3	-	0	735
USB Edge TPU ML Accelerator	1	589	-	4		210	799
GPIO Extension Board	1	61.07	2.91	5	-	20	81.07
Jumper Wires	1	40.90	2	6	1	0	42.9
SD Card Reader	1	99	4.95	7	1	0	103.95
Battery Pack	1	95	4.75	8	1	10	109.75
Radar Aiming Corner Reflector	1	375.51	23.59	9	-	72.64	471.74
Fog Simulator	1	70.05	-	10	-	155.75	225.8
FFC Ribbon Flexible Flat Cable for Camera Module	1	18.31	-	11	-	20	38.31
Total Cost	**3668.53 AED**						

Reference

[1]. R. Zhang and S. Cao, “Extending Reliability of mmWave Radar Tracking and Detection via Fusion With Camera,” *IEEE Access*, vol. 7, pp. 137065–137079, 2019, doi: https://doi.org/10.1109/access.2019.2942382.

[2] P. Green, “Radar Antennas in Action: Analyzing Beamwidth and Polarization with SkySim,” *Skyradar.com*, Nov. 07, 2024. https://www.skyradar.com/blog/radar-antennas-in-action-analyzing-beamwidth-and-polarization-with-skysim (accessed Jan. 31, 2025).

[3] Dipl.-I. . F. Gerhardes, S. Leuchs, and O. Arpe, “E-Band Based Car Radar Emblem Measurements,” 2019.

[4] “ARS 408-21 Long Range Radar Sensor 77 GHz.” Accessed: Jan. 31, 2025. [Online]. Available: https://conti-engineering.com/wp-content/uploads/2020/02/ARS-408-21_EN_HS-1.pdf

[5] “AWR1642 Single-Chip 77 and 79GHz FMCW Radar sensor.” Accessed: Jan. 31, 2025. [Online]. Available: https://www.ti.com/lit/ds/symlink/awr1642.pdf

[6] “IWR6843, IWR6443 Single-Chip 60-to 64-GHz mmWave Sensor.” Accessed: Jan. 31, 2025. [Online]. Available: https://www.ti.com/lit/ds/symlink/iwr6843.pdf

[7] “OPS243 Radar Sensor.” https://www.novitronic.com/Datenbl%C3%A4tter/Sensoren/OPS243-Product-Brief_004-E.pdf (accessed Jan. 31, 2025).

[8] Riccardo Giubilato, M. Pertile, and Stefano Debei, “A comparison of monocular and stereo visual FastSLAM implementations,” *ResearchGate*, pp. 227–232, Jun. 2016, doi: https://doi.org/10.1109/metroaerospace.2016.7573217.

[9] “Understanding the trade-off between image resolution and field of view - asmag.com,” *www.asmag.com*, Jun. 29, 2021. https://www.asmag.com/showpost/19058.aspx

[10] “Raspberry Pi Camera Module 3 Standard NoIR Wide NoIR Wide,” 2023. Available: https://datasheets.raspberrypi.com/camera/camera-module-3-product-brief.pdf

[11] “Arducam High Quality IR-CUT Camera for Raspberry Pi, 12.3MP 1/2.3 Inch 477P HQ Camera Module with 6mm CS Lens for Pi 4B, 3B+, 2B, 3A+, Pi Zero and more - Arducam,” *Arducam*, Jul. 03, 2024. https://www.arducam.com/product/arducam-high-quality-ir-cut-camera-for-raspberry-pi-12-3mp-1-2-3-inch-imx477-hq-camera-module-with-6mm-cs-lens-for-pi-4b-3b-2b-3a-pi-zero-and-more/ (accessed Jan. 31, 2025).

[12] “Synchronized Stereo Camera HAT for Raspberry Pi Datasheet.” Accessed: Jan. 31, 2025. [Online]. Available: https://arducam.com/downloads/modules/RaspberryPi_camera/Synchronized_Stereo_Camera_HAT_DS.pdf

[13] “Chameleon3 USB3 | Teledyne Vision Solutions,” *Teledynevisionsolutions.com*, 2017. https://www.teledynevisionsolutions.com/en-150/products/chameleon3-usb3/?vertical=machine+vision&segment=iis (accessed Jan. 31, 2025).

[14] “C920 HD Pro Webcam,” *Logitech.com*, 2025. https://www.logitech.com/en-ae/products/webcams/c920-pro-hd-webcam.960-001055.html?srsltid=AfmBOoru1SIoJYN4Ay22K3tCz6B9EdEG_q2o3M8eQJlDS6tCgHl7GhGP (accessed Jan. 31, 2025).