Project Report

on

SIMULATION OF SENSOR FUSION MODULE USING RADAR AND VISION SENSOR FOR AUTONOMOUS VEHICLE IN MATLAB

Submitted in the partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

In

ELECTRONICS AND COMMUNICATION ENGINEERING

By

K KISHORE REDDY **16311A0429**

M SUMA **16311A0430**

UNDER THE GUIDANCE OF

Mr. MRK Naidu

HEAD R&D

Pedvak Technologies Pvt. Ltd

Dr.Ameet Chavan

Professor

Department of ECE

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

SREENIDHI INSTITUTE OF SCIENCE AND TECHNOLOGY

(Autonomous)

Yamnampet (V), Ghatkesar (M), Hyderabad – 501 301.

April 2020

SREENIDHI INSTITUTE OF SCIENCE AND TECHNOLOGY

(Affiliated to Jawaharlal Nehru Technological University, Hyderabad)

Yamnampet (V), Ghatkesar (M), Hyderabad – 501 301

Department of Electronics and Communication Engineering

CERTIFICATE

This is to certify that the project report entitled **"SIMULATION OF SENSOR FUSION MODULE USING RADAR AND VISION SENSOR FOR AUTONOMOUS VEHICLE IN MATLAB"** is being submitted by

K KISHORE REDDY 16311A0429

M SUMA 16311A0430

in partial fulfillment of the requirements for the award of **Bachelor of Technology** degree in **Electronics and Communication Engineering** to **Sreenidhi Institute of Science and Technology** affiliated to **Jawaharlal Nehru Technological University, Hyderabad** (Telangana)**.** This record is a bona fide work carried out by them under our guidance and supervision. The results embodied in the report have not been submitted to any other University or Institution for the award of any degree or diploma.

Dr.Ameet Chavan
Professor
Departmentof ECE

Dr. T.Ramaswamy
AssociateProfessor
Department ofECE

Head of the Department
Dr. S.P.V. SUBBA RAO
Professor, Department of ECE

Signature of the External Examiner

DECLARATION

We hereby declare that the work described in this thesis titled **"SIMULATION OF SENSOR FUSION MODULE USING RADAR AND VISION SENSOR FOR AUTONOMOUS VEHICLE IN MATLAB"** which is being submitted by us in partial fulfillment for the award of Bachelor of Technology in the Department of **Electronics and Communication Engineering,** Sreenidhi Institute Of Science and Technology is the result of investigations carried out by us under the guidance of **Dr. Ameet Chavan, Professor, Department of ECE, Sreenidhi Institute of Science and Technology, Hyderabad.**

No part of the thesis is copied from books/ journals/ internet and whenever the portion is taken, the same has been duly referred. The report is based on the project work done entirely by us and not copied from any other source. The work is original and has not been submitted for any Degree/ Diploma of this or any other university.

Place: Hyderabad
Date: 23th April, 2020

K KISHORE REDDY **16311A0429**

M SUMA **16311A0430**

ACKNOWLEDGEMENTS

We would like to take this opportunity to express our sincere gratitude to our Internal Guide - **Dr. Ameet Chavan** (Professor, Department of ECE), Sreenidhi Institute of Science and Technology, Hyderabad, and our External Guide – **Mr. M.R.K.Naidu** (Head R&D), Pedvak Technologies Pvt. Ltd, ECIL for their invaluable support and guidance throughout the duration of this project phase.

We thank our Project Co-ordinator - **Dr.T.Ramaswamy** (Associate Professor, Department of ECE), Sreenidhi Institute of Science and Technology, Hyderabad, for his valuable comments and suggestions that greatly helped in improving the quality of our project.

We express our heartfelt gratitude to **Dr. Narsimha reddy** (Executive Director, SNIST), **Mr. K. Abhijit Rao** (Director, SNIST), **Dr. T. Ch. Shiva Reddy (**Principal, SNIST) and **Dr. S. P. V. Subba Rao**, (Prof. & Head, Department of ECE, SNIST) for supporting us in executing this project throughout the course of this semester.

We also wish to extend a special thanks to our colleagues, friends and family for their unceasing encouragement and support and for helping us indirectly or directly to complete this project.

K KISHORE REDDY **16311A0429**

M SUMA **16311A0430**

ABSTRACT

Autonomous systems use sensors to sense the surroundings and take decisions based on the sensed data. But the sensor reading may not be accurate all the time. There might be errors in the sensed data compared to the actual data. These errors may arise due to various reasons and may result in failure of the system. To overcome this barrier sensor fusion technique is used. In this technique multiple sensors of the same type or multiple sensors of the different types are used to sense a particular physical parameter. The sensed data from those various sensors are fused in a way that generates a better understanding of the system surroundings and thus facilitate good decision making capabilities.

The aim of this project is to design and simulate Sensor Fusion Module in MATLAB to fuse the data from RADAR and Vision sensor to differentiate the necessary object detections and unnecessary object detections. The necessary object detections are refered as important object detections and the unnecessary object detections are refered as clusters in the project. Automated Driving toolbox, Sensor Fusion toolbox and Computer Vision toolbox available in MATLAB are used to achieve the purpose.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

FOV	Field Of View
AV	Autonomous Vehicle
SAE	Society of Automotive Egineers
ACC	Adaptive Cruise Control
LKA	Lane Keeping Assistance
IMU	Inertial Measurement Unit
KF	Kalman Filter
EKF	Extended Kalman Filter
UKF	Unscented Kalman Filter
LKA	Lane Keeping Assistance
ADAS	Advance Driver Assistance System
FCW	Forward Collision Warning
AEB	Auto Emergency Braking
Euro NCAP	European New Car Assessment Programme

CHAPTER 1
INTRODUCTION

1.1. PROBLEM STATEMENT

For automated movement of Autonomous Vehicles various Sensors like RADAR, LIDAR, Vision Sensor, IMU Sensor, Odometer etc. are employed. For the purpose of object detections and tracking RADAR, LIDAR and Vision Sensor are used. Each of RADAR, LIDAR and Vision Sensor has different Field Of View (FOV) and so detect and track different number of objects based on their field of view. Each of RADAR, LIDAR and Vision Sensor has different abilities. RADAR and LIDAR sensors give the exact information about the distance of the objects from the Ego vehicle but cannot identify the objects. The Vision Sensor has the ability to identify the objects but cannot directly give information about the distance of the objects from the ego vehicle. So individually the RADAR, LIDAR and Vision Sensors cannot facilitate object detection and tracking in all possible scenarios.

RADAR, LIDAR and Vision Sensors provide different set of information about the objects. RADAR and LIDAR sensors provide the exact information about the distance of the objects from the ego vehicle. The Vision Sensor provides information about the type of detected object. By combining/fusing the data from both the RADAR and Vision Sensor, differentiation of necessary and unnecessary object detections in driving scenarios can be possible.

1.2. OBJECTIVE OF THE PROJECT

The aim of this project is to design and simulate Sensor Fusion Module in MATLAB to fuse the data from RADAR and Vision sensor to differentiate the necessary object detections and unnecessary object detections.
The unnecessary object detections are the object detections that arise due to stationary objects, objects whose relative velocity is very less or objects in the lanes other than the lane of the vehicle.
The necessary object detections are the object detections due to the presence of object in front of the vehicle in the same lane.

1.3. ORGANISATION OF THE REPORT

This report is organized as follows:

Chapter 1 describes the problem statement and the objective of this project.
Chapter 2 talks about the history of Autonomous Vehicles, classifications of Autonomous Vehicles and the capabilities of the Autonomous Vehicles.
Chapter 3 talks about Sensor Fusion, the need for Sensor Fusion, the sensors used in Autonomous Vehicles, types of Kalman Filters and the benefits of Sensor Fusion.
Chapter 4 talks about the need for simulation in Sensor Fusion design and the tools available in MATLAB to facilitate Sensor Fusion design.
Chapter 5 talks about the proposed designflow. It provides information about the functions that are designed, functions available in MATLAB which help in designing the Sensor Fusion Module.
Chapter 6 shows the results of simulation which includes the processed data and the snapshots of the simulation.
Finally the conclusion and future scope are mentioned in chapter 7.

CHAPTER 2
LITERATURE SURVEY

2.1. HISTORY OF AUTONOMOUS VEHICLES

Around the world efforts are being made to develop self driving vehicles. A **self-driving car**, also known as an **autonomous vehicle** (**AV**), **connected and autonomous vehicle** (**CAV**), **driverless car**, **robo-car**, or **robotic car**, is a vehicle that is capable of sensing its environment and moving safely with little or no human input.

Experiments have been conducted on self-driving cars since at least the 1920s, promising trials took place in the 1950s and work has proceeded since then. The first self-sufficient and truly autonomous cars appeared in the 1980s, with Carnegie Mellon University's Navlab and ALV projects in 1984 and Mercedes-Benz and Bundeswehr University Munich's Eureka Prometheus Project in 1987. Since then, numerous major companies and research organizations have developed working autonomous vehicles including Mercedes-Benz, General Motors, Continental Automotive Systems, AutolivInc., Bosch, Nissan, Toyota, Audi, Volvo, Vislab from Universiy of Parma, Oxford University and Google. In July 2013, Vislab demonstrated BRAiVE, a vehicle that moved autonomously on a mixed traffic route open to public traffic.

As of 2019, twenty-nine U.S. states have passed laws permitting autonomous cars. In Europe, cities in Belgium, France, Italy and the UK are planning to operate transport systems for driverless cars, and Germany, the Netherlands, and Spain have allowed testing robotic cars in traffic.

With the advancement of technology and the availability of technologies like Artificial Intelligence and Neural Networks Autonomous vehicles are evolving from futuristic dream to modern reality.

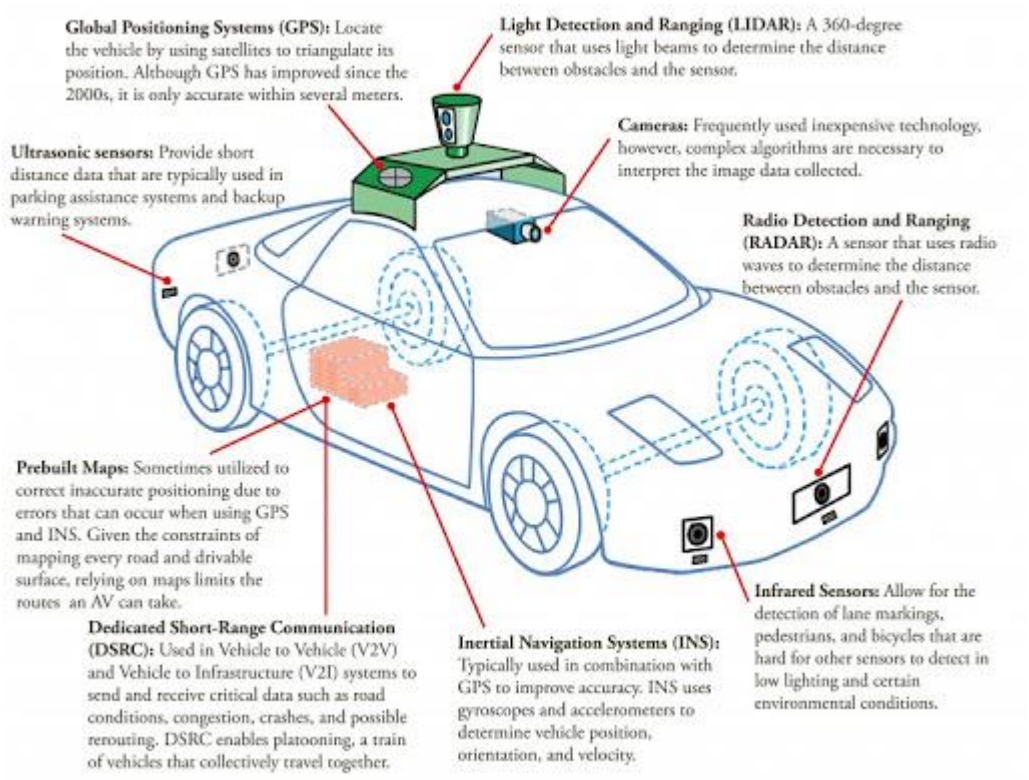


Figure.2.1. Autonomous Car

2.2. CLASSIFICATION OF AUTONOMOUS VEHICLES

A classification system for Autonomous vehicles – ranging from fully manual vehicles to fully automated vehicles – was published as **J3016** by **Society of Automotive Engineers (SAE)**, an international non-profit educational and scientific organization dedicated to advancing mobility technology, in 2014.

According to the classification system, the autonomous vehicles are classified into six levels – Level 0 to Level 5.

In SAE's automation level definitions, "driving mode" means "a type of driving scenario with characteristic dynamic driving task requirements (e.g., expressway merging, high speed cruising, low speed traffic jam, closed-campus operations, etc.)".

- **Level 0:** The automated system issues warnings and may momentarily intervene but has no sustained vehicle control.
- **Level 1 ("hands on"):** The driver and the automated system share control of the vehicle. Examples are systems where the driver controls steering

and the automated system controls engine power to maintain a set speed (Cruise Control) or engine and brake power to maintain and vary speed (Adaptive Cruise Control or ACC); and Parking Assistance, where steering is automated while speed is under manual control. The driver must be ready to retake full control at any time. Lane Keeping Assistance (LKA) Type II is a further example of Level 1 self-driving. The Automatic Emergency Braking feature which alerts the driver to a crash and deploys full braking capacity is also a Level 1 feature.

- **Level 2 ("hands off"):** The automated system takes full control of the vehicle: accelerating, braking, and steering. The driver must monitor the driving and be prepared to intervene immediately at any time if the automated system fails to respond properly. The shorthand "hands off" is not meant to be taken literally – contact between hand and wheel is often mandatory during SAE 2 driving, to confirm that the driver is ready to intervene.
- **Level 3 ("eyes off"):** The driver can safely turn their attention away from the driving tasks, e.g. the driver can text or watch a movie. The vehicle will handle situations that call for an immediate response, like emergency braking. The driver must still be prepared to intervene within some limited time, specified by the manufacturer, when called upon by the vehicle to do so.
- **Level 4 ("mind off"):** As level 3, but no driver attention is ever required for safety, e.g. the driver may safely go to sleep or leave the driver's seat. Self-driving is supported only in limited spatial areas (geofenced) or under special circumstances. Outside of these areas or circumstances, the vehicle must be able to safely abort the trip, e.g. park the car, if the driver does not retake control. An example would be a robotic taxi or a robotic delivery service that only covers selected locations in a specific area.
- **Level 5 ("steering wheel optional"):** No human intervention is required at all. An example would be a robotic taxi that works an all roads all over the world, all year around, in all weather conditions.

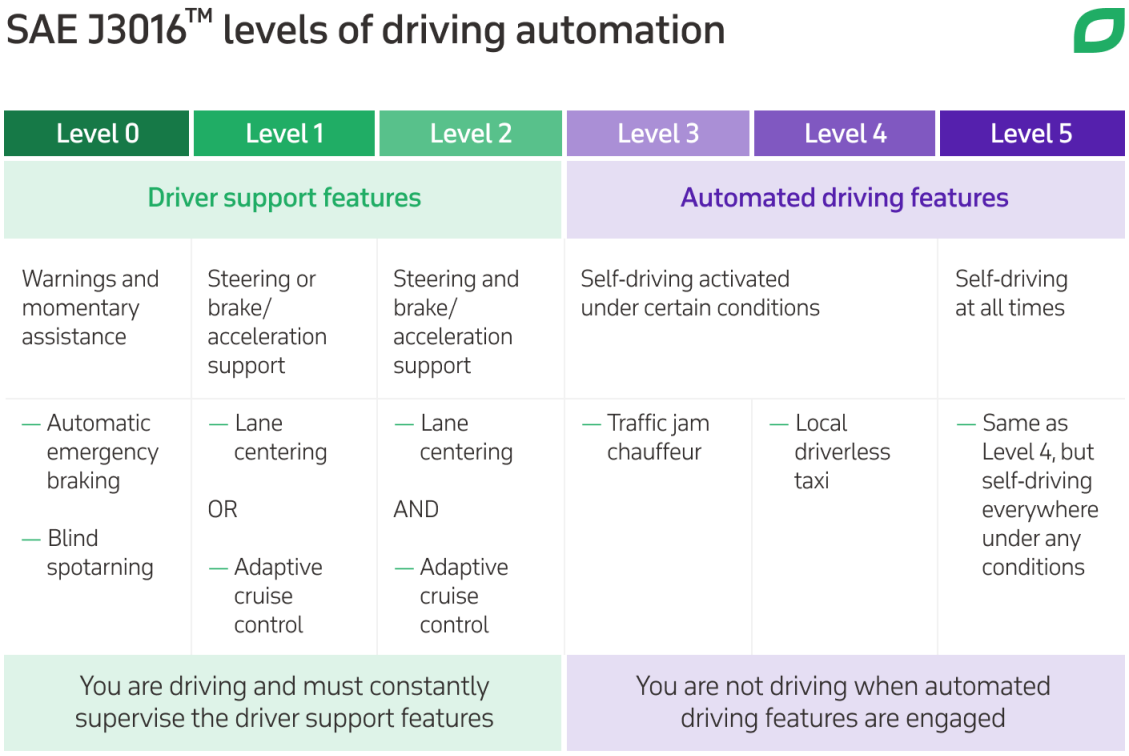


Figure.2.2.Classification of Autonomous Vehicles

2.3. CAPABILITIES OF AUTONOMOUS SYSTEMS

Autonomous systems need to interact with the world around them and in order to be successful there are certain capabilities that the system needs to have. These capabilities can be divided into four main areas: **sense, perceive, plan, and act.**

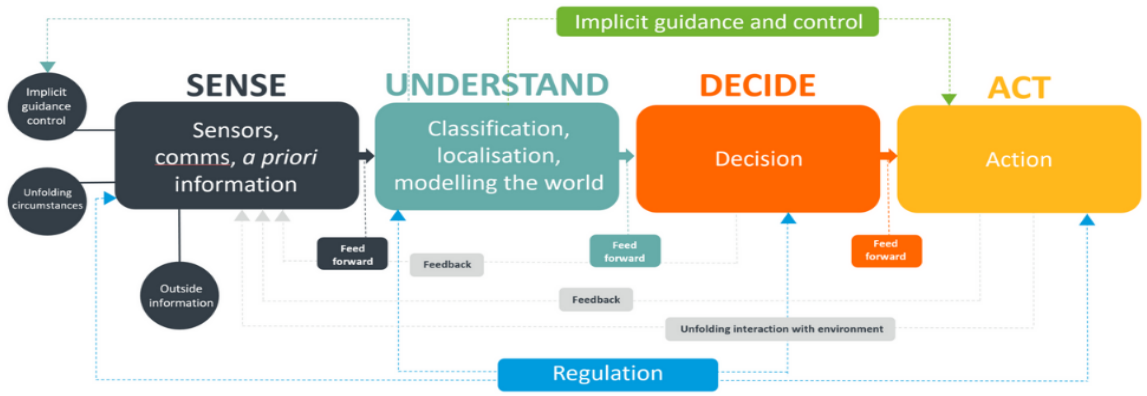


Figure.2.3. Capabilities of Autonomous Vehicles

Sense refers to directly measuring the environment with sensors. It's collecting information from the system and the external world. But simply gathering data with sensors isn't good enough, because the system needs to be able to interpret the data and turn it into something that can be understood and acted on by the autonomous system. This is the role of the perceive step: to make sense of, well, the sensed data. For example, let's say this is an image from a vehicle's camera sensor. The car has to ultimately interpret the blob of pixels as a road with lane lines, and that there's something off to the side that could be a pedestrian about to cross the street or a stationary mailbox. This level of understanding is critical in order for the system to determine what to do next. This is the planning step, where it figures out what it would like to do and finds a path to get there. Lastly, the system calculates the best actions that get the system to follow that path. This last step is what the controller and the control system is doing.

The **perceive** step has two different but equally important responsibilities. It is responsible for self-awareness, which is referred to as localization or positioning. It is also responsible for situational awareness, things like detecting other objects in the environment and tracking them.

Sensor fusion straddles **sense** and **perceive**, as it has a hand in both of these capabilities. It's the process of taking the multiple sensor measurements, combining them, and mixing in additional information from mathematical models with the goal of having a better understanding of the world with which the system can use to **plan** and **act**.

CHAPTER 3
SENSOR FUSION

3.1. SENSOR FUSION - DEFINITION

Sensor fusion is an integral part of the design of autonomous systems; things like self-driving cars, RADAR tracking stations, and the Internet of Things all rely on sensor fusion of one sort or another.

Sensor fusion is combining two or more data sources in a way that generates a better understanding of the system. “Better” here refers to the solution being more consistent over time, more accurate, and more dependable than it would be with a single data source. The data sources are generally sensors. Mathematical models, designed with knowledge of the physical world and encoding that knowledge into the fusion algorithm to improve the measurements from the sensors, can also be considered as data sources.

3.2. NEED FOR SENSOR FUSION

Even the very precise sensors may not give correct output values all the time. Due to various reasons output values of the sensors may differ from the original values. For systems like autonomous vehicles, these errors may be fatal.

To overcome this problem, multiple sensors of the same type or of different types are used to measure a single parameter thereby reducing the probability of error in sensor data.

The system can be understood in a better way by combining different parameters obtained from different type of sensors.

This combining of data from different sources is referred to as Sensor Fusion.

3.3. SENSORS USED IN AUTONOMOUS VEHICLES

For an Autonomous Vehicle, the sensor suite includes RADAR, LIDAR, visible cameras, and many more. The three primary autonomous vehicle sensors are camera, RADAR and LIDAR. Working together, they provide the car visuals of its surroundings and help it detect the speed and distance of nearby objects, as well as their three-dimensional shape. In addition, sensors known as inertial measurement units help track a vehicle's acceleration and location.

3.3.1. CAMERA SENSOR

Figure.3.1.Camera Sensor

- From photos to video, cameras are the most accurate way to create a visual representation of the world, especially when it comes to self-driving cars.
- Autonomous vehicles rely on cameras placed on every side — front, rear, left and right — to stitch together a 360-degree view of their environment. Some have a wide field of view — as much as 120 degrees — and a shorter range. Others focus on a more narrow view to provide long-range visuals.
- Some cars even integrate fish-eye cameras, which contain super-wide lenses that provide a panoramic view, to give a full picture of what's behind the vehicle for it to park itself.
- Though they provide accurate visuals, cameras have their limitations. They can distinguish details of the surrounding environment, however, the distances of those objects needs to be calculated to know exactly where they are. It's also more difficult for camera-based sensors to detect objects in low visibility conditions, like fog, rain or nighttime.

3.3.2. RADAR SENSOR

Figure.3.2.RADAR Sensor

- Radar sensors can supplement camera vision in times of low visibility, like night driving, and improve detection for self-driving cars.
- Traditionally used to detect ships, aircraft and weather formations, radar works by transmitting radio waves in pulses. Once those waves hit an object, they return to the sensor, providing data on the speed and location of the object.
- Like the vehicle's cameras, radar sensors typically surround the car to detect objects at every angle. They're able to determine speed and distance, however, they can't distinguish between different types of vehicles.

Camera and radar are common sensors: most new cars today already use them for advanced driver assistance and park assist. They can also cover lower levels of autonomy when a human is supervising the system. However, for full driverless capability, lidar — a sensor that measures distances by pulsing lasers — has proven to be incredibly useful.

3.3.3. LIDAR SENSOR

Figure.3.3. LIDAR Sensor

- Lidar makes it possible for self-driving cars to have a 3D view of their environment. It provides shape and depth to surrounding cars and pedestrians as well as the road geography. And, like radar, it works just as well in low-light conditions.
- By emitting invisible lasers at incredibly fast speeds, lidar sensors are able to paint a detailed 3D picture from the signals that bounce back instantaneously. These signals create “point clouds” that represent the vehicle’s surrounding environment to enhance safety and diversity of sensor data.
- Vehicles only need lidar in a few key places to be effective. However, the sensors are more expensive to implement — as much as 10 times the cost of camera and radar — and have a more limited range.

3.3.4. INERTIAL MEASUREMENT UNIT (IMU)

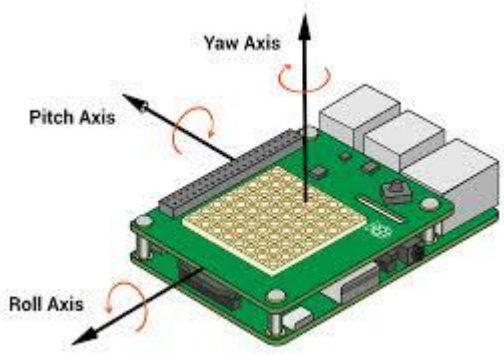


Figure.3.4.IMU Sensor

- An inertial measurement unit (IMU) is a device that directly measures a vehicle's three linear acceleration components and three rotational rate components (and thus it has six degrees of freedom).
- The typical IMU for autonomous vehicle use includes a three-axis accelerometer and three-axis rate sensor. Those that are 9-DOF (degrees of freedom) units include a three-axis magnetometer.
- The IMU helps provide "localization" data i.e., information about where the vehicle is.
- An IMU is unique among the sensors typically found in an autonomous vehicle (AV) because an IMU needs no connection to or knowledge of the external world. This environment independence makes the IMU a core technology for both safety and sensor-fusion.
- A true IMU is just 6-DOF. A magnetometer is not particularly useful in automotive apps due to a vehicle's local magnetic field and fields of nearby cars and trucks.

3.4. KALMAN FILTER

The algorithm used to merge the data is called a **Kalman filter**.

The Kalman filter is one of the most popular algorithms in data fusion. Invented in 1960 by Rudolph Kalman, it is now used in our phones or satellites for navigation and tracking. The most famous use of the filter was during the Apollo 11 mission to send and bring the crew back to the moon.

A Kalman filter is generally used for data fusion to estimate the state of a dynamic system (evolving with time) in the present (filtering), the past (smoothing) or the future (prediction). Sensors embedded in autonomous vehicles emit measures that are sometimes incomplete and noisy. The inaccuracy of the sensors (noise) is a very important problem and can be handled by the Kalman filters.

A Kalman filter is used to **estimate the state of a system**, denoted **x**. This vector is composed of a position **p** and a velocity **v**. At each estimate, we associate a measure of **uncertainty P**.

$$x = \begin{pmatrix} p \\ v \end{pmatrix}$$

State Representation of a system

By performing a fusion of sensors, we take into account different data for the same object. A radar can estimate that a pedestrian is 10 meters away while the Lidar estimates it to be 12 meters. The use of Kalman filters allows you to have a precise idea to decide how many meters really is the pedestrian by eliminating the noise of the two sensors.

A Kalman filter can generate estimates of the state of objects around it. To make an estimate, it only needs the current observations and the previous prediction. The measurement history is not necessary. This tool is therefore light and improves with time.State and uncertainty are represented by Gaussians.

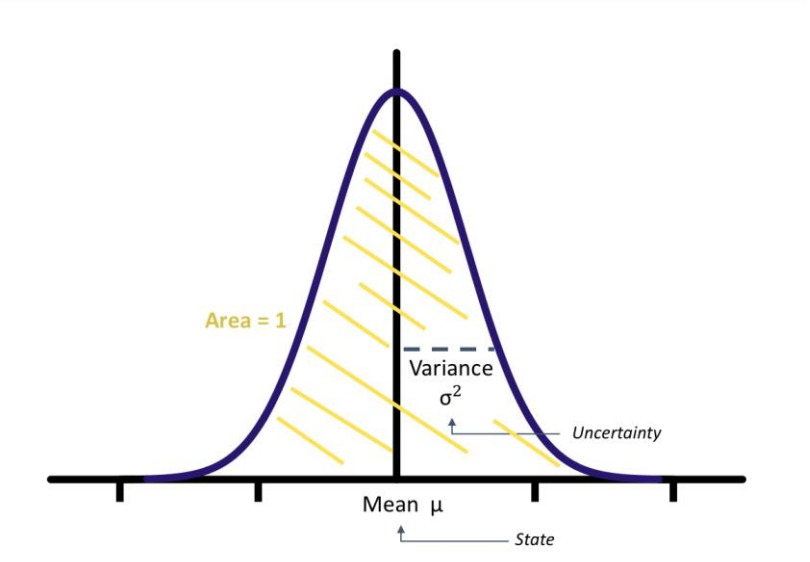


Figure 3.5.Representation of Gaussian

A Gaussian is a continuous function under which the area is 1. This allows us to represent probabilities. We are on a **probability to normal distribution**. The uni-modality of the Kalman filters means that we have a single peak each time to estimate the state of the system.

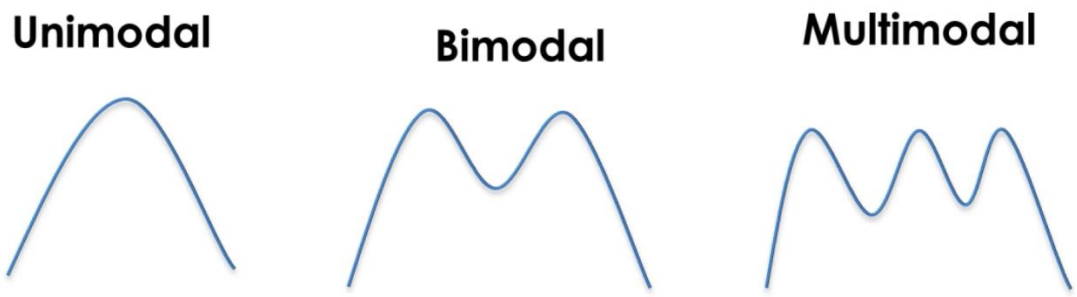


Figure 3.6. Representation of Unimodal, Bimodal and Multimodal Gaussians

We have a **mean μ** representing a state and a **variance σ²** representing an uncertainty. The larger the variance, the greater is the uncertainty.

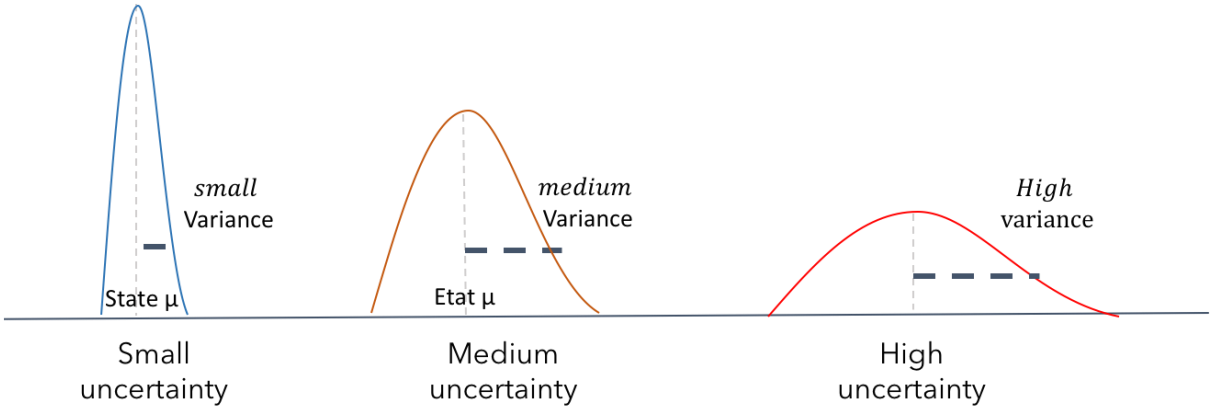


Figure 3.7. Gaussians with different variance

Gaussians make it possible to estimate probabilities around the state and the uncertainty of a system. A Kalman filter is a continuous and uni-modal function.

3.4.1. BAYESIAN FILTERING

In general, a Kalman filter is an implementation of a Bayesian filter, ie a sequence of alternations between prediction and update or correction.

Figure 3.8. Bayes filter

Prediction: We use the estimated state to predict the current state and uncertainty.

Update: We use the observations of our sensors to correct the predicted state and obtain a more accurate estimate.

To make an estimate, a Kalman filter only needs current observations and the previous prediction. The measurement history is not necessary.

The mathematics behind the Kalman filters are made of additions and multiplications of matrices. We have two stages: **Prediction** and **Update**.

Prediction

Our prediction consists of estimating **a state x'** and **an uncertainty P'** at time **t** from the previous states x and P at **time t-1**.

$$x' = Fx + u$$

$$P' = FPF^T + Q$$

Prediction formulas

- F: Transition matrix from t-1 to t
- ν: Noise added
- Q: Covariance matrix including noise

Update

The update phase consists of using a ***z* measurement** from a sensor to correct our prediction and thus **predict x and P**.

$$y = z - Hx'$$

$$S = HP'H^T + R$$

$$K = P'H^T S^{-1}$$

$$x = x' + Ky$$

$$P = (I - KH)P'$$

Update formulas

- y: Difference between actual measurement and prediction, ie the error.
- S: Estimated system error
- H: Matrix of transition between the marker of the sensor and ours.
- R: Covariance matrix related to sensor noise (given by the sensor manufacturer).
- K: Kalman gain. Coefficient between 0 and 1 reflecting the need to correct our prediction.

The update phase makes it possible to estimate an x and a P closer to reality than what the measurements provide.

A Kalman filter allows predictions in real time, without data beforehand. We use a mathematical model based on the multiplication of matrices for each time defining a state x (position, speed) and uncertainty P.

3.4.2. REPRESENTATION OF STATES IN KALMAN FILTERING

This diagram shows what happens in a Kalman filter.

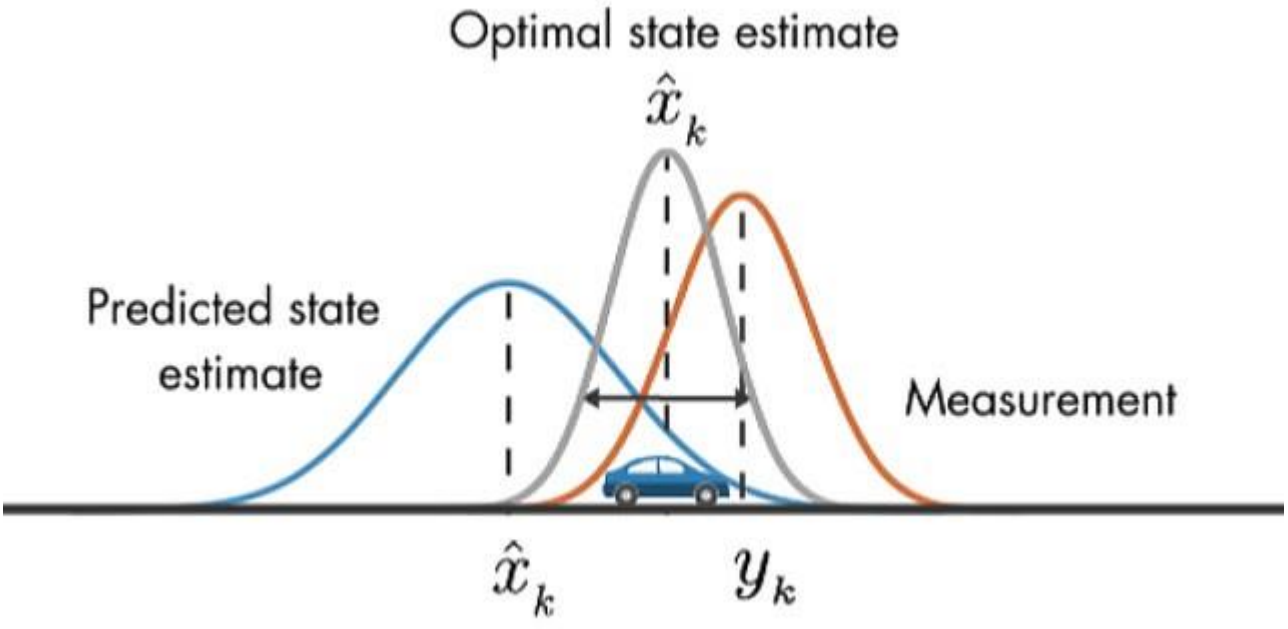
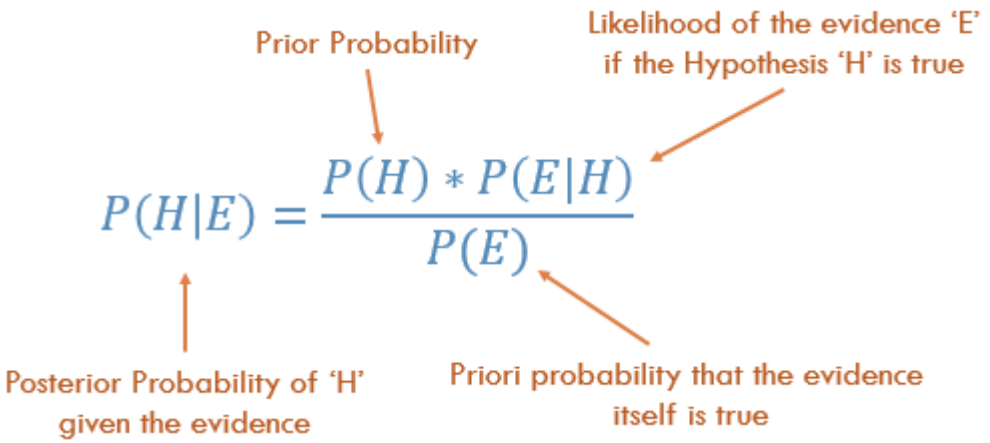


Figure3.9. Kalman filter estimation

- **Predicted state estimate** represents our first estimate, our prediction phase. We are talking about prior.
- **Measurement** is the measurement from one of our sensors. We have better uncertainty but the noise of the sensors makes it a measurement that is always difficult to estimate. We talk about likelihood.
- **Optimal State Estimate** is our update phase. The uncertainty is this time the weakest, we accumulated information and allowed to generate a value more sure than with our sensor alone. This value is our best guess. We speak of posterior.

Kalman filter actually implements a Bayes rule.



In a Kalman filter, we loop predictions from measurements. Our predictions are always more precise since we keep a measure of uncertainty and regularly calculate the error between our prediction and reality. We are able from matrix multiplications and probability formulas to estimate velocities and positions of vehicles around us.

There are two assumptions while working with Kalman Filter

1. KalmanFilter will always work with **Gaussian Distribution.**
2. 2. Kalman Filter will always work with **Linear Functions.**

3.5. EXTENDED KALMAN FILTER (EKF)

In the extended Kalman filter, the state transition and observation models don't need to be linear functions of the state but may instead be differentiable functions.

$$\boldsymbol{x}_k = f(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k) + \boldsymbol{w}_k$$
$$\boldsymbol{z}_k = h(\boldsymbol{x}_k) + \boldsymbol{v}_k$$

Here $\mathbf{w}_k$ and $\mathbf{v}_k$ are the process and observation noises which are both assumed to be zero mean multivariate Gaussian noises with covariance $\mathbf{Q}_k$ and $\mathbf{R}_k$ respectively. $\mathbf{u}_k$ is the control vector.

The function f can be used to compute the predicted state from the previous estimate and similarly the function h can be used to compute the predicted measurement from the predicted state. However, f and h cannot be applied to the covariance directly. Instead a matrix of partial derivatives (the Jacobian) is computed.

At each time step, the Jacobian is evaluated with current predicted states. These matrices can be used in the Kalman filter equations. This process essentially linearizes the non-linear function around the current estimate.

3.5.1. DISCRETE-TIME PREDICT AND UPDATE EQUATIONS

Notation $\hat{\mathbf{x}}_{n|m}$ represents the estimate of $\mathbf{x}$at time n given observations up to and including at time $\mathbf{m} \leq \mathbf{n}$.

PREDICT

Predicted state estimate

$$\hat{\boldsymbol{x}}_{k|k-1} = f(\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k)$$

Predicted covariance estimate

$$\boldsymbol{P}_{k|k-1} = \boldsymbol{F}_k \boldsymbol{P}_{k-1|k-1} \boldsymbol{F}_k^\top + \boldsymbol{Q}_k$$

UPDATE

Innovation or measurement residual

$$\tilde{\boldsymbol{y}}_k = \boldsymbol{z}_k - h(\hat{\boldsymbol{x}}_{k|k-1})$$

Innovation (or residual) covariance

$$\boldsymbol{S}_k = \boldsymbol{H}_k \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^\top + \boldsymbol{R}_k$$

Near-optimal Kalman gain

$$\boldsymbol{K}_k = \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^\top \boldsymbol{S}_k^{-1}$$

Updated state estimate

$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \tilde{\boldsymbol{y}}_k$$

Updated covariance estimate

$$\boldsymbol{P}_{k|k} = (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{P}_{k|k-1}$$

Where the state transition and observation matrices are defined to be the following Jacobians

$$\boldsymbol{F}_k = \left.\frac{\partial f}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k}$$

$$\boldsymbol{H}_k = \left.\frac{\partial h}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k|k-1}}$$

3.5.2. CONTINUOUS-TIME PREDICT AND UPDATE EQUATIONS

Model

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)) + \mathbf{w}(t) \qquad \mathbf{w}(t) \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}(t))$$
$$\mathbf{z}(t) = h(\mathbf{x}(t)) + \mathbf{v}(t) \qquad \mathbf{v}(t) \sim \mathcal{N}(\mathbf{0}, \mathbf{R}(t))$$

Initialize

$$\hat{\mathbf{x}}(t_0) = E[\mathbf{x}(t_0)], \mathbf{P}(t_0) = Var[\mathbf{x}(t_0)]$$

Predict-Update

$$\dot{\hat{\mathbf{x}}}(t) = f(\hat{\mathbf{x}}(t), \mathbf{u}(t)) + \mathbf{K}(t)\big(\mathbf{z}(t) - h(\hat{\mathbf{x}}(t))\big)$$
$$\dot{\mathbf{P}}(t) = \mathbf{F}(t)\mathbf{P}(t) + \mathbf{P}(t)\mathbf{F}(t)^\top - \mathbf{K}(t)\mathbf{H}(t)\mathbf{P}(t) + \mathbf{Q}(t)$$
$$\mathbf{K}(t) = \mathbf{P}(t)\mathbf{H}(t)^\top \mathbf{R}(t)^{-1}$$
$$\mathbf{F}(t) = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}(t), \mathbf{u}(t)}$$
$$\mathbf{H}(t) = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}(t)}$$

Unlike the discrete-time extended Kalman filter, the prediction and update steps are coupled in the continuous-time extended Kalman filter.

3.6. UNSCENTED KALMAN FILTER (UKF)

When the state transition and observation models, i.e., the predict and update functions and are highly nonlinear, the extended Kalman filter can give particularly poor performance. This is because the covariance is propagated through linearization of the underlying nonlinear model. The unscented Kalman filter (UKF) uses a deterministic sampling technique known as the unscented transformation (UT) to pick a minimal set of sample points (called sigma points) around the mean. The sigma points are then propagated through the nonlinear functions, from which a new mean and covariance estimate are then formed. The resulting filter depends on how the transformed statistics of the UT are calculated and which set of sigma points are used. It should be remarked that it is always possible to construct new UKFs in a consistent way. For certain systems, the resulting UKF more accurately estimates the true mean and covariance. This can be verified with Monte Carlo sampling or Taylor series expansion of the posterior statistics. In addition, this technique removes the requirement to explicitly calculate Jacobians, which for complex functions can be a difficult task in itself (i.e., requiring complicated derivatives if done analytically or being computationally costly if done numerically), if not impossible (if those functions are not differentiable).

3.7. BENEFITS OF SENSOR FUSION

The sensor fusion has many benefits. The **first benefit** is sensor fusion increases the quality of the data. We always want to work with data that has less noise, less uncertainty, and fewer deviations from the truth.

As a simple example, let's take a single accelerometer and place it on a stationary table so that it's only measuring the acceleration due to gravity. If this was a perfect sensor, the output would read a constant 9.81 m/s^2. However, the actual measurement will be noisy - how noisy depends on the quality of the sensor. This is unpredictable noise so we can't get rid of it through calibration, but we can reduce the overall noise in the signal if we add a second accelerometer and average the two readings. As long as the noise isn't correlated across the sensors, fusing them together like this reduces the

combined noise by a factor of the square root of the number of sensors. So four identical sensors fused together will have half the noise of a single one. In this case, all that makes up this very simple sensor fusion algorithm is an averaging function.

Noise can also reduced by combining measurements from two or more different types of sensors, and this can help if we have to deal with correlated noise sources. For example, let's say we're trying to measure the direction your phone is facing relative to north. We could use the phone magnetometer to measure the angle from magnetic north, easy. However, just like with the accelerometer, this sensor measurement will be noisy. And if we want to reduce that noise, then we may be tempted to add a second magnetometer. However, at least some contribution of noise is coming from the moving magnetic fields created by the electronics within the phone itself. This means that every magnetometer will be affected by this correlated noise source and so averaging the sensors won't remove it.

Two ways to solve this problem are to simply move the sensors away from the corrupting magnetic fields—hard to do with a phone—or to filter the measurement through some form of a low-pass filter, which would add lag and make the measurement less responsive. But another option is to fuse the magnetometer with an angular rate sensor, a gyro. The gyro will be noisy as well, but by using two different sensor types, we're reducing the likelihood that the noise is correlated and so they can be used to calibrate each other. The basic gist is that if the magnetometer measures a change in the magnetic field, the gyro can be used to confirm if that rotation came from the phone physically moving or if it's just from noise.

There are several different sensor fusion algorithms that can accomplish this blending, but a Kalman filter is probably one of the more common methods. The interesting thing about Kalman filters is that a mathematical model of the system is already built into the filter. So you're getting the benefit of fusing together sensor measurements and your knowledge of the physical world

The **second benefit** of sensor fusion is that it can increase the reliability of the measurements. An obvious example is that if we have two identical sensors fused together, like we had with the averaged accelerometers, then we have a backup in case one fails. Of course, with this scenario, we lose quality if one sensor fails, but at least we don't lose the whole measurement. We can also add a third sensor into the mix and the fusion algorithm could vote out the data of any single sensor that is producing a measurement that differs from the other two.

An example here could be using three pitot tubes to have a reliable measure of an aircraft's air speed. If one breaks or reads incorrectly, then the airspeed is still known using the other two. So duplicating sensors is an effective way to increase reliability; however, we have to be careful of single failure modes that affect all of the sensors at the same time. An aircraft that flies through freezing rain might find that all three pitot tubes freeze up and no amount of voting or sensor fusion will save the measurement.

Again, this is where fusing together sensors that measure different quantities can help the situation. The aircraft could be set up to supplement the air speed measurements from the pitot tubes with an airspeed estimate using GPS and atmospheric wind models. In this case, air speed can still be estimated when the primary sensor suite is unavailable. Again, quality may be reduced, but the airspeed can still be determined, which is important for the safety of the aircraft.

Losing a sensor doesn't always mean the sensor failed. It could mean that the quantity they're measuring drops out momentarily. For example, take a RADAR system that is tracking the location of a small boat on the ocean. The RADAR station is sending out a radio signal which reflects off the boat and back and the round-trip travel time, the doppler shift of the signal, and the azimuth and elevation of the tracking station are all combined to estimate the location and range rate of the boat. However, if a larger cargo ship gets between the RADAR station and the smaller boat, then the measurement will shift instantly to the location and range rate of that blocking object.

So in this case, we don't need an alternate sensor type or a secondary RADAR tracking station to help when the measurement drops out because we can use a model of the physical world. An algorithm could develop a speed and heading model of the object that's being tracked. And then when the object is out of the RADAR line of sight, the model can take over and make predictions. This method works only when the object you're tracking is relatively predictable and you don't have to rely on your predictions long term.

The **third benefit** of sensor fusion is it helps in estimating the unmeasured states. Now, it's important to recognize that unmeasured doesn't mean unmeasurable; it just means that the system doesn't have a sensor that can directly measure the state we're interested in.

For example, a visible camera can't measure the distance to an object in its field of view. A large object far away can take up the same number of pixels as a small but close object. However, we can add a second optical sensor and through sensor fusion, extract three-dimensional information. The fusion algorithm would compare the scene from the two different angles and measure the relative distances between the objects in the two images. So in this way, these two sensors can't measure distance individually but they can when combined. In the next video, we'll expand on this concept of using sensors to estimate unmeasured states by showing how we can estimate position using an accelerometer and a gyro.

The **fourth benefit** of sensor fusion is it helps in increasing the coverage area. Let's imagine the short-range ultrasonic sensors on a car which are used for parking assist. These are the sensors that are measuring the distance to nearby objects, like other parked cars and the curb, to let you know when you're close to impact. Each individual sensor may only have a range of a few feet and a narrow field of view. Therefore, if the car needs to have full coverage on all four sides, additional sensors need to be added and the measurements fused together to produce a larger total field of view.

CHAPTER 4

SENSOR FUSION IN MATLAB

4.1. NEED FOR SIMULATION

Developing Sensor Fusion algorithms and testing the algorithms on all the possible scenarios is practically impossible as it requires huge amounts of space and time. It is also difficult to replicate few real time scenarios. Simulation Tools can be used to severe this purpose. Even the impossible scenarios can be simulated using Simulation Tools and the algorithms can be tested in those scenarios. So Simulation tools play a major role in testing and verifying the designed algorithms.

4.2. TOOLS FOR SENSOR FUSION IN MATLAB

To facilitate designing and testing of autonomous system models, MATLAB provides several tools. All essential tools required to implement similar functions are bundled into a toolbox.

The available toolboxes for Autonomous System design in MATLAB are

4.2.1. SENSOR FUSION AND TRACKING TOOLBOX

Sensor Fusion and Tracking Toolbox includes algorithms and tools for the design, simulation, and analysis of systems that fuse data from multiple sensors to maintain position, orientation, and situational awareness. Reference examples provide a starting point for implementing components of airborne, ground-based, shipborne, and underwater surveillance, navigation, and autonomous systems.

The toolbox includes multi-object trackers, sensor fusion filters, motion and sensor models, and data association algorithms that let you evaluate fusion architectures using real and synthetic data. With Sensor Fusion and Tracking Toolbox you can import and define scenarios and trajectories, stream signals, and generate synthetic data for active and passive sensors, including RF, acoustic, EO/IR, and GPS/IMU sensors. You can also evaluate system accuracy and performance with standard benchmarks, metrics, and animated plots.

4.2.2. AUTOMATED DRIVING TOOLBOX

Automated Driving Toolbox provides algorithms and tools for designing, simulating, and testing ADAS and autonomous driving systems. You can design and test vision and lidar perception systems, as well as sensor fusion, path planning, and vehicle controllers. Visualization tools include a bird's-eye-view plot and scope for sensor coverage, detections and tracks, and displays for video, lidar, and maps. The toolbox lets you import and work with HERE HD Live Map data and OpenDRIVE road networks.

Using the Ground Truth Labeler app, you can automate the labeling of ground truth to train and evaluate perception algorithms. For hardware-in-the-loop (HIL) testing and desktop simulation of perception, sensor fusion, path planning, and control logic, you can generate and simulate driving scenarios. You can simulate camera, radar, and lidar sensor output in a photorealistic 3D environment and sensor detections of objects and lane boundaries in a 2.5D simulation environment.

Automated Driving Toolbox provides reference application examples for common ADAS and automated driving features, including FCW, AEB, ACC, LKA, and parking valet. The toolbox supports C/C++ code generation for rapid prototyping and HIL testing, with support for sensor fusion, tracking, path planning, and vehicle controller algorithms.

CHAPTER 5
SIMULATION

5.1. POSSIBLE TYPES OF SENSOR FUSION MODULES

Fusion of Vision Sensor (camera sensor) data and RADAR/LIDAR data can give a better understanding about the obstacles to the vehicle along the path and can be useful to differentiate necessary and unnecessary object detections.

Fusion of data from Inertial Measurement Unit (IMU) can give a better understanding about the position and orientation of the vehicle.
Fusion of GPS data and Inertial Measurement Unit (IMU) data gives a better understanding on the geo-position, spacial-position and orientation of the vehicle.
To meet our objective algorithm to fuse the RADAR and Vision Sensor data is designed in MATLAB.

5.2. SIMULATION IN MATLAB

MATLAB is a platform that provides tools to learn and work on various branches of study.The main advantage of MATLAB is that it provides tools for simulation.The Automated Driving Toolbox, Sensor Fusion Toolbox and Computer Vision Toolbox in MATLAB provide tools for designing and testing the sensor fusion algorithms.

- The Sensor Fusion Toolbox provides tools to fuse RADAR data and VISION data.
- The Automated Driving Toolbox provides tools to track multiple object detections,.
- The Computer Vision Toolbox provides tools to perform cluster removal, predictions and approximations based on the available information and conditions.
-

5.3. DESIGN OF THE MODULE

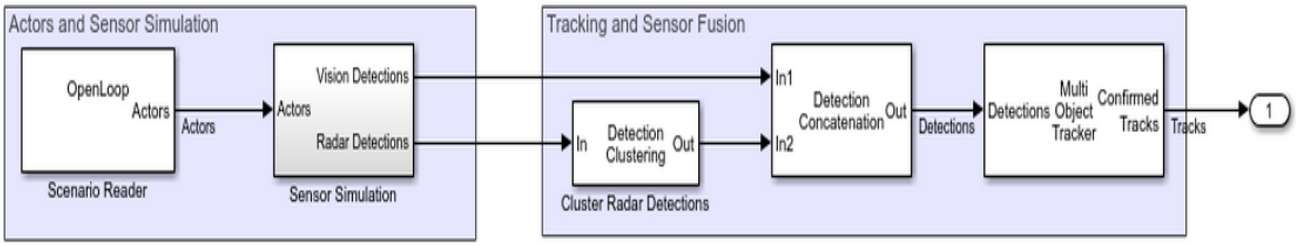


Figure.5.1.DesignFlow

The objectives of the design process are:

1. Obtain the data from the sensors.
2. Fuse the sensor data to get a list of tracks, i.e., estimated positions and velocities of the objects in front of the car.
3. Issue warnings based on the tracks and Forward Collision Warning (FCW) criteria.

The FCW criteria are based on the Euro NCAP AEB test procedure and take into account the relative distance and relative speed to the object in front of the car.

5.3.1. FUNCTION TO READ THE DATA FROM SENSORS

The first step in the sensor fusion process is to read the data from RADAR and VISION sensors. For proper fusion of data from both the sensors it is necessary to have data from both the sensors at the same time. The following function reads the data from the sensor recordings file and arranges them based on the time of the reading. The data from the sensors for every 50 milliseconds is noted.

MATLAB CODE

```
function [visionObjects, radarObjects,laneReports, ...
    timeStep, numSteps] =
readSensorRecordingsFile(sensorRecordingFileName)
% Read Sensor Recordings
% The |ReadDetectionsFile| function reads the recorded sensor data file.
```

```
% The recorded data is a single structure that is divided into the
% following substructures:
% # |laneReports|,
% # |radarObjects|,
% # |visionObjects
A = load(sensorRecordingFileName);
visionObjects = A.vision;
radarObjects = A.radar;
laneReports = A.lane;
timeStep = 0.05;                % Data is provided every 50 milliseconds
numSteps = numel(visionObjects); % Number of recorded timesteps
end
```

5.3.2. CREATION OF A MULTI-OBJECT TRACKER

The multiObjectTracker tracks the objects around the ego vehicle based on the object lists reported by the vision and radar sensors. By fusing information from both sensors, the probability of a false collision warning is reduced.

The setupTracker function returns the multiObjectTracker. When creating a multiObjectTracker, consider the following:

1. FilterInitializationFcn: The likely motion and measurement models. In this case, the objects are expected to have a constant acceleration motion. Although you can configure a linear Kalman filter for this model, initConstantAccelerationFilter configures an extended Kalman filter. See the 'Define a Kalman filter' section.
2. AssignmentThreshold: How far detections can fall from tracks. The default value for this parameter is 30. If there are detections that are not assigned to tracks, but should be, increase this value. If there are detections that get assigned to tracks that are too far, decrease this value. This example uses 35.
3. DeletionThreshold: When a track is confirmed, it should not be deleted on the first update that no detection is assigned to it. Instead, it should be coasted (predicted) until it is clear that the track is not getting any sensor information to update it. The logic is that if the track is missed *P*-out-of-_Q_

times it shold be deleted. The default value for this parameter is 5-out-of-5. In this case, the tracker is called 20 times a second and there are two sensors, so there is no need to modify the default.

4. ConfirmationThreshold: The parameters for confirming a track. A new track is initialized with every unassigned detection. Some of these detections might be false, so all the tracks are initialized as 'Tentative'. To confirm a track, it has to be detected at least *M* times in *N* tracker updates. The choice of *M* and *N* depends on the visibility of the objects. This example uses the default of 2 detections out of 3 updates.

The outputs of setupTracker are:

- tracker - The multiObjectTracker that is configured for this case.
- positionSelector - A matrix that specifies which elements of the State vector are the position: position = positionSelector * State
- velocitySelector - A matrix that specifies which elements of the State vector are the velocity: velocity = velocitySelector * State

MATLAB CODE

```
function [tracker, positionSelector, velocitySelector] = setupTracker()

    tracker = multiObjectTracker(...
        'FilterInitializationFcn', @initConstantAccelerationFilter, ...
        'AssignmentThreshold', 35, 'ConfirmationThreshold', [2 3], ...
        'DeletionThreshold', 5);


    % The State vector is:
    %   In constant velocity:     State = [x;vx;y;vy]
    %   In constant acceleration: State = [x;vx;ax;y;vy;ay]

    % Define which part of the State is the position. For example:
    %   In constant velocity:     [x;y] = [1 0 0 0; 0 0 1 0] * State
    %   In constant acceleration: [x;y] = [1 0 0 0 0 0; 0 0 0 1 0 0] * State
    positionSelector = [1 0 0 0 0 0; 0 0 0 1 0 0];
```

```
        % Define which part of the State is the velocity. For example:
        %   In constant velocity:      [x;y] = [0 1 0 0; 0 0 0 1] * State
        %   In constant acceleration: [x;y] = [0 1 0 0 0 0; 0 0 0 0 1 0] * State
        velocitySelector = [0 1 0 0 0 0; 0 0 0 0 1 0];
    end
```

The multiObjectTracker object has the following properties.

Property	Description
FilterInitializationFcn	Kalman filter initialization function @initcvkf (default) | function handle | character vector | string scalar
AssignmentThreshold	Detection assignment threshold 30 (default) | positive real scalar
ConfirmationParameters	Confirmation parameters for track creation [2 3] (default) | two-element vector of positive increasing integers
NumCoastingUpdates	Coasting threshold for track deletion 5 (default) | positive integer
MaxNumTracks	Maximum number of tracks 200 (default) | positive integer
MaxNumSensors	Maximum number of sensors 20 (default) | positive integer
HasCostMatrixInput	Enable cost matrix input false (default) | true
NumTracks	Number of tracks maintained by multi-object tracker nonnegative integer
NumConfirmedTracks	Number of confirmed tracks nonnegative integer

Table.5.1. Properties of multiObjectTracker

5.3.3. DEFINE A KALMAN FILTER

The sensor data might not be true each and every time. Errors may occur in the readings due to various noises in the nature.To reduce these errors, Kalman filter is used. The Kalman Filter predicts the future value based on the present and previous values using a mathematical model.

The multiObjectTracker defined in the previous section uses the filter initialization function defined in this section to create a Kalman filter (linear, extended, or unscented). This filter is then used for tracking each object around the ego vehicle.

MATLAB CODE

```
function filter = initConstantAccelerationFilter(detection)

% Steps for creating a filter:
%   1. Define the motion model and state
%   2. Define the process noise
%   3. Define the measurement model
%   4. Initialize the state vector based on the measurement
%   5. Initialize the state covariance based on the measurement noise
%   6. Create the correct filter

    % Step 1: Define the motion model and state
    STF = @constacc;     % State-transition function, for EKF and UKF
    STFJ = @constaccjac; % State-transition function Jacobian, only for EKF
    % The motion model implies that the state is [x;vx;ax;y;vy;ay]

    % Step 2: Define the process noise
    dt = 0.05; % Known timestep size
    sigma = 1; % Magnitude of the unknown acceleration change rate
    % The process noise along one dimension
    Q1d = [dt^4/4, dt^3/2, dt^2/2; dt^3/2, dt^2, dt; dt^2/2, dt, 1] * sigma^2;
    Q = blkdiag(Q1d, Q1d); % 2-D process noise

    % Step 3: Define the measurement model
    MF = @fcwmeas;       % Measurement function, for EKF and UKF
    MJF = @fcwmeasjac;   % Measurement Jacobian function, only for EKF

    % Step 4: Initialize a state vector based on the measurement
    % The sensors measure [x;vx;y;vy] and the constant acceleration model's
```

```
    % state is [x;vx;ax;y;vy;ay], so the third and sixth elements of the
    % state vector are initialized to zero.
    state = [detection.Measurement(1); detection.Measurement(2); 0;
detection.Measurement(3); detection.Measurement(4); 0];

    % Step 5: Initialize the state covariance based on the measurement
    % noise. The parts of the state that are not directly measured are
    % assigned a large measurement noise value to account for that.
    L = 100; % A large number relative to the measurement noise
    stateCov = blkdiag(detection.MeasurementNoise(1:2,1:2), L,
detection.MeasurementNoise(3:4,3:4), L);

    % Step 6: Create the correct filter.
    % Use 'KF' for trackingKF, 'EKF' for trackingEKF, or 'UKF' for trackingUKF
    FilterType = 'EKF';

    % Creating the filter:
    switch FilterType
        case 'EKF'
            filter = trackingEKF(STF, MF, state,...
                'StateCovariance', stateCov, ...
                'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
                'StateTransitionJacobianFcn', STFJ, ...
                'MeasurementJacobianFcn', MJF, ...
                'ProcessNoise', Q ...
                );
        case 'UKF'
            filter = trackingUKF(STF, MF, state, ...
                'StateCovariance', stateCov, ...
                'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
                'Alpha', 1e-1, ...
                'ProcessNoise', Q ...
                );
        case 'KF' % The ConstantAcceleration model is linear and KF can be used
```

```
        % Define the measurement model: measurement = H * state
        H = [1 0 0 0 0 0; 0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 1 0];
        filter = trackingKF('MotionModel', '2D Constant Acceleration', ...
            'MeasurementModel', H, 'State', state, ...
            'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
            'StateCovariance', stateCov);
    end
end
```

The trackingEKF class available in Sensor Fusion and Tracking Toolbox is used create discrete-time Extended Kalman Filter. The object created with trackingEKF class has the following arguments:

Argument	**Description**
State	Kalman filter state real-valued *M*-element vector
StateCovariance	State estimation error covariance positive-definite real-valued *M*-by-*M* matrix
StateTransitionFcn	State transition function function handle
StateTransitionJacobianFcn	Jacobian of state transition function function handle
ProcessNoise	Process noise covariance 1 (default) | positive real scalar | positive-definite real-valued matrix
HasAdditiveProcessNoise	Model additive process noise true (default) | false
MeasurementFcn	Measurement model function function handle
MeasurementJacobianFcn	Jacobian of measurement function function handle
MeasurementNoise	Measurement noise covariance 1 (default) | positive scalar | positive-definite real-valued matrix
HasAdditiveMeasurmentNoise	Model additive measurement noise true (default) | false

Table.5.2.Properties of trackingEKF object

OBJECT FUNCTIONS

The following are the functions of trackingEKF object

Predict	Predict state and state estimation error covariance of linear Kalman filter
Correct	Correct state and state estimation error covariance using tracking filter
Correctjpda	Correct state and state estimation error covariance using tracking filter and JPDA
Distance	Distances between current and predicted measurements of tracking filter
Likelihood	Likelihood of measurement from tracking filter
Clone	Create duplicate tracking filter
Residual	Measurement residual and residual noise from tracking filter
Initialize	Initialize state and covariance of tracking filter

Table.5.3.Functions of trackingEKF object

The trackingKF class available in Sensor Fusion and Tracking Toolbox is used create discrete-time Kalman Filter. The object created with trackingKF class has the following arguments:

Argument	**Description**
State	Kalman filter state real-valued M-element vector
StateCovariance	State estimation error covariance positive-definite real-valued M-by-M matrix
StateTransitionModel	State transition model between time steps [1 1 0 0; 0 1 0 0; 0 0 1 1; 0 0 0 1] (default) | real-valued M-by-M matrix
MotionModel	Kalman filter motion model 'Custom' (default) | '1D Constant Velocity' | '2D Constant Velocity' | '3D Constant Velocity' | '1D Constant Acceleration' | '2D Constant Acceleration'
ProcessNoise	Process noise covariance 1 (default) | positive real scalar | positive-definite real-valued matrix

ControlModel	Control model [] (default) | M-by-L real-valued matrix
MeasurementModel	Measurement model from state vector [1 0 0 0; 0 0 1 0] (default) | real-valued N-by-M matrix
MeasurementNoise	Measurement noise covariance 1 (default) | positive scalar | positive-definite real-valued matrix

Table.5.4.Properties of trackingKF object

OBJECT FUNCTIONS

The following are the functions of trackingKF object

Predict	Predict state and state estimation error covariance of linear Kalman filter
Correct	Correct state and state estimation error covariance using tracking filter
Correctjpda	Correct state and state estimation error covariance using tracking filter and JPDA
Distance	Distances between current and predicted measurements of tracking filter
Likelihood	Likelihood of measurement from tracking filter
Clone	Create duplicate tracking filter
Residual	Measurement residual and residual noise from tracking filter
Initialize	Initialize state and covariance of tracking filter

Table.5.5.Functions of trackingKF object

5.3.4. FUNCTION TO FIND NON CLUSTER RADAR OBJECTS

Clutter is defined as a stationary object that is not in front of the car. The following types of objects pass as nonclutter:

- Any object in front of the car
- Any moving object in the area of interest around the car, including objects that move at a lateral speed around the car

For the accuracy and proper functioning of the sensor fusion module, the clusters must be detected and processed. The following function detects clusters and processed.

MATLAB CODE

```
function realRadarObjects = findNonClutterRadarObjects(radarObject,
numRadarObjects, egoSpeed, laneBoundaries)
% The radar objects include many objects that belong to the clutter.
    % Allocate memory
    normVs = zeros(numRadarObjects, 1);
    inLane = zeros(numRadarObjects, 1);
    inZone = zeros(numRadarObjects, 1);
    % Parameters
    LaneWidth = 3.6;            % What is considered in front of the car
    ZoneWidth = 1.7*LaneWidth;  % A wider area of interest
    minV = 1;                   % Any object that moves slower than minV is
considered stationary
    for j = 1:numRadarObjects
        [vx, vy] =
calculateGroundSpeed(radarObject(j).velocity(1),radarObject(j).velocity(2),egoS
peed);
        normVs(j) = norm([vx,vy]);
        laneBoundariesAtObject = computeBoundaryModel(laneBoundaries,
radarObject(j).position(1));
        laneCenter = mean(laneBoundariesAtObject);
        inLane(j) = (abs(radarObject(j).position(2) - laneCenter) <= LaneWidth/2);
        inZone(j) = (abs(radarObject(j).position(2) - laneCenter) <= max(abs(vy)*2,
ZoneWidth));
    end
    realRadarObjectsIdx = union(...
        intersect(find(normVs > minV), find(inZone == 1)), ...
        find(inLane == 1));
    realRadarObjects = radarObject(realRadarObjectsIdx);
end
```

5.3.5. FUNCTION TO PROCESS AND FORMAT THE DETECTIONS

The recorded information must be processed and formatted before it can be used by the tracker. This has the following steps:

1. Filtering out unnecessary radar clutter detections. The radar reports many objects that correspond to fixed objects, which include: guard-rails, the road median, traffic signs, etc. If these detections are used in the tracking, they create false tracks of fixed objects at the edges of the road and therefore must be removed before calling the tracker. Radar objects are considered nonclutter if they are either stationary in front of the car or moving in its vicinity.
2. Formatting the detections as input to the tracker, i.e., an array of objectDetection elements.

MATLAB CODE

```
function [detections,laneBoundaries, egoLane] = processDetections...
        (visionFrame, radarFrame, laneFrame, egoLane, time)
    % Inputs:
    %   visionFrame - objects reported by the vision sensor for this time
frame
    %   radarFrame  - objects reported by the radar sensor for this time
frame
    %   laneFrame   - lane reports for this time frame
    %   egoLane     - the estimated ego lane
    %   time        - the time corresponding to the time frame

    % Remove clutter radar objects
    [laneBoundaries, egoLane] = processLanes(laneFrame, egoLane);
    realRadarObjects = findNonClutterRadarObjects(radarFrame.object,...
        radarFrame.numObjects, IMUFrame.velocity, laneBoundaries);
    % Return an empty list if no objects are reported

    % Counting the total number of objects
```

```
    detections = {};
    if (visionFrame.numObjects + numel(realRadarObjects)) == 0
        return;
    end
    % Process the remaining radar objects
    detections = processRadar(detections, realRadarObjects, time);
    % Process video objects
    detections = processVideo(detections, visionFrame, time);
end
```

The additional functions used in the above the above function are listed below:

processLanes Converts sensor reportedlanes to parabolicLaneBoundary lanes and maintains a persistent ego lane estimate

calculateGroundSpeed Calculates the true ground speed of a radar-reported object from the relative speed and the ego vehicle speed

processRadar Converts reported radar objects into list of objectDetection objects

fcwmeas The measurement function used in this forward collision warning

fcwmeasjac The Jacobian of the measurement function used in this forward collision warning

The MATLAB code of above functions is included in the appendix for proper formatting of the report.

5.3.6. FUNCTION TO DETECT MOST IMPORTANT OBJECT

The most important object (MIO) is defined as the track that is in the ego lane and is closest in front of the car, i.e., with the smallest positive x value. To lower the probability of false alarms, only confirmed tracks are considered.

Once the MIO is found, the relative speed between the car and MIO is calculated. The relative distance and relative speed determine the forward collision warning. There are 3 cases of FCW:

1. Safe (green): There is no car in the ego lane (no MIO), the MIO is moving away from the car, or the distance to the MIO remains constant.
2. Caution (yellow): The MIO is moving closer to the car, but is still at a distance above the FCW distance. FCW distance is calculated using the Euro NCAP AEB Test Protocol. Note that this distance varies with the relative speed between the MIO and the car, and is greater when the closing speed is higher.
3. Warn (red): The MIO is moving closer to the car, and its distance is less than the FCW distance, d_{FCW}.

Euro NCAP AEB Test Protocol defines the following distance calculation:

$$d_{FCW} = 1.2 * v_{rel} + \frac{v_{rel}^2}{2a_{max}}$$

where:

d_{FCW} is the forward collision warning distance.

v_{rel} is the relative velocity between the two vehicles.

a_{max} is the maximum deceleration, defined to be 40% of the gravity acceleration.

MATLAB CODE

```
function mostImportantObject =
findMostImportantObject(confirmedTracks,egoLane,positionSelector,velocitySe
lector)
    % Initialize outputs and parameters
    MIO = [];              % By default, there is no MIO
    trackID = [];          % By default, there is no trackID associated with an
MIO
    FCW = 3;               % By default, if there is no MIO, then FCW is 'safe'
    threatColor = 'green';  % By default, the threat color is green
    maxX = 1000;  % Far enough forward so that no track is expected to
exceed this distance
    gAccel = 9.8; % Constant gravity acceleration, in m/s^2
    maxDeceleration = 0.4 * gAccel; % Euro NCAP AEB definition
    delayTime = 1.2; % Delay time for a driver before starting to brake, in
seconds

    positions = getTrackPositions(confirmedTracks, positionSelector);
    velocities = getTrackVelocities(confirmedTracks, velocitySelector);

    for i = 1:numel(confirmedTracks)
        x = positions(i,1);
        y = positions(i,2);
        relSpeed = velocities(i,1); % The relative speed between the cars, along
the lane
        if x < maxX && x > 0 % No point checking otherwise
            yleftLane  = polyval(egoLane.left,  x);
            yrightLane = polyval(egoLane.right, x);
            if (yrightLane <= y) && (y <= yleftLane)
                maxX = x;
                trackID = i;
                MIO = confirmedTracks(i).TrackID;
                if relSpeed < 0 % Relative speed indicates object is getting closer
                    % Calculate expected braking distance according to
```

```
                % Euro NCAP AEB Test Protocol
                d = abs(relSpeed) * delayTime + relSpeed^2 / 2 /
maxDeceleration;
                if x <= d % 'warn'
                    FCW = 1;
                    threatColor = 'red';
                else % 'caution'
                    FCW = 2;
                    threatColor = 'yellow';
                end
            end
        end
    end
    mostImportantObject = struct('ObjectID', MIO, 'TrackIndex', trackID,
'Warning', FCW, 'ThreatColor', threatColor);
  end
```

5.4. TESTING

For the purpose of testing the algorithm, RADAR data and VISION data provided by MATLAB are used.

These data are obtained in the following way.

A test car (the ego vehicle) equipped with various sensors is simulated and the sensors outputs are recorded. The sensors used are:

1. Vision sensor, which provided lists of observed objects with their classification and information about lane boundaries. The object lists were reported 10 times per second. Lane boundaries were reported 20 times per second.
2. LIDAR sensor with medium and long range modes, which provided lists of unclassified observed objects. The object lists were reported 20 times per second.

3. Video camera, which recorded a video clip of the scene in front of the car. Note: This video is not used by the tracker and only serves to display the tracking results on video for verification.

5.5. VISUALIZATION

The visualization in the algorithm is done using **monoCamera** and **birdsEyePlot**.

monoCamera

The monoCamera object holds information about the configuration of a monocular camera sensor. Configuration information includes the camera intrinsics, camera extrinsics such as its orientation (as described by pitch, yaw, and roll), and the camera location within the vehicle.

For images captured by the camera, you can use the imageToVehicle and vehicleToImage functions to transform point locations between image coordinates and vehicle coordinates. These functions apply projective transformations (homography), which enable you to estimate distances from a camera mounted on the vehicle to locations on a flat road surface.

SYNTAX

sensor = monoCamera(intrinsics,height)
sensor = monoCamera(intrinsics,height,Name,Value)

DESCRIPTION

sensor = monoCamera(intrinsics,height) creates a monoCamera object that contains the configuration of a monocular camera sensor, given the intrinsic parameters of the camera and the height of the camera above the ground. intrinsics and height set the Intrinsics and Height properties of the camera.

sensor = monoCamera(intrinsics,height,Name,Value) sets properties using one or more name-value pairs.

For example, monoCamera(intrinsics,1.5,'Pitch',1) creates a monocular camera sensor that is 1.5 meters above the ground and has a 1-degree pitch toward the ground.

The monoCamera object has the following properties:

Property	**Description**
Intrinsics	Intrinsic camera parameters cameraIntrinsics object | cameraParameters object
Height	Height from road surface to camera sensor real scalar
Pitch	Pitch angle real scalar
Yaw	Yaw angle real scalar
Roll	Roll angle real scalar
SensorLocation	Location of center of camera sensor [0 0] (default) | two-element vector
WorldUnits	World coordinate system units 'meters' | character vector | string scalar

Table.5.6.Properties of monoCamera object

OBJECT FUNCTIONS

imageToVehicle	Convert image coordinates to vehicle coordinates
vehicleToImage	Convert vehicle coordinates to image coordinates

Table.5.3.Functions of monoCamera object

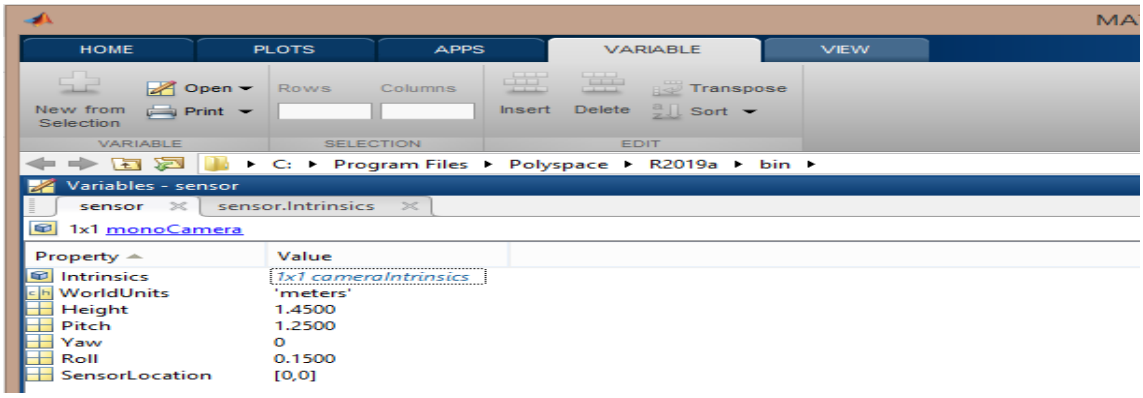


Figure.5.2.Variables of monoCamera object -1

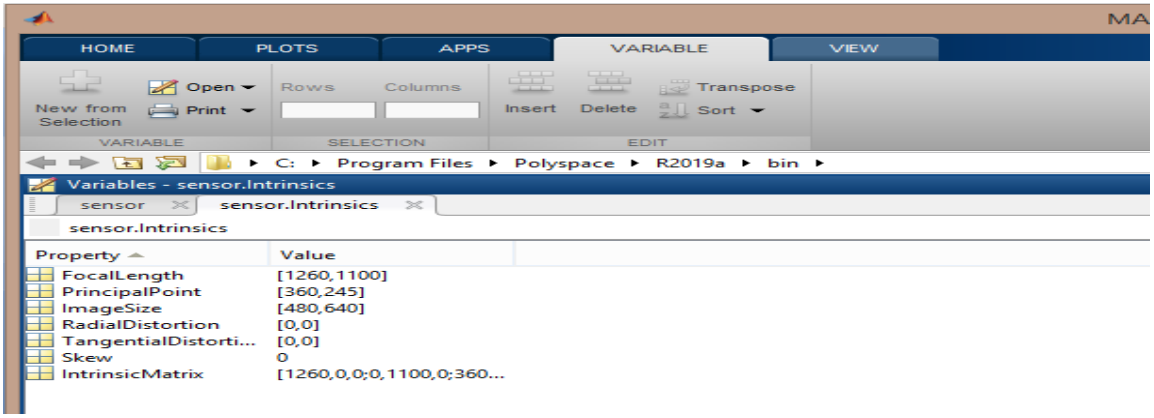


Figure.5.3.Variables of monoCamera object -2

birdsEyePlot

The birdsEyePlot object displays a bird's-eye plot of a 2-D driving scenario in the immediate vicinity of an ego vehicle. You can use this plot with sensors capable of detecting objects and lanes.

To display aspects of a driving scenario on a bird's-eye plot:

1. Create a birdsEyePlot object.
2. Create plotters for the aspects of the driving scenario that you want to plot.
3. Use the plotters with their corresponding plot functions to display those aspects on the bird's-eye plot.

This table shows the plotter functions to use based on the driving scenario aspect that you want to plot.

Driving Scenario Aspect to Plot	Plotter Creation Function	Plotter Display Function
Sensor coverage areas	coverageAreaPlotter	plotCoverageArea
Sensor detections	detectionPlotter	plotDetection
Lane boundaries	laneBoundaryPlotter	plotLaneBoundary
Lane markings	laneMarkingPlotter	plotLaneMarking
Object outlines	outlinePlotter	plotOutline

Driving Scenario Aspect to Plot	Plotter Creation Function	Plotter Display Function
Ego vehicle path	pathPlotter	plotPath
Point cloud	pointCloudPlotter	plotPointCloud
Object tracking results	trackPlotter	plotTrack

Table.5.8.BirdsEyePlotter Functions

SYNTAX

bep = birdsEyePlot

bep = birdsEyePlot(Name,Value)

DESCRIPTION

bep = birdsEyePlot creates a bird's-eye plot in a new figure.

bep = birdsEyePlot(Name,Value) sets properties using one or more Name,Value pair arguments. For example, birdsEyePlot('XLimits',[0 60],'YLimits',[-20 20]) displays the area that is 60 meters in front of the ego vehicle and 20 meters to either side of the ego vehicle. Enclose each property name in quotes.

The birdsEyePlot has the following properties:

Property	Description
Parent	Axes on which to plot axes handle
Plotters	Plotters created for bird's-eye plot array of plotter objects
XLimits	XLimits — X-axis range real-valued vector of the form [Xmin Xmax]
YLimits	YLimits — Y-axis range real-valued vector of the form [Ymin Ymax]

Table.5.9.Properties of birdsEyePlot object

OBJECT FUNCTIONS

Plotter Creation

coverageAreaPlotter	Coverage area plotter for bird's-eye plot
detectionPlotter	Detection plotter for bird's-eye plot
laneBoundaryPlotter	Lane boundary plotter for bird's-eye plot
laneMarkingPlotter	Lane marking plotter for bird's-eye plot
outlinePlotter	Outline plotter for bird's-eye plot
pathPlotter	Path plotter for bird's-eye plot
pointCloudPlotter	Point cloud plotter for bird's-eye plot
trackPlotter	Track plotter for bird's-eye plot

Table.5.10.Functions of Plotter Creation object

Plotter Display

plotCoverageArea	Display sensor coverage area on bird's-eye plot
plotDetection	Display object detections on bird's-eye plot
plotLaneBoundary	Display lane boundaries on bird’s-eye plot
plotLaneMarking	Display lane markings on bird’s-eye plot
plotOutline	Display object outlines on bird's-eye plot
plotPath	Display actor paths on bird’s-eye plot
plotPointCloud	Display generated point clouds on bird's-eye plot
plotTrack	Display object tracks on bird's-eye plot

Table.5.11.Functions of Plotter Display object

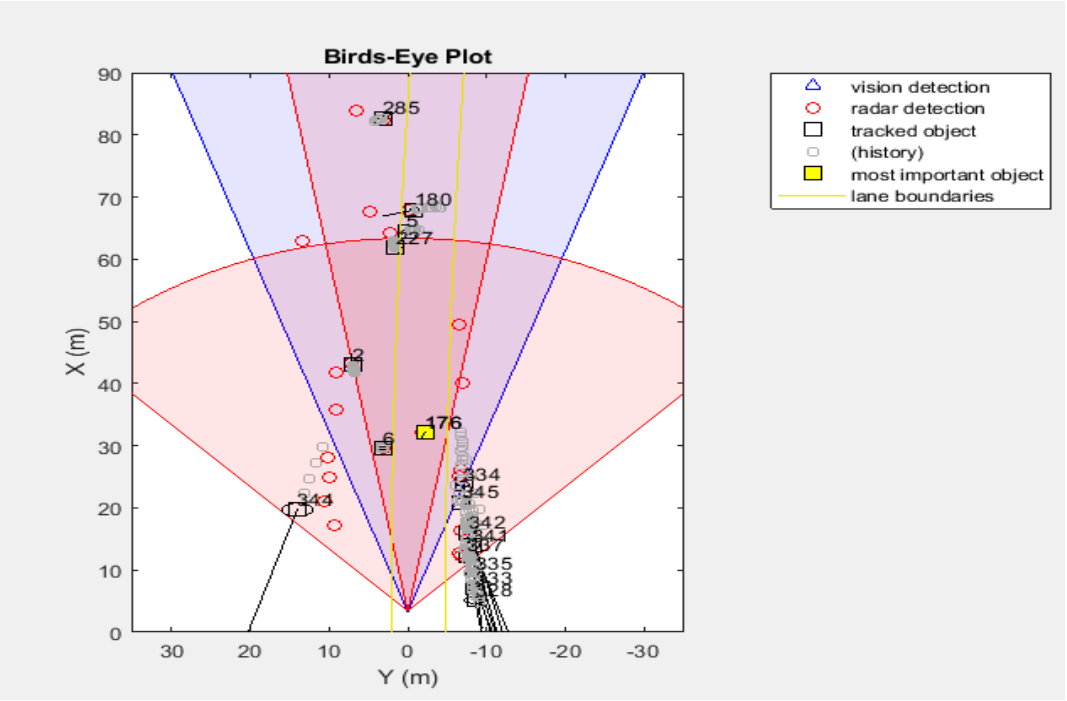


Figure.5.4. Birds-Eye Plot

5.6. MATLAB CODE

The complete MATLAB code for designing and testing the Sensor Fusion Module to detect the most important objects is included in the appendix for proper formatting of the report.

CHAPTER 6

RESULTS

The Sensor Fusion Module designed takes the RADAR sensor data and VISION sensor data as inputs and fuses the RADAR data and VISION data to provide warning regarding the position of the front vehicle with in the lane.

RADAR data and VISION data provided by MATLAB are used for testing the Sensor Fusion Module.

In the process of Sensor Fusion, initially the RADAR data and VISION data are loaded to the system. From the RADAR data clusters are removed and treated as a single object. Then the VISION data is used to relate the RADAR detections with the VISION detections. Trackers are assigned to each of the object detections to keep track of the detections. For assigning tracks to the object detections Cost matrix is used. An object in the lane of the vehicle is treated as the most important object. Warining are raised based on the distance between the vehicle and the most important object.

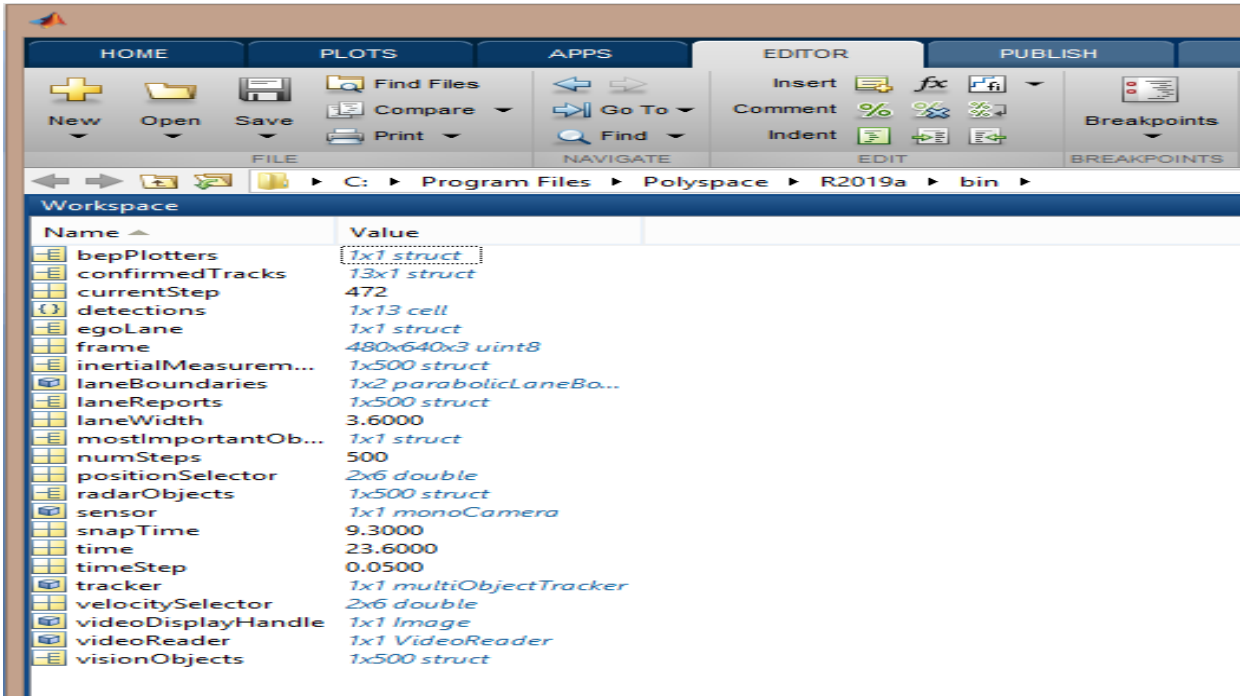
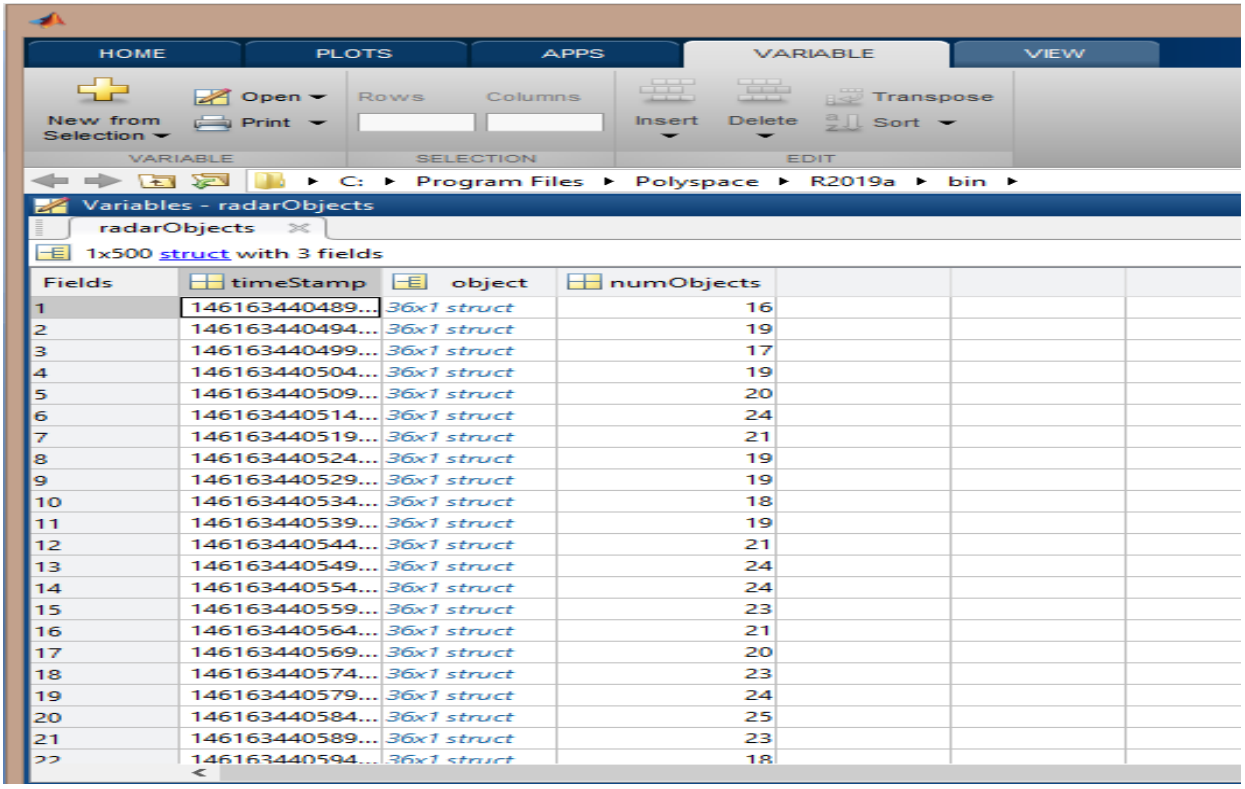
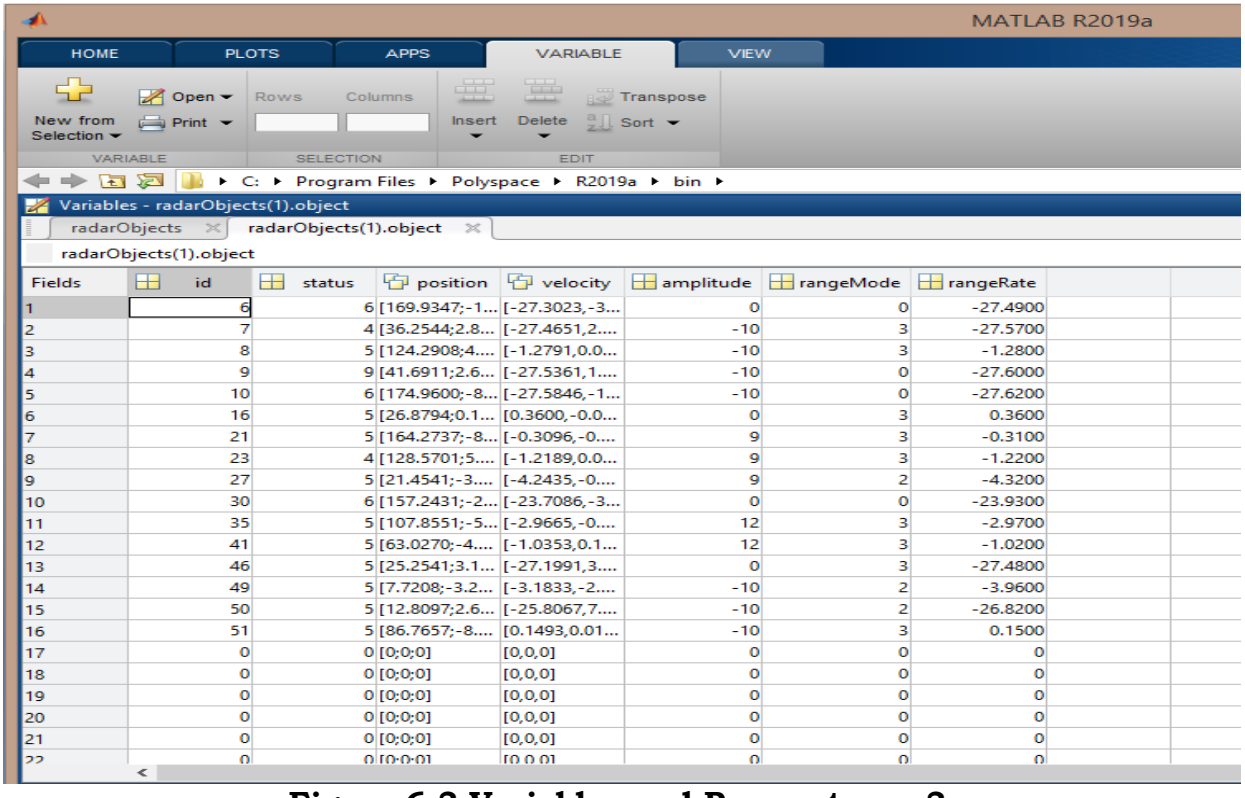


Figure.6.1.Variables and Parameters - 1



Fields	timeStamp	object	numObjects
1	146163440489...	36x1 struct	16
2	146163440494...	36x1 struct	19
3	146163440499...	36x1 struct	17
4	146163440504...	36x1 struct	19
5	146163440509...	36x1 struct	20
6	146163440514...	36x1 struct	24
7	146163440519...	36x1 struct	21
8	146163440524...	36x1 struct	19
9	146163440529...	36x1 struct	19
10	146163440534...	36x1 struct	18
11	146163440539...	36x1 struct	19
12	146163440544...	36x1 struct	21
13	146163440549...	36x1 struct	24
14	146163440554...	36x1 struct	24
15	146163440559...	36x1 struct	23
16	146163440564...	36x1 struct	21
17	146163440569...	36x1 struct	20
18	146163440574...	36x1 struct	23
19	146163440579...	36x1 struct	24
20	146163440584...	36x1 struct	25
21	146163440589...	36x1 struct	23
22	146163440594...	36x1 struct	18

Figure.6.2.Variables and Parameters - 2



Fields	id	status	position	velocity	amplitude	rangeMode	rangeRate
1	6	6	[169.9347;-1...	[-27.3023,-3...	0	0	-27.4900
2	7	4	[36.2544;2.8...	[-27.4651,2....	-10	3	-27.5700
3	8	5	[124.2908;4....	[-1.2791,0.0...	-10	3	-1.2800
4	9	9	[41.6911;2.6...	[-27.5361,1....	-10	0	-27.6000
5	10	6	[174.9600;-8...	[-27.5846,-1...	-10	0	-27.6200
6	16	5	[26.8794;0.1...	[0.3600,-0.0...	0	3	0.3600
7	21	5	[164.2737;-8...	[-0.3096,-0....	9	3	-0.3100
8	23	4	[128.5701;5....	[-1.2189,0.0...	9	3	-1.2200
9	27	5	[21.4541;-3....	[-4.2435,-0....	9	2	-4.3200
10	30	6	[157.2431;-2...	[-23.7086,-3...	0	0	-23.9300
11	35	5	[107.8551;-5...	[-2.9665,-0....	12	3	-2.9700
12	41	5	[63.0270;-4....	[-1.0353,0.1...	12	3	-1.0200
13	46	5	[25.2541;3.1...	[-27.1991,3....	0	3	-27.4800
14	49	5	[7.7208;-3.2...	[-3.1833,-2....	-10	2	-3.9600
15	50	5	[12.8097;2.6...	[-25.8067,7....	-10	2	-26.8200
16	51	5	[86.7657;-8....	[0.1493,0.01...	-10	3	0.1500
17	0	0	[0;0;0]	[0,0,0]	0	0	0
18	0	0	[0;0;0]	[0,0,0]	0	0	0
19	0	0	[0;0;0]	[0,0,0]	0	0	0
20	0	0	[0;0;0]	[0,0,0]	0	0	0
21	0	0	[0;0;0]	[0,0,0]	0	0	0
22	0	0	[0;0;0]	[0,0,0]	0	0	0

Figure.6.3.Variables and Parameters - 3

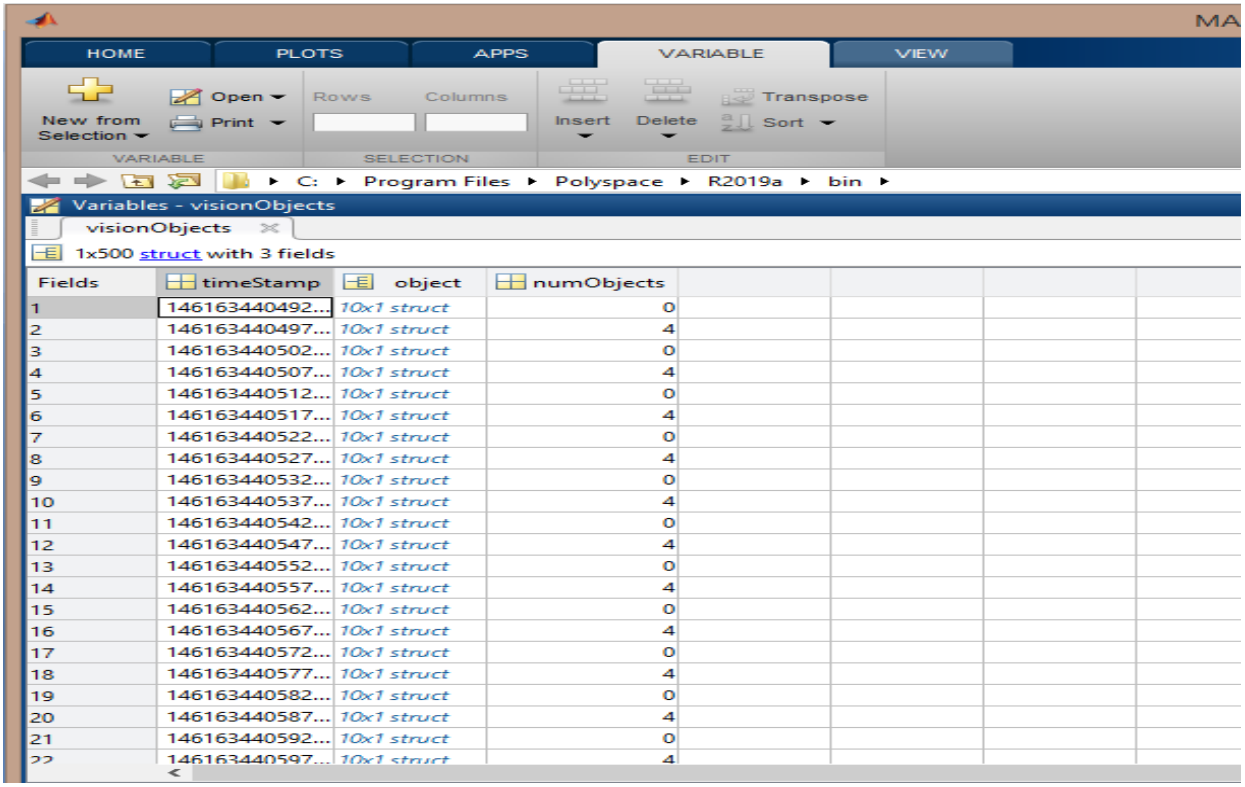


Figure.6.4.Variables and Parameters - 4

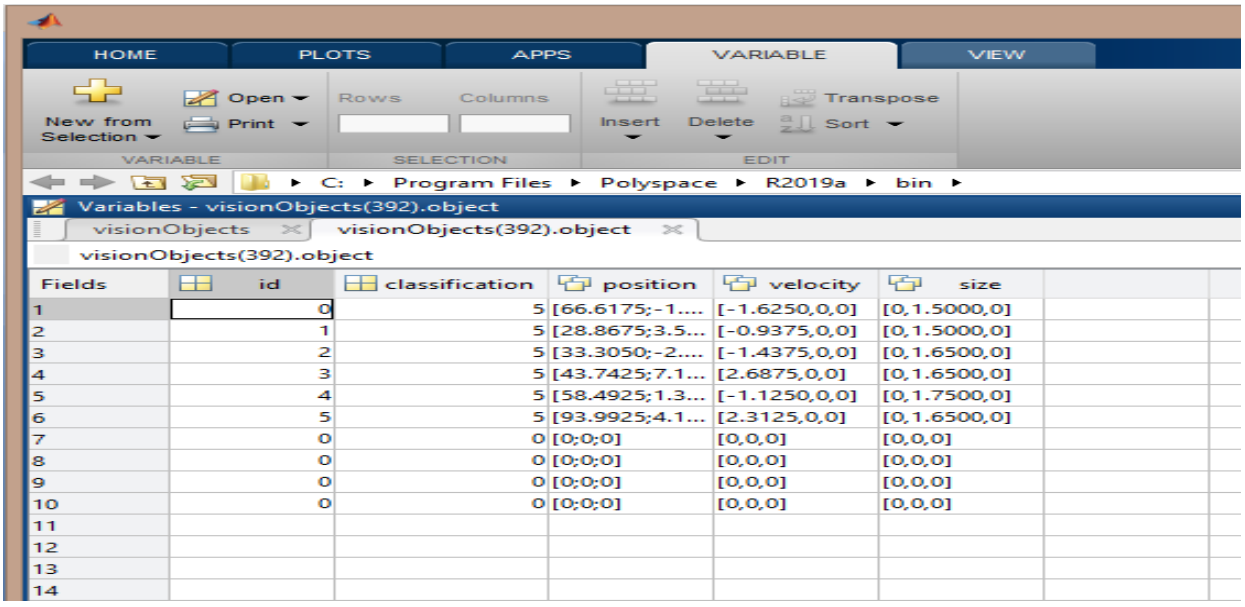
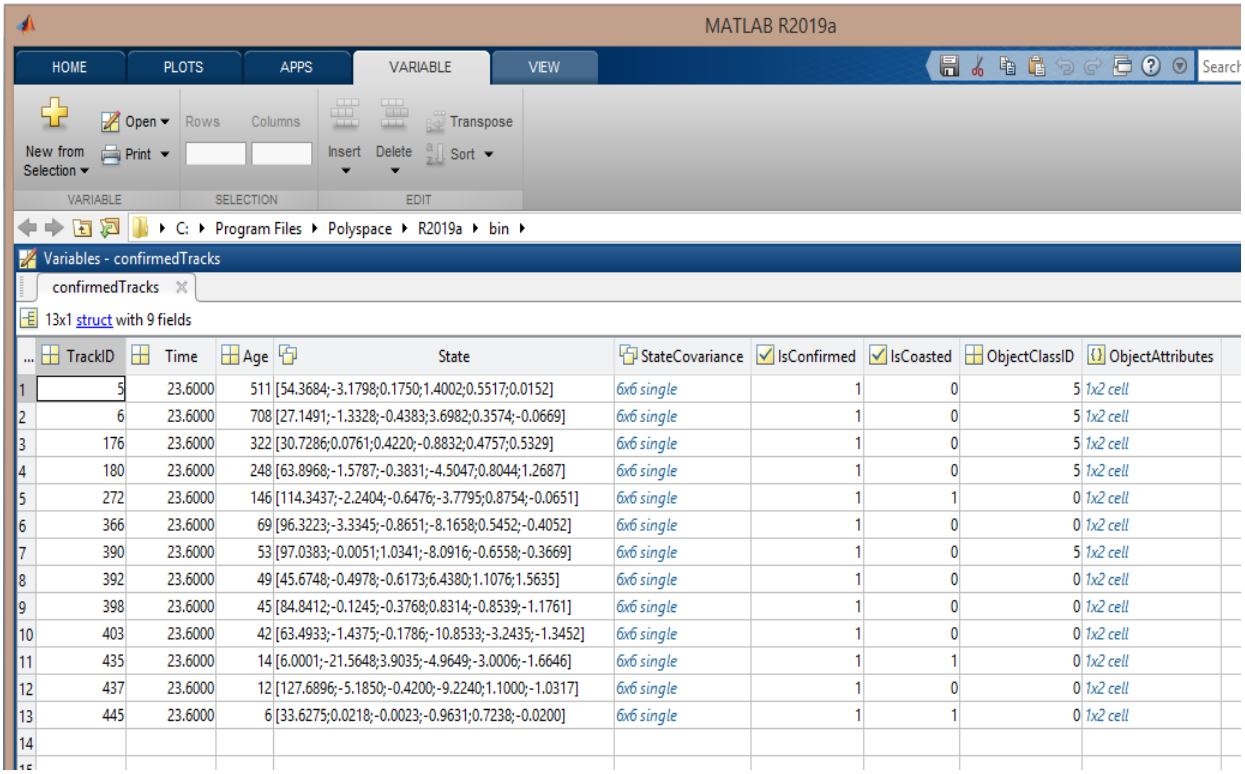
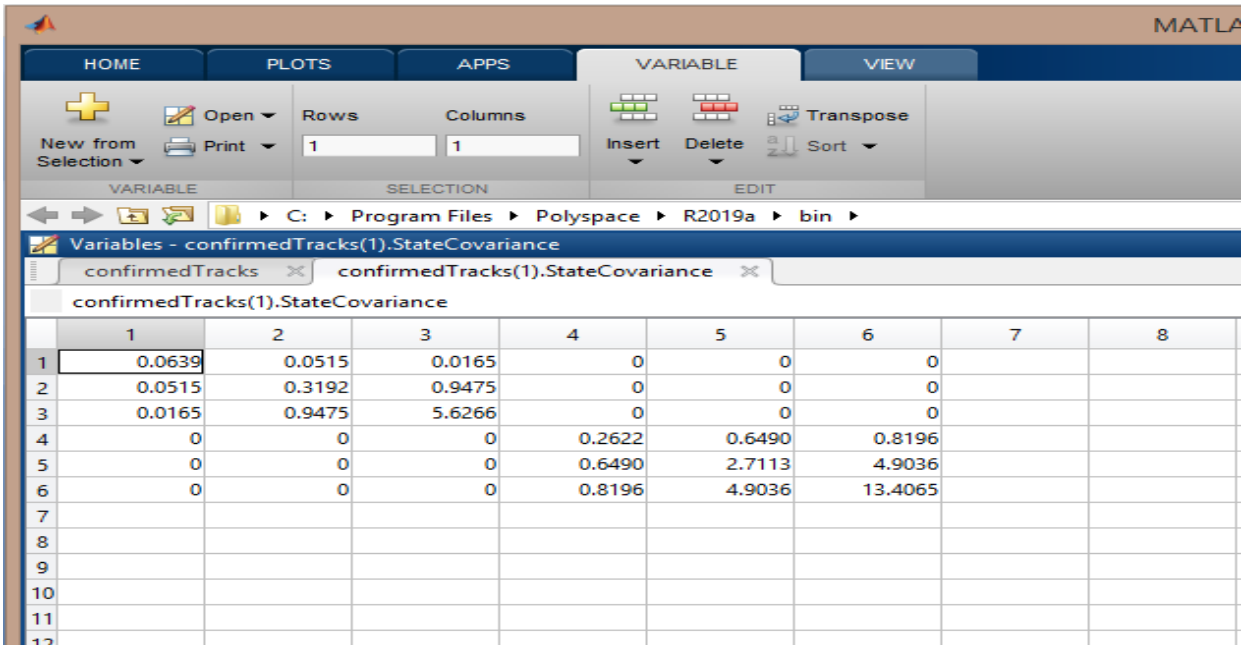


Figure.6.5.Variables and Parameters - 5



	TrackID	Time	Age	State	StateCovariance	IsConfirmed	IsCoasted	ObjectClassID	ObjectAttributes
1	5	23.6000	511	[54.3684;-3.1798;0.1750;1.4002;0.5517;0.0152]	6x6 single	1	0	5	1x2 cell
2	6	23.6000	708	[27.1491;-1.3328;-0.4383;3.6982;0.3574;-0.0669]	6x6 single	1	0	5	1x2 cell
3	176	23.6000	322	[30.7286;0.0761;0.4220;-0.8832;0.4757;0.5329]	6x6 single	1	0	5	1x2 cell
4	180	23.6000	248	[63.8968;-1.5787;-0.3831;-4.5047;0.8044;1.2687]	6x6 single	1	0	5	1x2 cell
5	272	23.6000	146	[114.3437;-2.2404;-0.6476;-3.7795;0.8754;-0.0651]	6x6 single	1	1	0	1x2 cell
6	366	23.6000	69	[96.3223;-3.3345;-0.8651;-8.1658;0.5452;-0.4052]	6x6 single	1	0	0	1x2 cell
7	390	23.6000	53	[97.0383;-0.0051;1.0341;-8.0916;-0.6558;-0.3669]	6x6 single	1	0	5	1x2 cell
8	392	23.6000	49	[45.6748;-0.4978;-0.6173;6.4380;1.1076;1.5635]	6x6 single	1	0	0	1x2 cell
9	398	23.6000	45	[84.8412;-0.1245;-0.3768;0.8314;-0.8539;-1.1761]	6x6 single	1	0	0	1x2 cell
10	403	23.6000	42	[63.4933;-1.4375;-0.1786;-10.8533;-3.2435;-1.3452]	6x6 single	1	0	0	1x2 cell
11	435	23.6000	14	[6.0001;-21.5648;3.9035;-4.9649;-3.0006;-1.6646]	6x6 single	1	1	0	1x2 cell
12	437	23.6000	12	[127.6896;-5.1850;-0.4200;-9.2240;1.1000;-1.0317]	6x6 single	1	0	0	1x2 cell
13	445	23.6000	6	[33.6275;0.0218;-0.0023;-0.9631;0.7238;-0.0200]	6x6 single	1	1	0	1x2 cell
14									

Figure.6.6.Variables and Parameters - 6



	1	2	3	4	5	6	7	8
1	0.0639	0.0515	0.0165	0	0	0		
2	0.0515	0.3192	0.9475	0	0	0		
3	0.0165	0.9475	5.6266	0	0	0		
4	0	0	0	0.2622	0.6490	0.8196		
5	0	0	0	0.6490	2.7113	4.9036		
6	0	0	0	0.8196	4.9036	13.4065		
7								
8								
9								
10								
11								

Figure.6.7.Variables and Parameters - 7

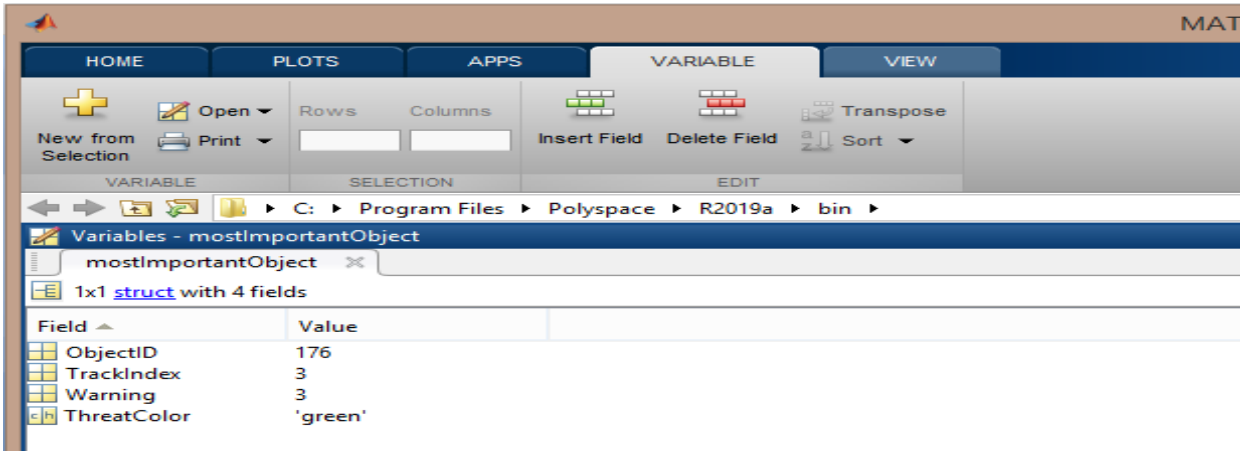


Figure.6.8.Variables and Parameters - 8

An object in the lane of the vehicle is treated as the most important object. Warining are raised based on the distance between the vehicle and the most important object.

- If the front vehicle in the lane of the Ego vehicle is at a safe distance from the ego vehicle, the track is indicated with GREEN color.
- If the front vehicle in the lane of the Ego vehicle is at a safe distance from the ego vehicle but if the relative speed is greater then the required, the track is indicated with YELLOW color.
- If the distance between front vehicle in the lane and the Ego vehicle is less than the safe distance, the track is indicated with RED color.

Figure 6.9. SNAPSHOT OF THE SIMULATION - 1

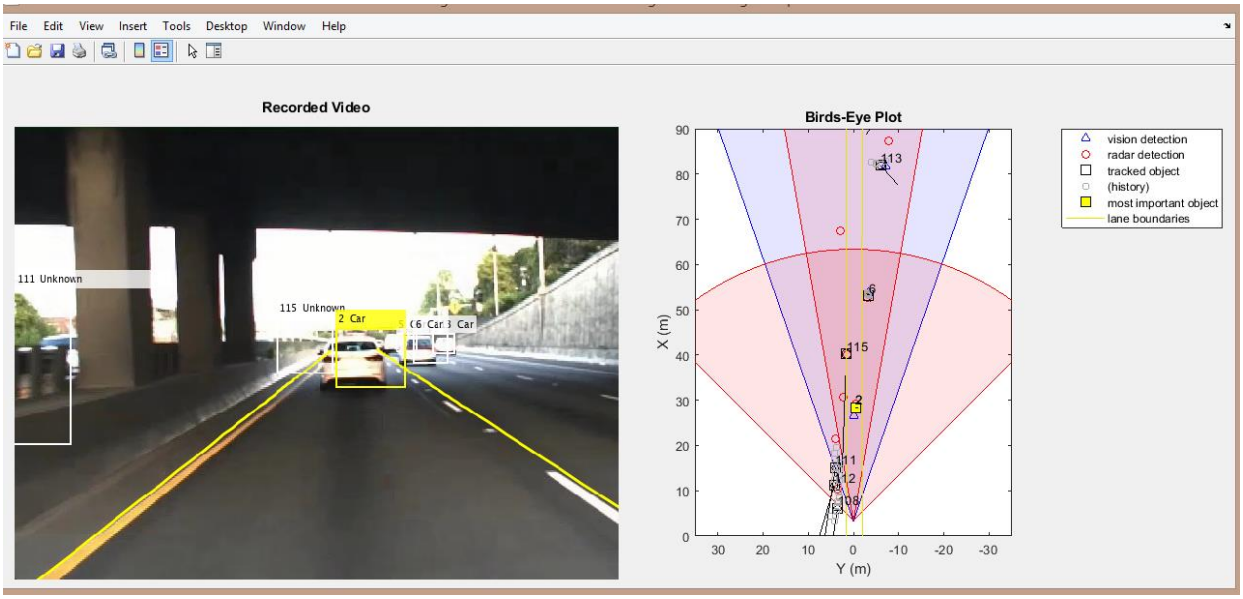


Figure 6.10. SNAPSHOT OF THE SIMULATION - 2

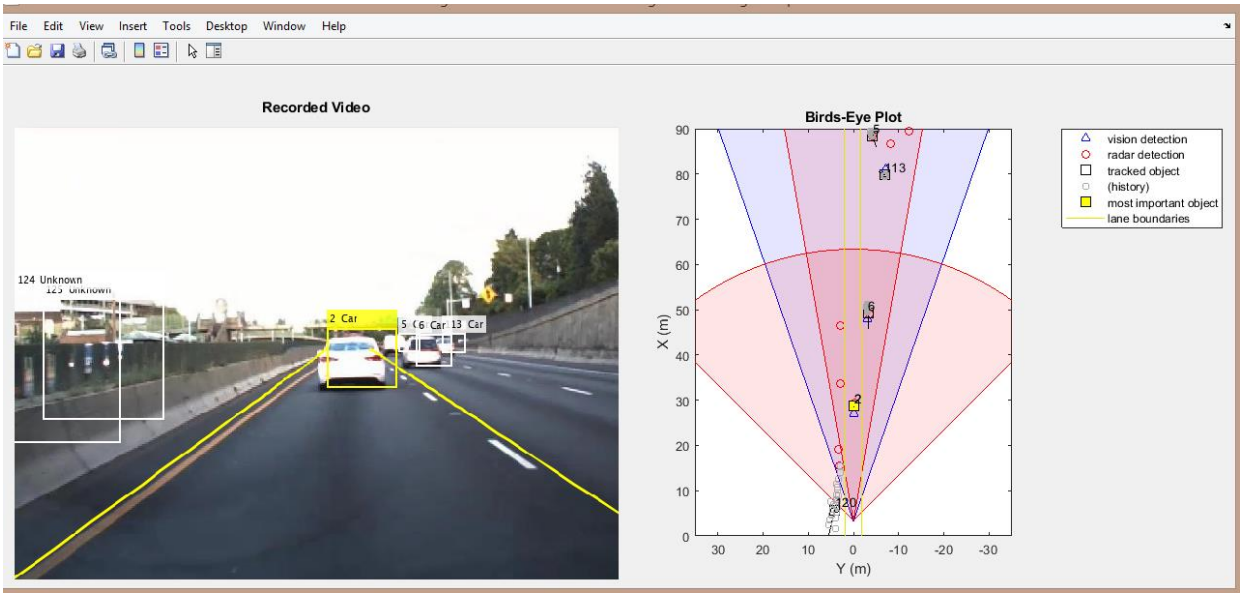


Figure 6.11. SNAPSHOT OF THE SIMULATION - 3

Figure 6.12. SNAPSHOT OF THE SIMULATION - 4

Figure 6.13. SNAPSHOT OF THE SIMULATION - 5

Figure 6.14. SNAPSHOT OF THE SIMULATION - 6

Figure 6.15. SNAPSHOT OF THE SIMULATION - 7

CHAPTER 7
CONCLUSION AND FUTURE SCOPE

7.1. CONCLUSION

Our project “Simulation Of Sensor Fusion Module Using Radar And Vision Sensor For Autonomous Vehicle In MATLAB” is a part of the Autonomous car project which was assigned to our team. Autonomous car (SREE-AV) is an in-house college project which has been further divided into multiple projects and assigned to various teams.

In this project sensor fusion module is designed and tested using simulation in MATLAB. Initially the RADAR and Vision Sensor are modeled and used in a scenario to obtain object detection data. Then the data from RADAR and Vision Sensor are fused using the designed module and the functionality of the sensor fusion module is tested. Automated Driving Toolbox and Sensor Fusion and Testing Toolbox available in MATLAB are used in designing the Sensor Fusion Module.

7.2. FUTURE SCOPE

The future scope of this project would be to convert the designed sensor fusion algorithms from MATLAB code into either C\C++ code or python code so that the designed sensor fusion module can be implemented on a hardware platform and tested. The same process could be applied to the other sensor fusion modules required for Autonomous vehicles so that all the sensor fusion modules could be combined to have a complete understanding of the Autonomous Vehicle surroundings thereby right decisions could be taken by the Autonomous Vehicle.

REFERENCES

[1] Rangesh and M. M. Trivedi, "No Blind Spots: Full-Surround Multi-Object Tracking for Autonomous Vehicles Using Cameras and LiDARs," in IEEE Transactions on Intelligent Vehicles, vol. 4, no. 4, pp. 588-599, Dec. 2019.

[2] X. Du, M. H. Ang and D. Rus, "Car detection for autonomous vehicle: LIDAR and vision fusion approach through deep learning framework," 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, 2017, pp. 749-754.

[3] F. Zhang, D. Clarke and A. Knoll, "Vehicle detection based on LiDAR and camera fusion," 17th International IEEE Conference on Intelligent Transportation Systems (ITSC), Qingdao, 2014, pp. 1620-1625.

[4] Q. Li, L. Chen, M. Li, S. Shaw and A. Nüchter, "A Sensor-Fusion Drivable-Region and Lane-Detection System for Autonomous Vehicle Navigation in Challenging Road Scenarios," in IEEE Transactions on Vehicular Technology, vol. 63, no. 2, pp. 540-555, Feb. 2014.

[5]S. Glaser, B. Vanholme, S. Mammar, D. Gruyer and L. Nouvelière, "Maneuver-Based Trajectory Planning for Highly Autonomous Vehicles on Real Road With Traffic and Driver Interaction," in IEEE Transactions on Intelligent Transportation Systems, vol. 11, no. 3, pp. 589-606, Sept. 2010.

[6]A. H. G. Al-Dhaher and D. Mackesy, "Multi-sensor data fusion architecture," The 3rd IEEE International Workshop on Haptic, Audio and Visual Environments and Their Applications, Ottawa, Ontario, Canada, 2004, pp. 159-163.

[7] Miller, Matt L., Harold S. Stone, and Ingemar J. Cox, “Optimizing Murty's Ranked Assignment Method,” *IEEE Transactions on Aerospace and Electronic Systems*, 33(3), 1997.

[8] Munkres, James, “Algorithms for Assignment and Transportation Problems,” *Journal of the Society for Industrial and Applied Mathematics*, Volume 5, Number 1, March, 1957

APPENDIX

```
%% Overview
% Forward collision warning (FCW) is an important feature in driver
% assistance and automated driving systems, where the goal is to provide
% correct, timely, and reliable warnings to the driver before an impending
% collision with the vehicle in front. To achieve the goal, vehicles are
% equipped with forward-facing vision and radar sensors. Sensor fusion is
% required to increase the probability of accurate warnings and minimize
% the probability of false warnings.

% This is a script, with the main body shown here and helper
% routines in the form of local functions in the sections that follow. For
% more details about local functions, see <docid:matlab_examples#bvfim1g
% Add Functions to Scripts>.
%

% Set up the display
[videoReader, videoDisplayHandle, bepPlotters, sensor] =
helperCreateFCWDemoDisplay('05_highway_lanechange_25s.mp4',
'SensorConfigurationData.mat');

% Read the recorded detections file
[visionObjects, radarObjects, inertialMeasurementUnit, laneReports, ...
    timeStep, numSteps] =
readSensorRecordingsFile('05_highway_lanechange_25s_sensor.mat');

% An initial ego lane is calculated. If the recorded lane information is
% invalid, define the lane boundaries as straight lines half a lane
% distance on each side of the car
laneWidth = 3.6; % meters
egoLane = struct('left', [0 0 laneWidth/2], 'right', [0 0 -laneWidth/2]);

% Prepare some time variables
time = 0;           % Time since the beginning of the recording
currentStep = 0;    % Current timestep
snapTime = 9.3;     % The time to capture a snapshot of the display

% Initialize the tracker
[tracker, positionSelector, velocitySelector] = setupTracker();

while currentStep < numSteps && ishghandle(videoDisplayHandle)
    % Update scenario counters
    currentStep = currentStep + 1;
    time = time + timeStep;

    % Process the sensor detections as objectDetection inputs to the tracker
    [detections, laneBoundaries, egoLane] = processDetections(...
        visionObjects(currentStep), radarObjects(currentStep), ...
        inertialMeasurementUnit(currentStep), laneReports(currentStep), ...
        egoLane, time);
```

```
    % Using the list of objectDetections, return the tracks, updated to time
    confirmedTracks = updateTracks(tracker, detections, time);

    % Find the most important object and calculate the forward collision
    % warning
    mostImportantObject = findMostImportantObject(confirmedTracks,
egoLane, positionSelector, velocitySelector);

    % Update video and birds-eye plot displays
    frame = readFrame(videoReader);     % Read video frame
    helperUpdateFCWDemoDisplay(frame, videoDisplayHandle, bepPlotters,
...
        laneBoundaries, sensor, confirmedTracks, mostImportantObject,
positionSelector, ...
        velocitySelector, visionObjects(currentStep),
radarObjects(currentStep));

    % Capture a snapshot
    if time >= snapTime && time < snapTime + timeStep
        snapnow;
    end
end

%% Create the Multi-Object Tracker
% The |<docid:driving_ref#bvigrmf-1 multiObjectTracker>| tracks the
objects around the ego vehicle based
% on the object lists reported by the vision and radar sensors. By fusing
% information from both sensors, the probability of a false collision
% warning is reduced.
%
% The |setupTracker| function returns the |multiObjectTracker|.
% When creating a |multiObjectTracker|, consider the following:
%
% # |FilterInitializationFcn|: The likely motion and measurement models.
%   In this case, the objects are expected to have a constant acceleration
%   motion. Although you can configure a linear Kalman filter for this
%   model, |initConstantAccelerationFilter| configures an extended Kalman
%   filter. See the 'Define a Kalman filter' section.
% # |AssignmentThreshold|: How far detections can fall from tracks.
%   The default value for this parameter is 30. If there are detections
%   that are not assigned to tracks, but should be, increase this value. If
%   there are detections that get assigned to tracks that are too far,
%   decrease this value. This example uses 35.
% # |NumCoastingUpdates|: How many times a track is coasted before
deletion.
%   Coasting is a term used for updating the track without an assigned
%   detection (predicting).
%   The default value for this parameter is 5. In this case, the tracker is
%   called 20 times a second and there are two sensors, so there is no need
%   to modify the default.
```

```
% # |ConfirmationParameters|: The parameters for confirming a track.
%   A new track is initialized with every unassigned detection. Some of
%   these detections might be false, so all the tracks are initialized as
%   |'Tentative'|. To confirm a track, it has to be detected at least _M_
%   times in _N_ tracker updates. The choice of _M_ and _N_ depends on
the
%   visibility of the objects. This example uses the default of 2
%   detections out of 3 updates.
%
% The outputs of |setupTracker| are:
%
% * |tracker| - The |multiObjectTracker| that is configured for this case.
% * |positionSelector| - A matrix that specifies which elements of the
%   State vector are the position: |position = positionSelector * State|
% * |velocitySelector| - A matrix that specifies which elements of the
%   State vector are the velocity: |velocity = velocitySelector * State|
    function [tracker, positionSelector, velocitySelector] = setupTracker()
        tracker = multiObjectTracker(...
            'FilterInitializationFcn', @initConstantAccelerationFilter, ...
            'AssignmentThreshold', 35, 'ConfirmationParameters', [2 3], ...
            'NumCoastingUpdates', 5);

        % The State vector is:
        %   In constant velocity:     State = [x;vx;y;vy]
        %   In constant acceleration: State = [x;vx;ax;y;vy;ay]

        % Define which part of the State is the position. For example:
        %   In constant velocity:     [x;y] = [1 0 0 0; 0 0 1 0] * State
        %   In constant acceleration: [x;y] = [1 0 0 0 0 0; 0 0 0 1 0 0] * State
        positionSelector = [1 0 0 0 0 0; 0 0 0 1 0 0];

        % Define which part of the State is the velocity. For example:
        %   In constant velocity:     [x;y] = [0 1 0 0; 0 0 0 1] * State
        %   In constant acceleration: [x;y] = [0 1 0 0 0 0; 0 0 0 0 1 0] * State
        velocitySelector = [0 1 0 0 0 0; 0 0 0 0 1 0];
    end

%% Define a Kalman Filter
% The |multiObjectTracker| defined in the previous section uses the filter
% initialization function defined in this section to create a Kalman filter
% (linear, extended, or unscented). This filter is then used for tracking
% each object around the ego vehicle.
function filter = initConstantAccelerationFilter(detection)
% This function shows how to configure a constant acceleration filter. The
% input is an objectDetection and the output is a tracking filter.
% For clarity, this function shows how to configure a trackingKF,
% trackingEKF, or trackingUKF for constant acceleration.
%
% Steps for creating a filter:
%   1. Define the motion model and state
%   2. Define the process noise
```

```
%   3. Define the measurement model
%   4. Initialize the state vector based on the measurement
%   5. Initialize the state covariance based on the measurement noise
%   6. Create the correct filter

    % Step 1: Define the motion model and state
    % This example uses a constant acceleration model, so:
    STF = @constacc;     % State-transition function, for EKF and UKF
    STFJ = @constaccjac; % State-transition function Jacobian, only for EKF
    % The motion model implies that the state is [x;vx;ax;y;vy;ay]
    % You can also use constvel and constveljac to set up a constant
    % velocity model, constturn and constturnjac to set up a constant turn
    % rate model, or write your own models.

    % Step 2: Define the process noise
    dt = 0.05; % Known timestep size
    sigma = 1; % Magnitude of the unknown acceleration change rate
    % The process noise along one dimension
    Q1d = [dt^4/4, dt^3/2, dt^2/2; dt^3/2, dt^2, dt; dt^2/2, dt, 1] *
sigma^2;
    Q = blkdiag(Q1d, Q1d); % 2-D process noise

    % Step 3: Define the measurement model
    MF = @fcwmeas;       % Measurement function, for EKF and UKF
    MJF = @fcwmeasjac;   % Measurement Jacobian function, only for EKF

    % Step 4: Initialize a state vector based on the measurement
    % The sensors measure [x;vx;y;vy] and the constant acceleration model's
    % state is [x;vx;ax;y;vy;ay], so the third and sixth elements of the
    % state vector are initialized to zero.
    state = [detection.Measurement(1); detection.Measurement(2); 0;
detection.Measurement(3); detection.Measurement(4); 0];

    % Step 5: Initialize the state covariance based on the measurement
    % noise. The parts of the state that are not directly measured are
    % assigned a large measurement noise value to account for that.
    L = 100; % A large number relative to the measurement noise
    stateCov = blkdiag(detection.MeasurementNoise(1:2,1:2), L,
detection.MeasurementNoise(3:4,3:4), L);

    % Step 6: Create the correct filter.
    % Use 'KF' for trackingKF, 'EKF' for trackingEKF, or 'UKF' for
trackingUKF
    FilterType = 'EKF';

    % Creating the filter:
    switch FilterType
        case 'EKF'
            filter = trackingEKF(STF, MF, state,...
                'StateCovariance', stateCov, ...
                'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
```

```
            'StateTransitionJacobianFcn', STFJ, ...
            'MeasurementJacobianFcn', MJF, ...
            'ProcessNoise', Q ...
            );
        case 'UKF'
            filter = trackingUKF(STF, MF, state, ...
            'StateCovariance', stateCov, ...
            'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
            'Alpha', 1e-1, ...
            'ProcessNoise', Q ...
            );
        case 'KF' % The ConstantAcceleration model is linear and KF can be
used
            % Define the measurement model: measurement = H * state
            % In this case:
            %   measurement = [x;vx;y;vy] = H * [x;vx;ax;y;vy;ay]
            % So, H = [1 0 0 0 0 0; 0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 1 0]
            %
            % Note that ProcessNoise is automatically calculated by the
            % ConstantAcceleration motion model
            H = [1 0 0 0 0 0; 0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 1 0];
            filter = trackingKF('MotionModel', '2D Constant Acceleration', ...
            'MeasurementModel', H, 'State', state, ...
            'MeasurementNoise', detection.MeasurementNoise(1:4,1:4), ...
            'StateCovariance', stateCov);
    end
end

%%% Process and Format the Detections
% The recorded information must be processed and formatted before it can
be
% used by the tracker. This has the following steps:
%
% # Filtering out unnecessary radar clutter detections. The
%   radar reports many objects that correspond to fixed objects, which
%   include: guard-rails, the road median, traffic signs, etc. If these
%   detections are used in the tracking, they create false tracks of fixed
%   objects at the edges of the road and therefore must be removed before
%   calling the tracker. Radar objects are considered nonclutter if they
%   are either stationary in front of the car or moving in its vicinity.
% # Formatting the detections as input to the tracker, i.e., an array of
%   |<docid:driving_ref#bvie5il-1 objectDetection>| elements. See the
%   |processVideo| and |processRadar| supporting functions at the end of
%   this example.
    function [detections,laneBoundaries, egoLane] = processDetections...
            (visionFrame, radarFrame, IMUFrame, laneFrame, egoLane, time)
        % Inputs:
        %   visionFrame - objects reported by the vision sensor for this time
frame
        %   radarFrame  - objects reported by the radar sensor for this time
frame
```

```
        %   IMUFrame    - inertial measurement unit data for this time frame
        %   laneFrame   - lane reports for this time frame
        %   egoLane     - the estimated ego lane
        %   time        - the time corresponding to the time frame

        % Remove clutter radar objects
        [laneBoundaries, egoLane] = processLanes(laneFrame, egoLane);
        realRadarObjects = findNonClutterRadarObjects(radarFrame.object,...
            radarFrame.numObjects, IMUFrame.velocity, laneBoundaries);

        % Return an empty list if no objects are reported

        % Counting the total number of objects
        detections = {};
        if (visionFrame.numObjects + numel(realRadarObjects)) == 0
            return;
        end

        % Process the remaining radar objects
        detections = processRadar(detections, realRadarObjects, time);

        % Process video objects
        detections = processVideo(detections, visionFrame, time);
    end
%% Update the Tracker
% To update the tracker, call the |updateTracks| method with the
following
% inputs:
%
% # |tracker| - The |multiObjectTracker| that was configured earlier. See
%   the 'Create the Multi-Object Tracker' section.
% # |detections| - A list of |objectDetection| objects that was created by
%   |processDetections|
% # |time| - The current scenario time.
%
% The output from the tracker is a |struct| array of tracks.

%%% Find the Most Important Object and Issue a Forward Collision
Warning
% The most important object (MIO) is defined as the track that is in the
% ego lane and is closest in front of the car, i.e., with the smallest
% positive _x_ value. To lower the probability of false alarms, only
% confirmed tracks are considered.
%
% Once the MIO is found, the relative speed between the car and MIO is
% calculated. The relative distance and relative speed determine the
% forward collision warning. There are 3 cases of FCW:
%
% # Safe (green): There is no car in the ego lane (no MIO), the MIO is
%   moving away from the car, or the distance to the MIO remains
constant.
```

```
% # Caution (yellow): The MIO is moving closer to the car, but is still at
%   a distance above the FCW distance. FCW distance is calculated using
the
%   Euro NCAP AEB Test Protocol. Note that this distance varies with the
%   relative speed between the MIO and the car, and is greater when the
%   closing speed is higher.
% # Warn (red): The MIO is moving closer to the car, and its distance is
%   less than the FCW distance, $d_{FCW}$.
%
% Euro NCAP AEB Test Protocol defines the following distance calculation:
%%
% $d_{FCW} = 1.2 * v_{rel} + \frac{v_{rel}^2}{2a_{max}}$
%
% where:
%
% $d_{FCW}$ is the forward collision warning distance.
%
% $v_{rel}$ is the relative velocity between the two vehicles.
%
% $a_{max}$ is the maximum deceleration, defined to be 40% of the gravity
% acceleration.
   function mostImportantObject =
findMostImportantObject(confirmedTracks,egoLane,positionSelector,velocit
ySelector)

        % Initialize outputs and parameters
        MIO = [];               % By default, there is no MIO
        trackID = [];           % By default, there is no trackID associated with
an MIO
        FCW = 3;                % By default, if there is no MIO, then FCW is 'safe'
        threatColor = 'green';  % By default, the threat color is green
        maxX = 1000;  % Far enough forward so that no track is expected to
exceed this distance
        gAccel = 9.8; % Constant gravity acceleration, in m/s^2
        maxDeceleration = 0.4 * gAccel; % Euro NCAP AEB definition
        delayTime = 1.2; % Delay time for a driver before starting to brake, in
seconds

        positions = getTrackPositions(confirmedTracks, positionSelector);
        velocities = getTrackVelocities(confirmedTracks, velocitySelector);

        for i = 1:numel(confirmedTracks)
            x = positions(i,1);
            y = positions(i,2);

            relSpeed = velocities(i,1); % The relative speed between the cars,
along the lane

            if x < maxX && x > 0 % No point checking otherwise
                yleftLane  = polyval(egoLane.left,  x);
                yrightLane = polyval(egoLane.right, x);
```

```
            if (yrightLane <= y) && (y <= yleftLane)
                maxX = x;
                trackID = i;
                MIO = confirmedTracks(i).TrackID;
                if relSpeed < 0 % Relative speed indicates object is getting
closer
                    % Calculate expected braking distance according to
                    % Euro NCAP AEB Test Protocol
                    d = abs(relSpeed) * delayTime + relSpeed^2 / 2 /
maxDeceleration;
                    if x <= d % 'warn'
                        FCW = 1;
                        threatColor = 'red';
                    else % 'caution'
                        FCW = 2;
                        threatColor = 'yellow';
                    end
                end
            end
        end
        mostImportantObject = struct('ObjectID', MIO, 'TrackIndex', trackID,
'Warning', FCW, 'ThreatColor', threatColor);
    end

%% Summary
% This example showed how to create a forward collision warning system
for
% a vehicle equipped with vision, radar, and IMU sensors. It used
% |objectDetection| objects to pass the sensor reports to the
% |multiObjectTracker| object that fused them and tracked objects in front
% of the ego vehicle.
%
% Try using different parameters for the tracker to see how they affect the
% tracking quality. Try modifying the tracking filter to use |trackingKF|
% or |trackingUKF|, or to define a different motion model, e.g., constant
% velocity or constant turn. Finally, you can try to define your own motion
% model.

%% Supporting Functions
%%%
% *readSensorRecordingsFile*
% Reads recorded sensor data from a file
function [visionObjects, radarObjects, inertialMeasurementUnit,
laneReports, ...
    timeStep, numSteps] =
readSensorRecordingsFile(sensorRecordingFileName)
% Read Sensor Recordings
% The |ReadDetectionsFile| function reads the recorded sensor data file.
% The recorded data is a single structure that is divided into the
% following substructures:
```

```
%
% # |inertialMeasurementUnit|, a struct array with fields: timeStamp,
%   velocity, and yawRate. Each element of the array corresponds to a
%   different timestep.
% # |laneReports|, a struct array with fields: left and right. Each element
%   of the array corresponds to a different timestep.
%   Both left and right are structures with fields: isValid, confidence,
%   boundaryType, offset, headingAngle, and curvature.
% # |radarObjects|, a struct array with fields: timeStamp (see below),
%   numObjects (integer) and object (struct). Each element of the array
%   corresponds to a different timestep.
%   |object| is a struct array, where each element is a separate object,
%   with the fields: id, status, position(x;y;z), velocity(vx,vy,vz),
%   amplitude, and rangeMode.
%   Note: z is always constant and vz=0.
% # |visionObjects|, a struct array with fields: timeStamp (see below),
%   numObjects (integer) and object (struct). Each element of the array
%   corresponds to a different timestep.
%   |object| is a struct array, where each element is a separate object,
%   with the fields: id, classification, position (x;y;z),
%   velocity(vx;vy;vz), size(dx;dy;dz). Note: z=vy=vz=dx=dz=0
%
% The timeStamp for recorded vision and radar objects is a uint64 variable
% holding microseconds since the Unix epoch. Timestamps are recorded
about
% 50 milliseconds apart. There is a complete synchronization between the
% recordings of vision and radar detections, therefore the timestamps are
% not used in further calculations.

A = load(sensorRecordingFileName);
visionObjects = A.vision;
radarObjects = A.radar;
laneReports = A.lane;
inertialMeasurementUnit = A.inertialMeasurementUnit;

timeStep = 0.05;                % Data is provided every 50 milliseconds
numSteps = numel(visionObjects); % Number of recorded timesteps
end
%%%
% *processLanes*
% Converts sensor-reported lanes to |parabolicLaneBoundary| lanes and
% maintains a persistent ego lane estimate
function [laneBoundaries, egoLane] = processLanes(laneReports, egoLane)
% Lane boundaries are updated based on the laneReports from the
recordings.
% Since some laneReports contain invalid (isValid = false) reports or
% impossible parameter values (-1e9), these lane reports are ignored and
% the previous lane boundary is used.
leftLane    = laneReports.left;
rightLane   = laneReports.right;
```

```
% Check the validity of the reported left lane
cond = (leftLane.isValid && leftLane.confidence) && ...
    ~(leftLane.headingAngle == -1e9 || leftLane.curvature == -1e9);
if cond
    egoLane.left = cast([leftLane.curvature, leftLane.headingAngle,
leftLane.offset], 'double');
end

% Update the left lane boundary parameters or use the previous ones
leftParams  = egoLane.left;
leftBoundaries = parabolicLaneBoundary(leftParams);
leftBoundaries.Strength = 1;

% Check the validity of the reported right lane
cond = (rightLane.isValid && rightLane.confidence) && ...
    ~(rightLane.headingAngle == -1e9 || rightLane.curvature == -1e9);
if cond
    egoLane.right  = cast([rightLane.curvature, rightLane.headingAngle,
rightLane.offset], 'double');
end

% Update the right lane boundary parameters or use the previous ones
rightParams = egoLane.right;
rightBoundaries = parabolicLaneBoundary(rightParams);
rightBoundaries.Strength = 1;

laneBoundaries = [leftBoundaries, rightBoundaries];
end
%%%
% *findNonClutterRadarObjects*
% Removes radar objects that are considered part of the clutter
function realRadarObjects = findNonClutterRadarObjects(radarObject,
numRadarObjects, egoSpeed, laneBoundaries)
% The radar objects include many objects that belong to the clutter.
% Clutter is defined as a stationary object that is not in front of the
% car. The following types of objects pass as nonclutter:
%
% # Any object in front of the car
% # Any moving object in the area of interest around the car, including
%   objects that move at a lateral speed around the car

    % Allocate memory
    normVs = zeros(numRadarObjects, 1);
    inLane = zeros(numRadarObjects, 1);
    inZone = zeros(numRadarObjects, 1);

    % Parameters
    LaneWidth = 3.6;          % What is considered in front of the car
    ZoneWidth = 1.7*LaneWidth;  % A wider area of interest
    minV = 1;                 % Any object that moves slower than minV is
considered stationary
```

```
    for j = 1:numRadarObjects
        [vx, vy] =
calculateGroundSpeed(radarObject(j).velocity(1),radarObject(j).velocity(2),eg
oSpeed);
        normVs(j) = norm([vx,vy]);
        laneBoundariesAtObject = computeBoundaryModel(laneBoundaries,
radarObject(j).position(1));
        laneCenter = mean(laneBoundariesAtObject);
        inLane(j) = (abs(radarObject(j).position(2) - laneCenter) <=
LaneWidth/2);
        inZone(j) = (abs(radarObject(j).position(2) - laneCenter) <=
max(abs(vy)*2, ZoneWidth));
    end
    realRadarObjectsIdx = union(...
        intersect(find(normVs > minV), find(inZone == 1)), ...
        find(inLane == 1));

    realRadarObjects = radarObject(realRadarObjectsIdx);
end
%%%
% *calculateGroundSpeed*
% Calculates the true ground speed of a radar-reported object from the
% relative speed and the ego vehicle speed
function [Vx,Vy] = calculateGroundSpeed(Vxi,Vyi,egoSpeed)
% Inputs
%   (Vxi,Vyi) : relative object speed
%   egoSpeed  : ego vehicle speed
% Outputs
%   [Vx,Vy]   : ground object speed

    Vx = Vxi + egoSpeed;    % Calculate longitudinal ground speed
    theta = atan2(Vyi,Vxi); % Calculate heading angle
    Vy = Vx * tan(theta);   % Calculate lateral ground speed

end
%%%
% *processVideo*
% Converts reported vision objects to a list of |objectDetection| objects
function postProcessedDetections = processVideo(postProcessedDetections,
visionFrame, t)
% Process the video objects into objectDetection objects
numRadarObjects = numel(postProcessedDetections);
numVisionObjects = visionFrame.numObjects;
if numVisionObjects
    classToUse = class(visionFrame.object(1).position);
    visionMeasCov = cast(diag([2,2,2,100]), classToUse);
    % Process Vision Objects:
    for i=1:numVisionObjects
        object = visionFrame.object(i);
        postProcessedDetections{numRadarObjects+i} = objectDetection(t,...
            [object.position(1); object.velocity(1); object.position(2); 0], ...
```

```
            'SensorIndex', 1, 'MeasurementNoise', visionMeasCov, ...
            'MeasurementParameters', {1}, ...
            'ObjectClassID', object.classification, ...
            'ObjectAttributes', {object.id, object.size});
    end
end
end
%%%
% *processRadar*
% Converts reported radar objects to a list of |objectDetection| objects
function postProcessedDetections = processRadar(postProcessedDetections,
realRadarObjects, t)
% Process the radar objects into objectDetection objects
numRadarObjects = numel(realRadarObjects);
if numRadarObjects
    classToUse = class(realRadarObjects(1).position);
    radarMeasCov = cast(diag([2,2,2,100]), classToUse);
    % Process Radar Objects:
    for i=1:numRadarObjects
        object = realRadarObjects(i);
        postProcessedDetections{i} = objectDetection(t, ...
            [object.position(1); object.velocity(1); object.position(2);
object.velocity(2)], ...
            'SensorIndex', 2, 'MeasurementNoise', radarMeasCov, ...
            'MeasurementParameters', {2}, ...
            'ObjectAttributes', {object.id, object.status, object.amplitude,
object.rangeMode});
    end
end
end
%%%
% *fcwmeas*
% The measurement function used in this forward collision warning
example
function measurement = fcwmeas(state, sensorID)
% The example measurements depend on the sensor type, which is
reported by
% the MeasurementParameters property of the objectDetection. The
following
% two sensorID values are used:
%   sensorID=1: video objects, the measurement is [x;vx;y].
%   sensorID=2: radar objects, the measurement is [x;vx;y;vy].
% The state is:
%   Constant velocity       state = [x;vx;y;vy]
%   Constant turn           state = [x;vx;y;vy;omega]
%   Constant acceleration   state = [x;vx;ax;y;vy;ay]

    if numel(state) < 6 % Constant turn or constant velocity
        switch sensorID
            case 1 % video
                measurement = [state(1:3); 0];
```

```
            case 2 % radar
                measurement = state(1:4);
        end
    else % Constant acceleration
        switch sensorID
            case 1 % video
                measurement = [state(1:2); state(4); 0];
            case 2 % radar
                measurement = [state(1:2); state(4:5)];
        end
    end
end
%%%
% *fcwmeasjac*
% The Jacobian of the measurement function used in this forward collision
% warning example
function jacobian = fcwmeasjac(state, sensorID)
% The example measurements depend on the sensor type, which is
reported by
% the MeasurementParameters property of the objectDetection. We choose
% sensorID=1 for video objects and sensorID=2 for radar objects.  The
% following two sensorID values are used:
%   sensorID=1: video objects, the measurement is [x;vx;y].
%   sensorID=2: radar objects, the measurement is [x;vx;y;vy].
% The state is:
%   Constant velocity       state = [x;vx;y;vy]
%   Constant turn           state = [x;vx;y;vy;omega]
%   Constant acceleration   state = [x;vx;ax;y;vy;ay]

    numStates = numel(state);
    jacobian = zeros(4, numStates);

    if numel(state) < 6 % Constant turn or constant velocity
        switch sensorID
            case 1 % video
                jacobian(1,1) = 1;
                jacobian(2,2) = 1;
                jacobian(3,3) = 1;
            case 2 % radar
                jacobian(1,1) = 1;
                jacobian(2,2) = 1;
                jacobian(3,3) = 1;
                jacobian(4,4) = 1;
        end
    else % Constant acceleration
        switch sensorID
            case 1 % video
                jacobian(1,1) = 1;
                jacobian(2,2) = 1;
                jacobian(3,4) = 1;
            case 2 % radar
```

```
            jacobian(1,1) = 1;
            jacobian(2,2) = 1;
            jacobian(3,4) = 1;
            jacobian(4,5) = 1;
        end
    end
end
```

Batch9

ORIGINALITY REPORT

20%	18%	3%	10%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to University of Sheffield Student Paper	9%
2	www.mathworks.com Internet Source	7%
3	Submitted to German University of Technology in Oman Student Paper	1%
4	Submitted to Oxford Brookes University Student Paper	1%
5	Submitted to Middle East College of Information Technology Student Paper	1%
6	Stuart I. Granshaw. "Sensor fusion", The Photogrammetric Record, 2020 Publication	1%
7	encs.concordia.ca Internet Source	<1%
8	en.wikipedia.org Internet Source	<1%