KLE Technological University
Creating Value
Leveraging Knowledge

Department of Automation and Robotics
17EARW302

Minor Project Report

Simulation of Sensor Fusion and Object Tracking for KLE Tech Autonomous Electric Vehicle

KLE TECHNOLOGICAL UNIVERSITY

DEPARTMENT OF AUTOMATION & ROBOTICS

Certificate

This is to certify that the project entitled " **Simulation of Sensor Fusion and Object Tracking for KLE Tech Autonomous Vehicle "** is carried out by the below-mentioned student as part of the course **MINOR PROJECT (17EARW302)**, studying at KLE Technological University, Hubballi, during 6th Semester of the B.E program for the academic year 2019-20.

The project report fulfils the requirements prescribed.

Chinmaya Sabnis	01FE17BAR013
N Anjana	01FE17BAR029
Ranjitha A G	01FE17BAR048
Samruddhi S Parwapur	01FE17BAR051
Sushma R Hiremath	01FE17BAR057

Prof. A C Giriyapur,
Head of Dept.
Automation and Robotics

ABSTRACT

The minor project course offered the Self-Driving task for the KLE Tech AEV. Self-driving cars have a huge demand in the world and exist with the improvement in current technology. It is the most outstanding project and very challenging, also accounts for the major changes in the scenario which it must tackle.

The learning was done in numerous ways to understand each sensor and the purpose of the sensor and its typical usage to tackle the problem and to have a clear picture of an obstacle or a path to change and locate. The sensors such as lidar, radar, camera, and IMU helps in understanding the exact situation of the vehicle position, the forward path and obstacles. The fundamental for the vehicle to travel will be to completely analyze the situation and act accordingly. Self-driving cars have proved to be the most important task for a human to tackle. Throughout this project, the key points and figures explain how a team could able to analyze the sensors- modeling and simulation. The learning outcome was also be helpful to understand the ground truth labeler which is performed in MATLAB.

In this project, the presentation of work describes the various online courses completed by the team also analysis and visualization of sensors and ground truth labelers.

The presented work in this project report has been indulged in explanation of various techniques and analysis needed to understand the fundamentals of approaching the completion of the basics of a self-driving vehicle to implement in future use.

ACKNOWLEDGMENT

"Self-driving cars are the natural extension of active safety and something we should do" – Elon Musk

I would like to express my special thanks of gratitude to **Prof. A C Giriyapur, Head of Dept. Automation and Robotics** gave me the golden opportunity to do this wonderful project which also helped me in doing a lot of research and I was able to learn many new concepts.

I would also thank **Prof. Amit Talli** for his support and able directional guidance during the project.

I would also like to thank all the helpers for giving their precious time and relevant information and experience, I required, without which the Project would have been incomplete.

Finally, I would like to thank all teammates for their kind support and to all who have directly or indirectly helped me in learning the project and preparing this project report.

Contents

Introduction....1
Tracking scenario....2
Sensor detections....3
Lidar Detections....3
Radar Detections....5
Camera Detections....6
Extended Object Detections....7
Sensor Fusion....8
Fusing acceleration, magnetic fields, and orientations.....8
IMU and GPS fusion for inertial navigation....9
The fusion of Radar and Lidar....11
Camera And radar....13
IMU measurements....14
Orientation using an inertial sensor....15
Visual-inertial odometry using synthetic data....16
Occupancy grid and semantic segmentation....18
Automate ground-truth labeling of lane boundaries....19
Forward collision warning using sensor fusion....21
Estimating the distance of vehicles....22
Automated parking of a vehicle....27
Conclusions....31
Bibliography....32

List of Figures

Figure 1 Autonomous systems processing loop with a focus on perception 1
Figure 2 Sensor fusion and tracking workflow 1
Figure 3 Point and Frame rotation 2
Figure 4 Ground Reference frame 2
Figure 5 Bird's eye view 3
Figure 6 Point cloud 3
Figure 7 Lidar sequence 4
Figure 8 Lidar bounding box of vehicles 4
Figure 9 Position measurements 5
Figure 10 Pedestrian and Vehicle Detection 6
Figure 11 Monocular Camera View 6
Figure 12 Vehicles detected 7
Figure 13 Extended Object Detections 8
Figure 14 Sensor strength, capability and weakness table 8
Figure 15 Quaternion Distance, X, Y, Z Position error graphs 10
Figure 16 Position graph, Ground truth orientation, and Estimated orientation plots 10
Figure 17 Schematics of the workflow of track-level fusion 11
Figure 18 Front, rear and top view with the focus on the ego vehicle 12
Figure 19 Output array conventions 12
Figure 20 Top view, Chase camera view and Bird's – Eye plot 14
Figure 21 IMU Model 14
Figure 22 Inertia Sensor fusion for IMU 15
Figure 23 Accelerometer-Magnetometer Fusion: 15
Figure 24 Accelerometer-Gyroscope Fusion 16
Figure 25 Accelerometer-Gyroscope-Magnetometer Fusion 16
Figure 26 Visual-inertial odometry 17
Figure 27 Ground truth vehicle trajectory, the visual odometry estimate, and the fusion filter estimate. 18
Figure 28 Single frame segmented visual 18
Figure 29 Free space confidence scores 19
Figure 30 Occupancy grid(probability 19
Figure 31 Vehicle costmap 19
Figure 32 Automatically marked lane boundary points 20
Figure 33 Detected left lane boundary 20
Figure 34 Automatically marked lane boundary 20
Figure 35 scope output updated at the IMU sample rate 23
Figure 36 Monocular camera detecting vehicles and coordinates concerning camera as an origin 25
Figure 37 Vehicles detected by a camera and represented as bounding boxes 25
Figure 38 Vehicles detected by a camera with estimating distance 26
Figure 39 occupancy grid layers of a parking lot 27
Figure 40 combined costmap 28
Figure 41 Transition poses and directions from the planned path 29
Figure 42 Planning maneuver 30
Figure 43 Actual Path Traced bt Ego-Vehicle 30

TERMINOLOGIES

1. ACC: Adaptive Cruise Control, A cruise control system for vehicles which controls longitudinal speed. ACC can maintain the desired reference speed or adjust its speed accordingly to maintain safe driving distances to other vehicles.
2. Autonomy: The level of autonomy of a self-driving car refers to how much of the driving is done by a computer versus a human.
3. Azimuth: is defined as the angle from the x-axis to the orthogonal projection of the vector onto the XY-plane.
4. Detection: the perception that something has occurred or some state exists.
5. Ego: A term to express the notion of self, which is used to refer to the vehicle being controlled autonomously, as opposed to other vehicles or objects in the scene. It is most often used in the form ego-vehicle, meaning the self-vehicle.
6. Ego-vehicle: Within the creativity perspective, the ego is a vehicle that mind and the awareness of consciousness used as a vehicle for the experience of creation/Creation. Awareness is that which perceives, observes, and watches. It has no identity. It is seen to reside in mind but lies behind the mind and never changes.
7. Elevation: is defined as the angle from the projection onto the XY-plane to the vector.
8. Extended objects: Each extended object is an instance of an associated extended object type (ExtObjectType). (Some internal, base-Platform objects can be extended and those objects provide their own, predefined class objects that can be extended.
9. FMEA: Failure Mode and Effects Analysis, A bottom-up approach of failure analysis that examines individual causes and determines their effects on the higher-level system.
10. GNSS: Global Navigation Satellite System, A generic term for all satellite systems which provide position estimation. The Global Positioning System (GPS) made by the United States is a type of GNSS. Another example is the Russian made GLONASS
    (Globalnaya Navigazionnaya Sputnikovaya Sistema).
11. HAZOP: Hazard and Operability Study, A variation of FMEA (Failure Mode and Effects Analysis) which uses guide words to brainstorm over sets of possible failures that can arise.
12. IMU: Inertial Measurement Unit, A sensor device consisting of an accelerometer and a gyroscope. The IMU is used to measure vehicle acceleration and angular velocity, and its data can be fused with other sensors for state estimation.
13. kinematic states: Robot kinematics applies geometry to the study of the movement of multi-degree of freedom kinematic chains that form the structure of robotic systems.
14. LIDAR: Light Detection and Ranging, A type of sensor which detects range by transmitting light and measuring return time and shifts of the reflected signal.
15. Localization: Localization is a step implemented in the majority of robots and vehicles to locate with a really small margin of error
16. LQR: Linear Quadratic Regulation, A method of control utilizing full state feedback. The method seeks to optimize a quadratic cost function depends on the state and control input.
17. LTI: Linear Time-Invariant, A linear system whose dynamics do not change with time. For example, a car using the unicycle model is an LTI system. If the model includes the tires degrading over time (and changing the vehicle dynamics), then the system would no longer be LTI.
18. MPC: Model Predictive Control, A method of control whose control input optimizes a user-defined cost function over a finite time horizon. A common form of MPC is finite horizon LQR (linear quadratic regulation).
19. NHTSA: National Highway Traffic Safety Administration
20. Occupancy grid: Occupancy grids are used to represent a robot workspace as a discrete grid.

21. Orientation: is defined by angular displacement. Orientation can be described in terms of point or frame rotation. Orientation refers to the angular displacement of an object relative to a frame of reference.
22. Perception: This capability extends to the 360-degree field around the vehicle, enabling the car to detect and track all moving and static objects as it travels. Perception is the first stage in the computational pipeline for the safe functioning of a self-driving car.
23. Point clouds: A point cloud is a set of data points in space. Point clouds are generally produced by 3D scanners, which measure many points on the external surfaces of objects around them.
24. Point objects: A point object refers to a tiny object which is calculated or counted as a dot object to simplify the calculations. A real object can rotate as it moves. For example, a ball will be spinning while it is moving in a trajectory
25. Pose: Pose is defined as the combination of position and orientation.
26. Position: is defined as the translational distance from a parent frame origin to a child frame origin.
27. Quaternions: these are numbers of the form $a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$where $i^2 = j^2 = k^2 = ijk = -1$and a,b,c & d are real numbers. Such numbers are used to define the post of an object in 3-D space.
28. Sensor Fusion: Sensor fusion is combining of sensory data or data derived from disparate sources such that the resulting information has less uncertainty than would be possible when these sources were used individually.
29. Trajectory: defines how pose changes over time
30. Waypoints: A waypoint is an intermediate point or place on a route or line of travel, a stopping point or point at which course is changed.

Introduction

Autonomous systems range from vehicles that meet the various SAE levels of autonomy to systems including consumer quadcopters, package delivery drones, flying taxis, and robots for disaster relief and space exploration. Every autonomous system can be described at a high level with the block diagram shown in the figure below.

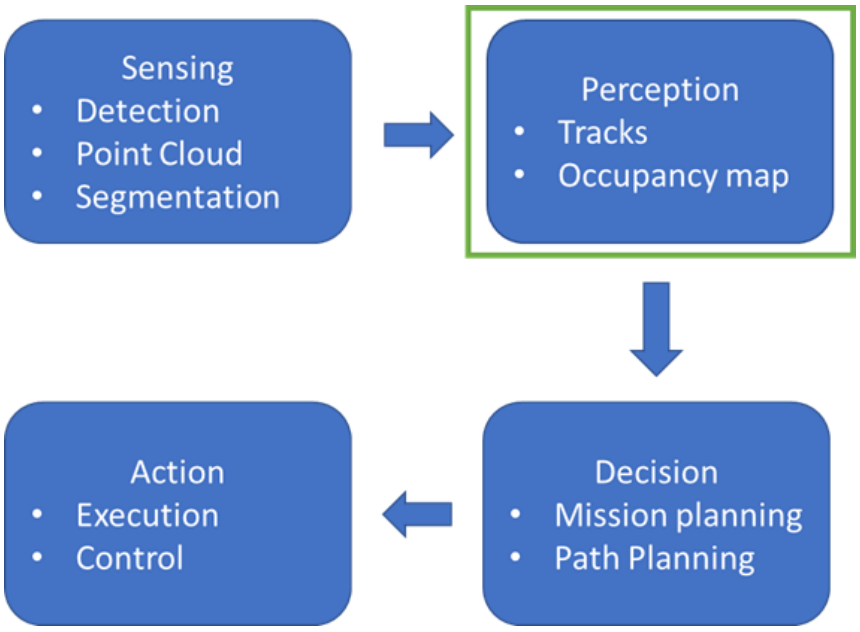

Figure 1 Autonomous systems processing loop with a focus on perception

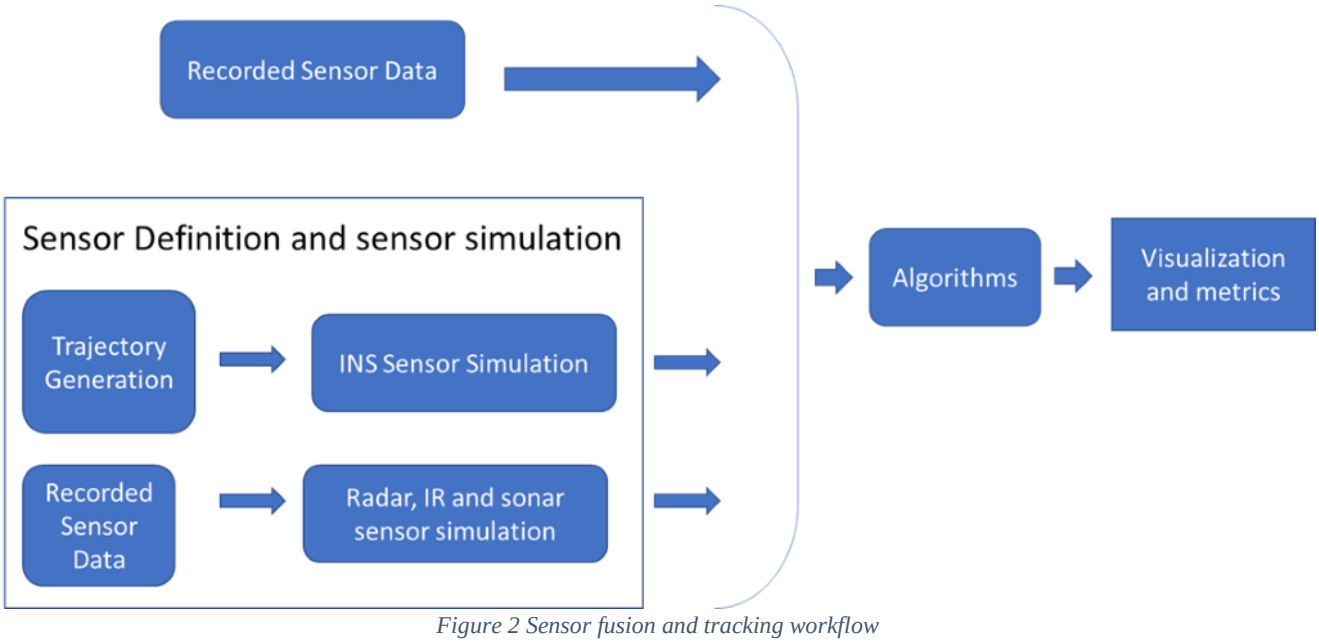

Figure 2 Sensor fusion and tracking workflow

The focus of this project is on the perception component of the autonomous systems processing loop. Sensor fusion and multi-object tracking are at the heart of perception, and that is where the examples below are focused. Autonomous systems rely on sensor suites that provide data about the surrounding environment to feed the perception system. These sensors include radars and cameras, which provide detections of objects in their field of view. They also include lidar sensors, which provide point clouds

of returns from obstacles in the environment, and in some cases, ultrasound and sonar sensors. Autonomous systems must also be able to estimate their position to maintain self-awareness.

For this, sensors such as inertial measurement units (IMUs) and global positioning system (GPS) receivers are used. The sensor system can also perform signal processing, including detection, segmentation, labeling, classification, and oftentimes basic tracking to reduce false alarms. Performing this function is a challenge for systems that must achieve the highest levels of autonomy because a lack of understanding of the occupied space around the autonomous system can have catastrophic results. Similarly, the presence of false tracks results in a confused state in the perception system, which impacts the planning and controls components. Models and simulations can extend to 2D and 3D environments. From a complete set of tracker libraries, sensor fusion and tracking algorithms are configured and integrated. Also, simulated sensor data, swap trackers and tracker components, and test sensor fusion architectures and compared them with the simulated truth.

Tracking scenario

A tracking scenario simulates a 3-D arena containing multiple platforms. Platforms represent anything that needs to be simulated, such as aircraft, ground vehicles, or ships. Some platforms carry sensors, such as radar, sonar, or infrared. The main benefit of using scenario generation and sensor simulation over sensor recording is the ability to create rare and potentially hazardous scenarios and test sensor fusion algorithms with these scenarios. These radar detections can be used to develop various tracking algorithms, such as trackerGNN and trackerJPDA. A platform refers generally to any object you want to track. The tracking of orientation, position, pose, and trajectory of a platform.

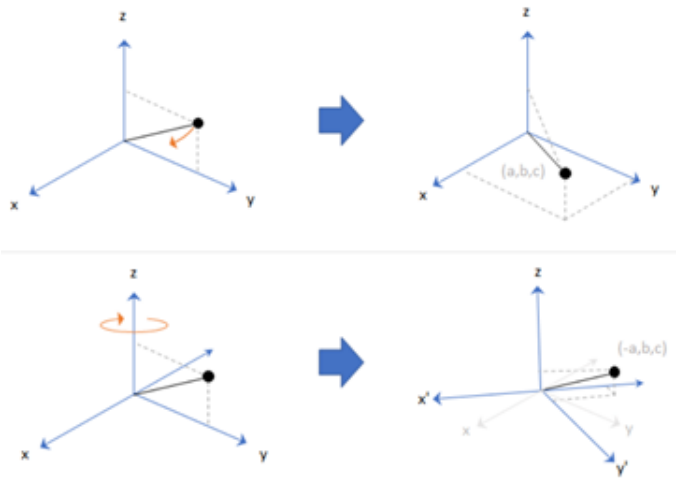

Figure 3 Point and Frame rotation

Orientation is defined by angular displacement. Orientation can be described in terms of point or frame rotation. In point rotation, the coordinate system is static and the point moves. In frame rotation, the point is static and the coordinate system moves.

There exist two right-handed variants: east, north, up (ENU) coordinates and north, east, down (NED) coordinates. They serve for representing state vectors that are commonly used in aviation and marine cybernetics.

The position is defined as the translational distance from a parent frame origin to a child frame origin.

The pose is to specify an object in 3-D space fully, you can combine position and orientation. The pose is defined as the combination of position and orientation.

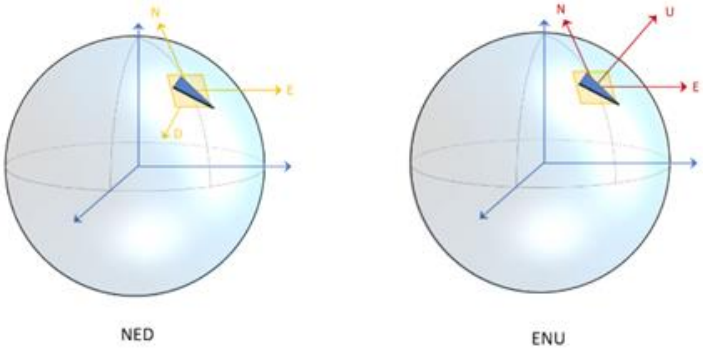

Figure 4 Ground Reference frame

Creating a tracking scenario, in which the motion of one or more moving targets is simulated. To set up a tracking scenario, first created a trackingScenario object. The step size is given by UpdateRate in Hz. Based on an update rate of 10 Hz, the step size of the scene is 0.1 sec.

To visualize the scenario created a theaterPlot and added a trajectoryPlotter and a platformPlotter to visualize the trajectory and the target, respectively. After this, the run option is used to the tracking scenario and visualized the target motion in a bird's eye view as shown in Figure 5.

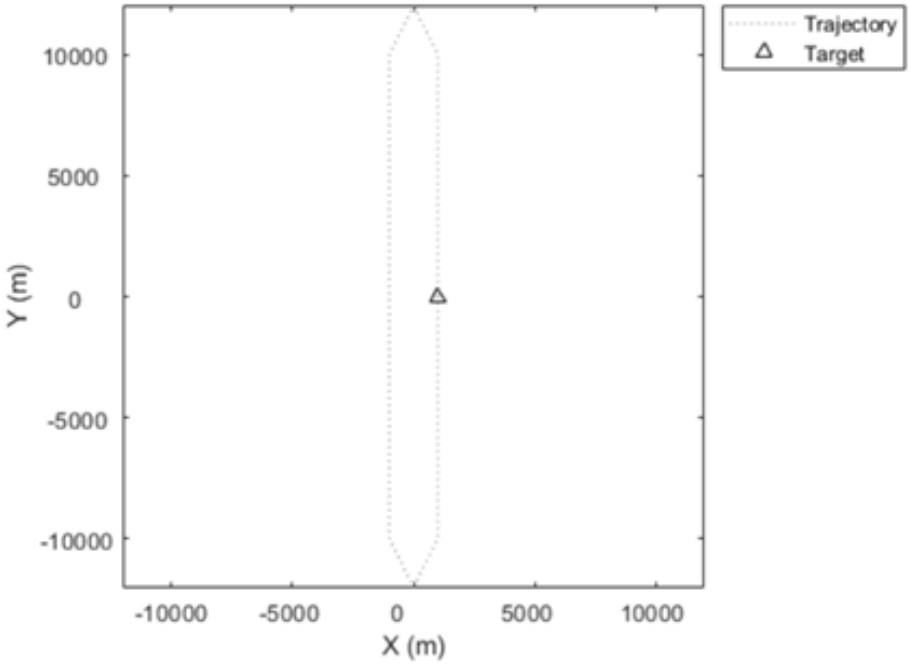

Figure 5 Bird's eye view

Sensor detections

Higher rates for data sampling are important for quick system activity learning. For example, in predictive maintenance, a sensor must sample at a fairly high rate to capture all nuances in time and detect all data that caused the failure of equipment.

Lidar Detections

The process explains the 3-D lidar data from a sensor mounted on a vehicle by segmenting the ground plane and finding nearby obstacles. This can facilitate drivable path planning for vehicle navigation. Creating a 'Velodyne' File Reader, later each scan of lidar data is stored as a 3-D point cloud. Efficiently processing this data using fast indexing and search is key to the performance of the sensor processing pipeline. This efficiency is achieved using the 'pointCloud' object, which internally organizes the data using a K-d tree data structure.

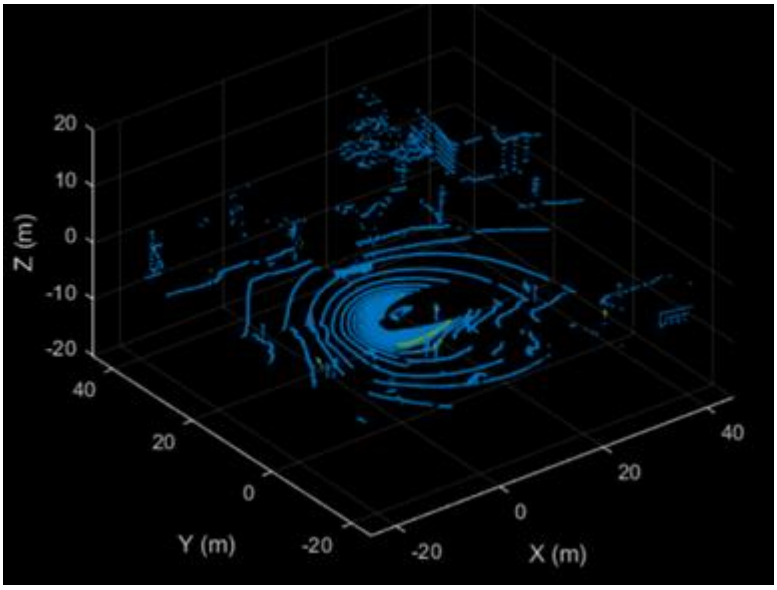

Figure 6 Point cloud

The lidar is mounted on top of the vehicle, and the point cloud may contain points belonging to the vehicle itself, such as on the roof or hood. Knowing the dimensions of the vehicle, out points that are closest to the vehicle are segmented. To identify obstacles from the lidar data, first segmenting the ground plane using the 'segmentGroundFromLidarData' function to

accomplish this. This function segments points belonging to the ground from organized lidar data. When the point cloud processing of pipeline for a single lidar scan has been laid out, this all is put together to process 30 seconds from the sequence of recorded data in Figure 7.

To track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Lidar sensors report measurements as a point cloud. The below figures illustrate the workflow in MATLAB for processing the point cloud and tracking the objects.

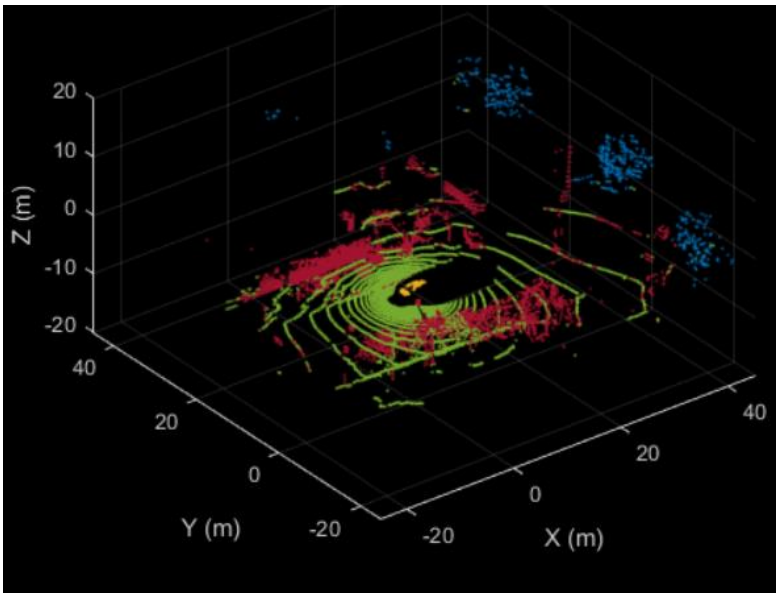

Figure 7 Lidar sequence

Due to the high-resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. This raw data must be pre-processed to extract objects of interest, such as cars, cyclists, and pedestrians. The bounding box is fit onto each cluster by using the minimum and maximum of coordinates of points in each dimension. The detector is implemented by a supporting class HelperBoundingBoxDetector, which wraps around point cloud segmentation and clustering functionalities.

Analyzing different events in the scenario and to understand how the combination of the lidar measurement model, joint probabilistic data association, and interacting multiple model filter, helps to achieve a good estimation of the vehicle tracks.

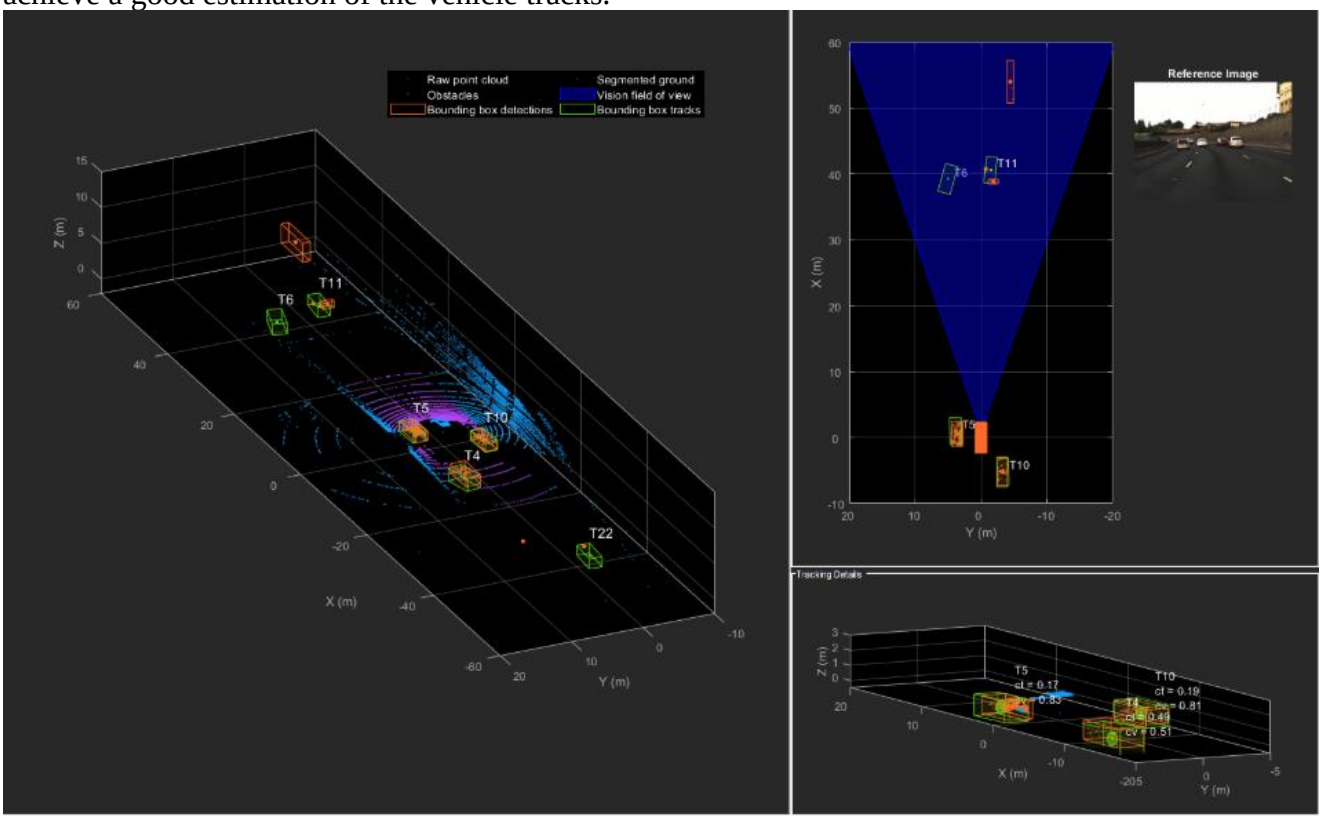

Figure 8 Lidar bounding box of vehicles

Radar Detections

After modeling the radar sensor, it uses the radarDetectionGenerator to generate synthetic radar detections. Simulating the radar and measuring the position of the target vehicle by advancing the simulation time of the scenario. The radar sensor generates detections from the true target pose (position, velocity, and orientation) expressed in the ego vehicle's coordinate frame.

Position Measurements - Over the duration of the FCW test, the target vehicle's distance from the ego vehicle spans a wide range of values. By comparing the radar's measured longitudinal and lateral positions of the target vehicle to the vehicle's ground truth position, the observation of the accuracy of radar's measured positions can be done.

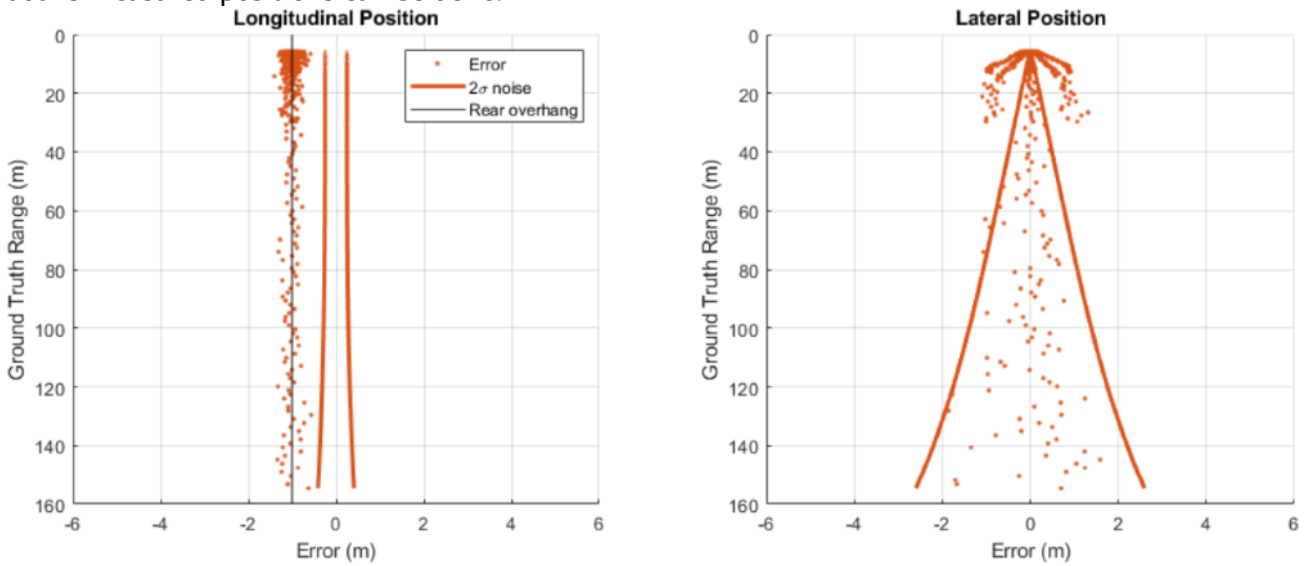

Figure 9 Position measurements

Velocity Measurements - A radar generates velocity measurements by observing the Doppler frequency shift on the signal energy returned from each target. The rate at which the target's range is changing relative to the radar is derived directly from these Doppler frequencies. Pedestrian and Vehicle Detection - A radar "sees" not only an object's physical dimensions (length, width, and height) but also is sensitive to an object's electrical size. An object's electrical size is referred to as its radar cross-section (RCS) and is commonly given in units of decibel square meters (dBsm). An object's RCS defines how effectively it reflects the electromagnetic energy received from the radar back to the sensor. An object's RCS value depends on many properties, including the object's size, shape, and the kind of materials it contains. An object's RCS also depends on the transmit frequency of the radar. The below figure represents pedestrian and vehicle detection. Detection of Closely Spaced Targets - When multiple targets occupy a radar's resolution cell, the group of closely spaced targets are reported as a single detection. The reported location is the centroid of the location of each contributing target. This merging of multiple targets into a single detection is common at long ranges because the area covered by the radar's azimuth resolution grows with increasing distance from the sensor. Detection of Closely Spaced Targets - When multiple targets occupy a radar's resolution cell, the group of closely spaced targets are reported as a single detection. The reported location is the centroid of the location of each contributing target. This merging of multiple targets into a single detection is common at long ranges because the area covered by the radar's azimuth resolution grows with increasing distance from the sensor. The Radar detection helped to understand the following:

1. Provides accurate longitudinal position and velocity measurements over long ranges, but has limited lateral accuracy at long ranges.

2. It generates multiple detections from a single target at close ranges but merges detections from multiple closely spaced targets into a single detection at long ranges.
3. Sees vehicles and other targets with large radar cross-sections over long ranges, but has limited detection performance for nonmetallic objects such as pedestrians.

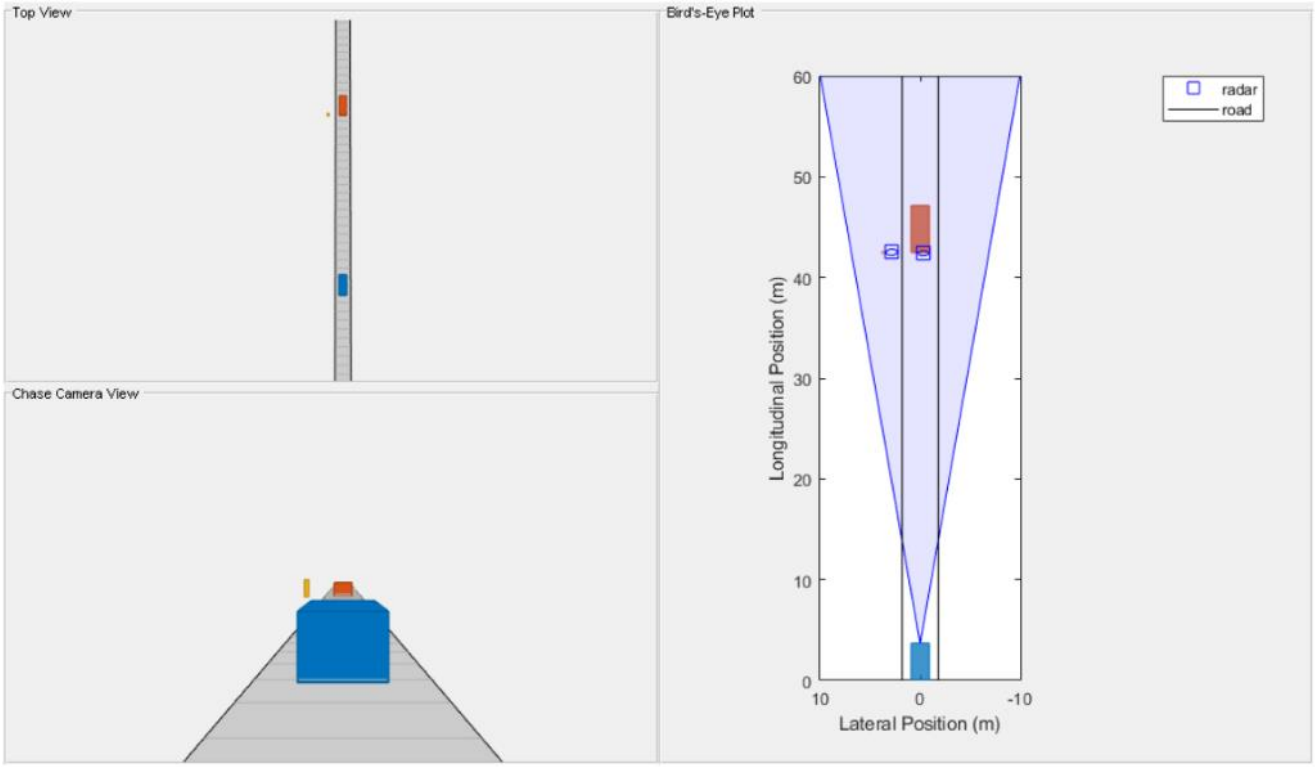

Figure 10 Pedestrian and Vehicle Detection

Camera Detections

For monocular camera systems, two distance estimation methods have emerged and for vehicle detection, an aggregate channel feature (ACF) detector is being used.

Figure 11 Monocular Camera View

To detect and track multiple vehicles with a monocular camera mounted in a vehicle. The vehicle detectors are based on ACF features and Faster R-CNN, a deep-learning-based object detection technique in figure 12.

Figure 12 Vehicles detected

Extended Object Detections

An extended object is an object whose dimensions span multiple sensor resolution cells. As a result, the sensors report multiple detections of the extended objects in a single scan. In conventional tracking approaches, tracked objects are assumed to return one detection per sensor scan. With the development of sensors that have better resolution, such as a high-resolution radar, the sensors typically return more than one detection of an object. Extended objects present new challenges to conventional trackers because these trackers assume a single detection per object per sensor. In contrast, extended object trackers can handle multiple detections per object. Also, these trackers can estimate not only the kinematic states, such as the position and velocity of the object but also the dimensions and orientation of the object. An example of dashed elliptical demonstrates the expected extent of the target. To track objects that return multiple detections in a single sensor scan using different approaches. These approaches can be used to track objects with high-resolution sensors, such as a radar or laser sensor.

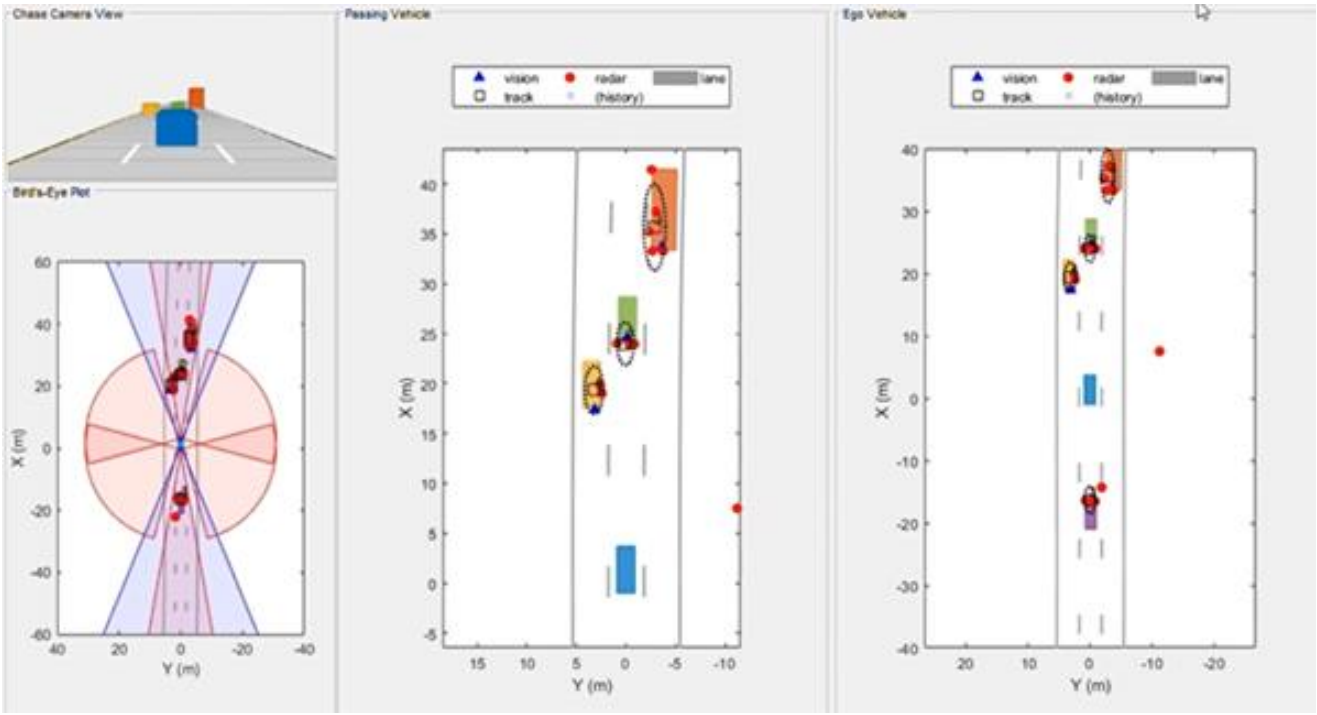

Figure 13 Extended Object Detections

Sensor Fusion

Sensor fusion is the ability to bring together inputs from multiple radars, lidars, and cameras to form a single model or image of the environment around a vehicle. The resulting model is more accurate because it balances the strengths of the different sensors. Vehicle systems can then use the information provided through sensor fusion to support more-intelligent actions.

Each sensor type, or "modality," has inherent strengths and weaknesses. Radars are very strong at accurately determining distance and speed — even in challenging weather conditions — but can't read street signs or "see" the color of a stoplight. Cameras do very well-read signs or classifying objects, such as pedestrians, bicyclists, or other vehicles. However, they can easily be blinded by dirt, sun, rain, snow, or darkness. Lidars can accurately detect objects, but they don't have the range of affordability of cameras or radar. Sensor fusion brings the data from each of these sensor types together, using software algorithms to provide the most comprehensive, and therefore accurate, environmental model possible. A vehicle could use sensor fusion to fuse information from multiple sensors of the same type as well, for instance, radar. This improves perception by taking advantage of partially overlapping fields of view. As multiple radars observe the environment around a vehicle, more than one sensor will detect objects at the same time. Interpreted through global 360° perception software, detections from those multiple sensors can be overlapped or fused, increasing the detection probability and reliability of objects around the vehicle and yielding a more accurate and reliable representation of the environment.

	RADAR	LIDAR	CAMERA	FUSION
Object detection	+	+	○	+
Pedestrian detection	−	○	+	+
Weather conditions	+	○	−	+
Lighting conditions	+	+	−	+
Dirt	+	○	−	+
Velocity	+	○	○	+
Distance - accuracy	+	+	○	+
Distance - range	+	○	○	+
Data density	−	○	+	+
Classification	−	○	+	+
Packaging	+	−	○	+

+ = Strength ○ = Capability − = Weakness

Figure 14 Sensor strength, capability and weakness table

Fusing acceleration, magnetic fields, and orientations.

An accelerometer is an electronic sensor that measures the acceleration forces acting on an object, to determine the object's position in space and monitor the object's movement. A magnetometer is a device that measures magnetism—the direction, strength, or relative change of a magnetic field at a particular location and gyroscope is a device that uses Earth's gravity to help determine orientation. A fusion of Accelerometer, Magnetometer, and Gyroscope, commonly referred to as a MARG sensor for Magnetic, Angular Rate, and Gravity, which is used to estimate pose using the data from these three sensors. The IMU (accelerometer and gyroscope) typically runs at the highest rate. The magnetometer generally runs at a lower rate than the IMU, and the altimeter runs at the lowest rate. Changing the sample rates causes parts of the fusion algorithm to run more frequently and can affect performance. The fusion of data read from an Inertial measurement unit to estimate the orientation and angular velocity. The 6-axis and 9-axis fusion algorithms are used to compute orientation. There are several algorithms to compute orientation from inertial measurement units (IMUs) and magnetic-angular rate-gravity (MARG) units. Accelerometer, gyroscope, and magnetometer sensor data were recorded while a device rotated around three different axes: first around its local Y-axis, then around its Z-axis, and finally around its X-axis. The device's X-axis was generally pointed southward. Using the load function, the inputs are loaded which is the acceleration, angular velocity, and the magnetic field. Then the 'ecompass' function fuses accelerometer and magnetometer data. This is a memoryless algorithm that requires no parameter tuning, but the algorithm is highly susceptible to sensor noise. The ‘ecompass’ algorithm correctly finds the location of the north. The 'imufilter' and 'complementaryFilter' System objects™' fuse accelerometer

and gyroscope data. Although the imufilter and complementaryFilter algorithms produce significantly smoother estimates of the motion, compared to the ecompass, they do not correctly estimate the direction of the north. The imufilter does not process magnetometer data, so it simply assumes the device's X-axis is initially pointing northward. The motion estimate given by imufilter is relative to the initially estimated orientation. An attitude and heading reference system (AHRS) consist of a 9-axis system that uses an accelerometer, gyroscope, and magnetometer to compute orientation. The 'ahrsfilter' and complementaryFilter System objects™ combine the best of the previous algorithms to produce a smoothly changing estimate of the device orientation, while correctly estimating the direction of north. The complementaryFilter uses the same complementary filter algorithm as before, with an extra step to include the magnetometer and improve the orientation estimate. The algorithms presented here, when properly tuned, enable estimation of orientation and are robust against environmental noise sources. It is important to consider the situations in which the sensors are used and tune the filters accordingly.

IMU and GPS fusion for inertial navigation

Create the filter to fuse IMU + GPS measurements. The fusion filter uses an extended Kalman filter to track the orientation (as a quaternion), velocity, position, sensor biases, and the geomagnetic vector.
This insfilter has a few methods to process sensor data, including predict, fusemag, and fusegps. The 'predict' method takes the accelerometer and gyroscope samples from the IMU as inputs. Call the predict method each time the accelerometer and gyroscope are sampled. This method predicts the state one-time step based on the accelerometer and gyroscope. The error covariance of the extended Kalman filter is updated here. The fusegps method takes GPS samples as input. This method updates the filter states based on GPS samples by computing a Kalman gain that weights the various sensor inputs according to their uncertainty. An error covariance is also updated here, this time using the Kalman gain as well. The fusemag method is similar but updates the states, Kalman gain, and error covariance based on the magnetometer samples. Though the insfilter takes accelerometer and gyroscope samples as inputs, these are integrated to compute delta velocities and delta angles, respectively. The filter tracks the bias of the magnetometer and these integrated signals. This example uses a saved trajectory recorded from a UAV as the ground truth. This trajectory is fed to several sensor simulators to compute simulated accelerometer, gyroscope, magnetometer, and GPS data streams. Set up the GPS at the specified sample rate and the reference location. Typically, a UAV uses an integrated MARG sensor (Magnetic, Angular Rate, Gravity) for pose estimation. To model a MARG sensor, define an IMU sensor model containing an accelerometer, gyroscope, and magnetometer. In a real-world application, the three sensors could come from a single integrated circuit or separate ones. The property values set here are typical for low-cost MEMS sensors.

States	units	index
Orientation (quaternion parts)	-	1:4
Gyroscope Bias (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Accelerometer Bias (XYZ)	m/s^2	14:16
Geomagnetic Field Vector (NED)	uT	17:19
Magnetometer Bias (XYZ)	uT	20:22

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly. The insfilter measurement noises describe how much noise is corrupting the sensor reading. These values are based on the imuSensor and gpsSensor parameters.
The process noises describe how well the filter equations describe the state evolution. Process noises are determined empirically using parameter sweeping to jointly optimize position and orientation estimates from the filter. The HelperScrollingPlotter scope enables the plotting of variables over time.

It is used here to track errors in a pose. The HelperPoseViewer scope allows 3-D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to false. The main simulation loop is a while loop with a nested for loop. The while loop executes at gpsFs, which is the GPS sample rate. The nested for loop executes at imuFs, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

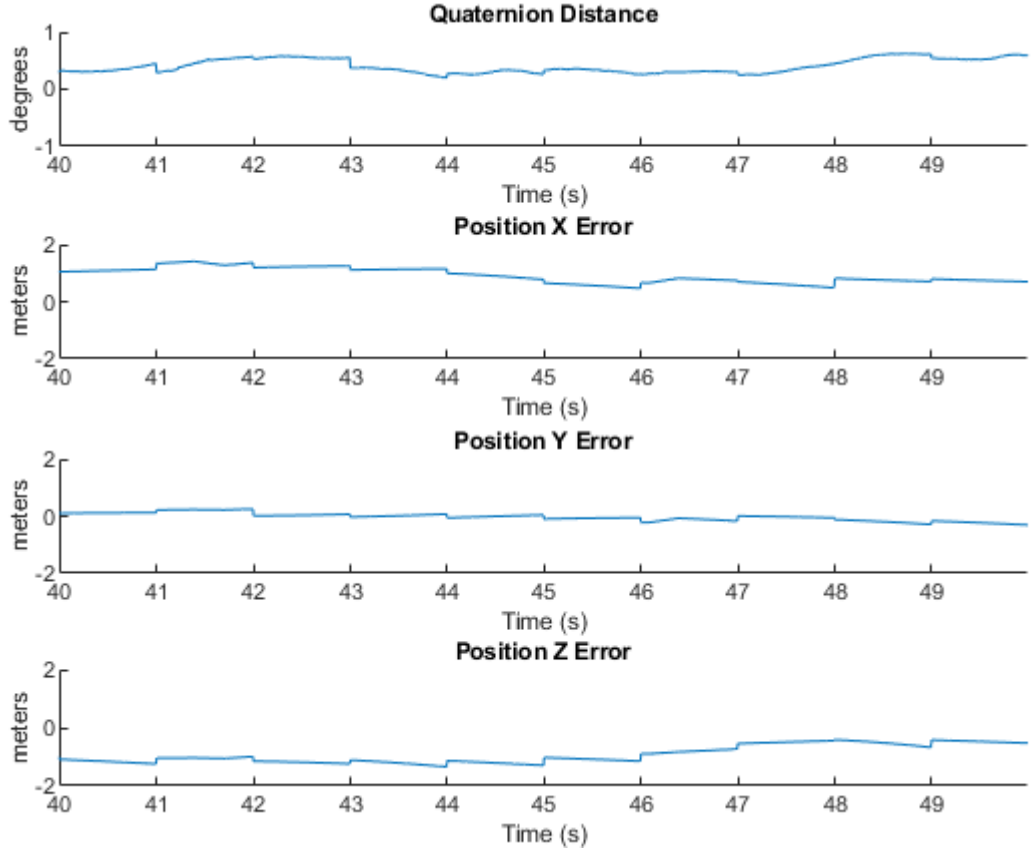

Figure 15 Quaternion Distance, X, Y, Z Position error graphs

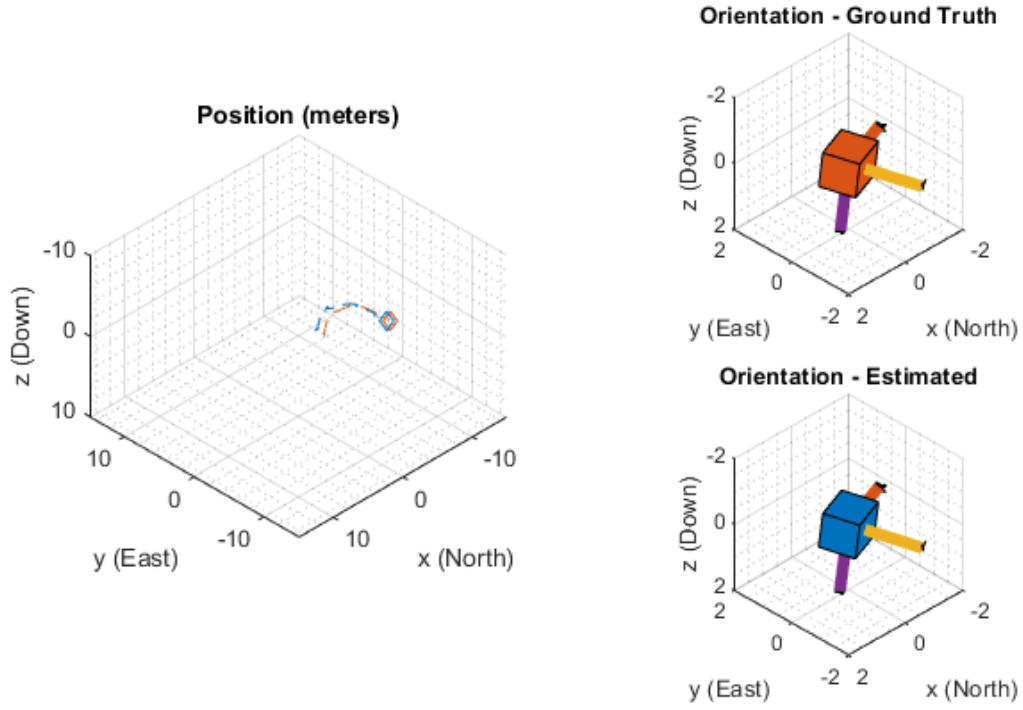

Figure 16 Position graph, Ground truth orientation, and Estimated orientation plots

Position and orientation estimates were logged throughout the simulation. Now computing end-to-end root mean squared error for both position and orientation.

End-to-End Simulation Position RMS Error:
X: 0.57 , Y: 0.53, Z: 0.68 (meters)

End-to-End Quaternion Distance RMS Error:
0.32 (degrees)

The fusion of Radar and Lidar

RADAR stands for Radio Detection and Ranging. It is a detection system that uses radio waves to determine the range, angle, or velocity of objects. RADAR is an object-detection, electronic device that uses an ultra-high frequency or microwave parts of the radio frequency spectrum to determine the position or range of an object. It can also be used to detect the speed and direction of a moving object.
LIDAR stands for Light Detection and Ranging. It is a method for measuring distances (ranging) by illuminating the target with laser light and measuring the reflection with a sensor. Differences in laser return times and wavelengths can then be used to make digital 3-D representations of the target.
Radar sensors provide reliable information on straight lanes but fail in curves due to their restricted field of view. On the other hand, Lidar sensors can cover the regions of interest in almost all situations but do not provide precise speed information. The advantages of both sensors with a sensor fusion approach are combined to provide permanent and precise spatial and dynamical data. Generating an object-level tracklist from measurements of a radar and a lidar sensor and further fuse them using a track-level fusion scheme. The radar measurements are processed using an extended object tracker and lidar measurements using a Joint Probabilistic Data Association (JPDA) tracker. A further fusion of the tracks is made using a track-level fusion scheme.

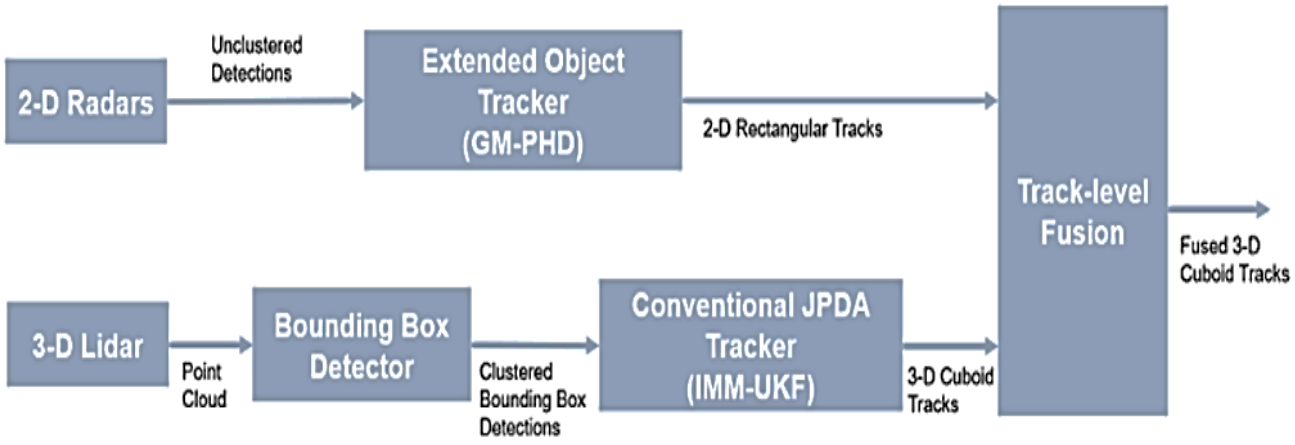

Figure 17 Schematics of the workflow of track-level fusion

The data from radar and lidar sensors are simulated using 'radarDetectionGenerator' and 'lidarPointCloudGenerator', respectively. The Ego vehicle is mounted with four 2-D radar sensors. The front and rear radar sensors have a field of view of 45 degrees. The left and right radar sensors have a field of view of 150 degrees. Each radar has a resolution of 6 degrees in azimuth and 2.5 meters in range. The Ego is also mounted with one 3-D lidar sensor with a field of view of 360 degrees in azimuth and 40 degrees in elevation. The lidar has a resolution of 0.2 degrees in azimuth and 1.25 degrees in elevation (32 elevation channels).

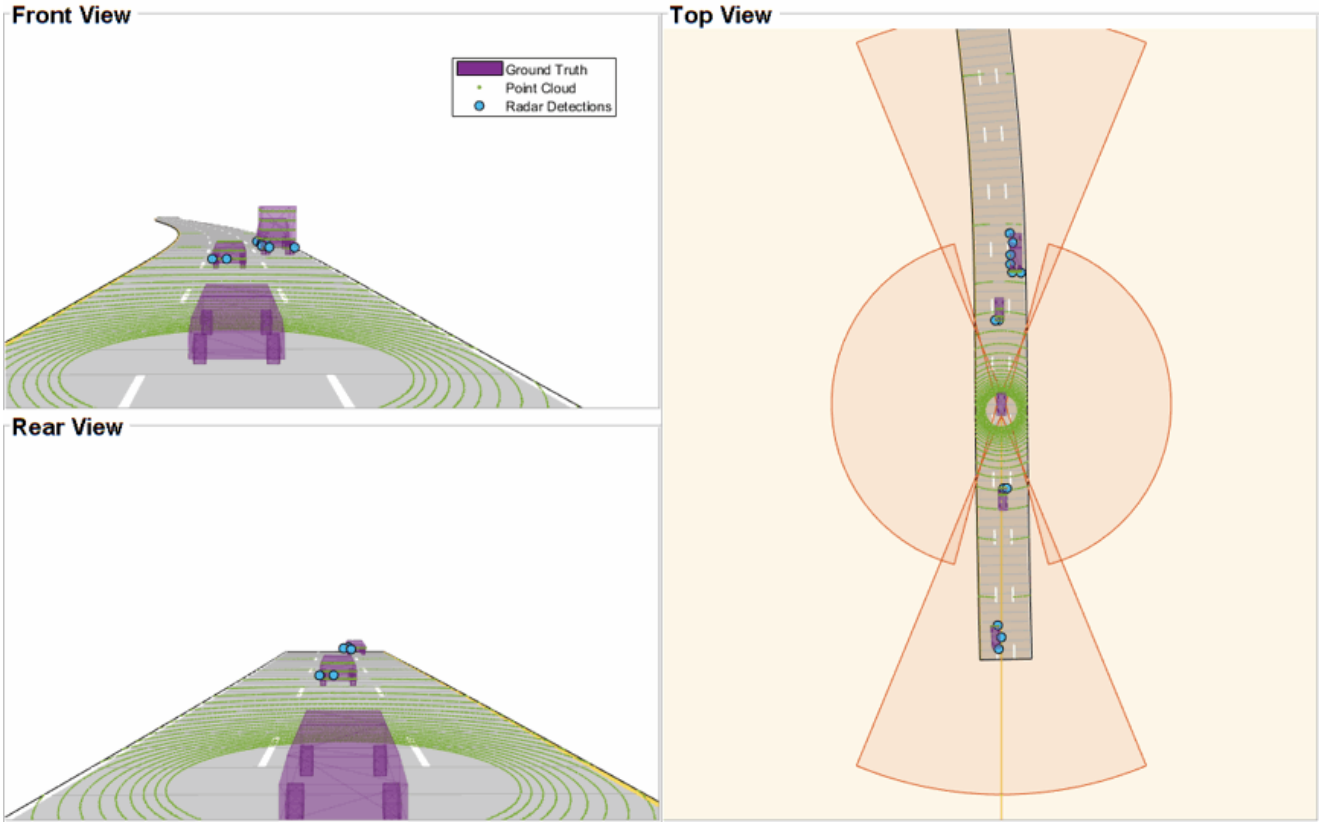

Figure 18 Front, rear and top view with the focus on the ego vehicle

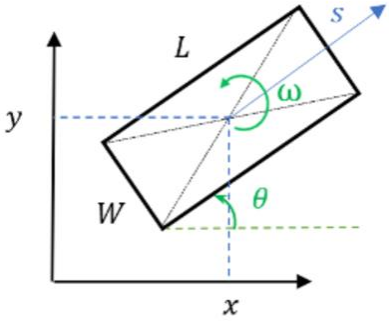

Figure 19 Output array conventions

The radars have a higher resolution than the objects and return multiple detections per object. Conventional trackers such as Global Nearest Neighbor (GNN) and Joint Probabilistic Data Association (JPDA) assume that the sensors return at most one detection per object per scan. Therefore, the detections from high-resolution sensors must be either clustered before processing it with conventional trackers or must be processed using extended object trackers. Extended object trackers do not require pre-clustering of detections and usually estimate both kinematic states (for example, position and velocity) and the extent of the objects. In general, extended object trackers offer a better estimation of objects as they handle clustering and data association simultaneously using the temporal history of tracks. The radar detections are processed using a Gaussian mixture probability hypothesis density (GM-PHD) tracker (trackerPHD and gmphd) with a rectangular target model.

The algorithm for tracking objects using radar measurements is wrapped inside the helper class, helperRadarTrackingAlgorithm, implemented as a System object™. This class outputs an array of objectTrack objects and define their state according to the following convention: $[x\ y\ s\ \theta\ \omega\ L\ W]$

Similar to radars, the lidar sensor also returns multiple measurements per object. Further, the sensor returns a large number of points from the road, which must be removed before used as inputs for an object-tracking algorithm. While lidar data from obstacles can be directly processed via extended object tracking algorithms, conventional tracking algorithms are still more prevalent for tracking using lidar data. One of the reasons for this trend is mainly observed due to the higher computational complexity of extended object trackers for large data sets. The lidar data is processed using a conventional joint probabilistic data association (JPDA) tracker, configured with an interacting multiple model (IMM) filters. The pre-processing of lidar data to remove point cloud is performed by using a RANSAC-based plane-fitting algorithm and bounding boxes are formed by performing a Euclidian-based distance

clustering algorithm. The algorithm for tracking objects using lidar data is wrapped inside the helper class, helperLidarTrackingAlgorithm implemented as a System object. This class outputs an array of objectTrack objects and defines their state according to the following convention: $[x\ y\ s\ \theta\ \omega\ z\ \dot{z}\ L\ W\ H]$ The states common to the radar algorithm are defined similarly. Also, as a 3-D sensor, the lidar tracker outputs three additional states, z, $\dot{z}$ and H, which refer to z-coordinate (m), z-velocity (m/s), and height (m) of the tracked object respectively. The first step towards setting up a track-level fusion algorithm is defining the choice of the state vector (or state-space) for the fused or central tracks. In this case, the state-space for fused tracks is chosen to be the same as the lidar. After choosing a central track state-space, you define the transformation of the central track state to the local track state. In this case, the local track state-space refers to states of radar and lidar tracks. Define the configuration of the lidar source. The helperRadarTrackingAlgorithm outputs tracks with SourceIndex set to 1. The SourceIndex is provided as a property on each tracker to uniquely identify it and allows a fusion algorithm to distinguish tracks from different sources. Since the state-space of a lidar track is the same as the central track, you do not define any transformations. The next step is to define the state-fusion algorithm. The state-fusion algorithm takes multiple states and state covariances in the central state-space as input and returns a fused estimate of the state and the covariances. A covariance intersection algorithm provided by the helper function, helperRadarLidarFusionFcn is used here. A generic covariance intersection algorithm for two Gaussian estimates with mean x_i and covariance P_i can be defined according to the following equations:

$$P_F^{-1} = w_1 {P_1}^{-1} + w_2 {P_2}^{-1} \qquad x_F = P_F(w_1 {P_1}^{-1} x_1 + w_2 {P_2}^{-1} x_2)$$

where x_F and P_F are the fused state and covariance and $w_1 w_2$ are mixing coefficients from each estimate. Typically, these mixing coefficients are estimated by minimizing the determinant or the trace of the fused covariance.

Camera And radar

The camera has established itself as the main sensor for building the perception module. In recent years, there is an increased emphasis on diversifying the set of sensors to improve the robustness to a range of operating conditions. A variety of sensors such as LiDAR, short-range radars, long-range radars, infrared cameras, and sonars have been used to enhance the quality of the perception module output. Here the focus is on the fusion of camera and radar sensors. Radar presents a low-cost alternative to LiDAR as a range determining sensor. A typical automotive radar is currently considerably cheaper than a LiDAR due to the nature of its fundamental design. Besides costs, radar is robust to different lighting and weather conditions (e.g., rain and fog) and is capable of providing an instantaneous measurement of velocity, providing the opportunity for improved system reaction times. Scenario generation comprises generating a road network, defining vehicles that move on the roads, and moving the vehicles. Testing the ability of the sensor fusion to track a vehicle that is passing on the left of the ego vehicle. The scenario simulates a highway setting, and additional vehicles are in front of and behind the ego vehicle. Add a stretch of 500 meters of typical highway road with two lanes. The road is defined using a set of points, where each point defines the center of the road in 3-D space. Create the ego vehicle and three cars around it: one that overtakes the ego vehicle and passes it on the left, one that drives right in front of the ego vehicle and one that drives right behind the ego vehicle. All the cars follow the trajectory defined by the road waypoints by using the trajectory driving policy. The passing car will start on the right lane, move to the left lane to pass, and return to the right lane. Simulate an ego vehicle that has 6 radar sensors and 2 vision sensors covering the 360 degrees field of view. The sensors have some overlap and some coverage gaps. The ego vehicle is equipped with a long-range radar sensor and a vision sensor on both the front and the back of the vehicle. Each side of the vehicle has two short-range radar sensors, each covering 90 degrees. One sensor on each side covers from the middle of the vehicle to the back. The other sensor on each side covers from the middle of the vehicle forward. Create a multiObjectTracker to track the vehicles that are close to the ego vehicle. The tracker uses

the initSimDemoFilter supporting function to initialize a constant velocity linear Kalman filter that works with position and velocity.
Tracking is done in 2-D. Although the sensors return measurements in 3-D, the motion itself is confined to the horizontal plane, so there is no need to track the height.

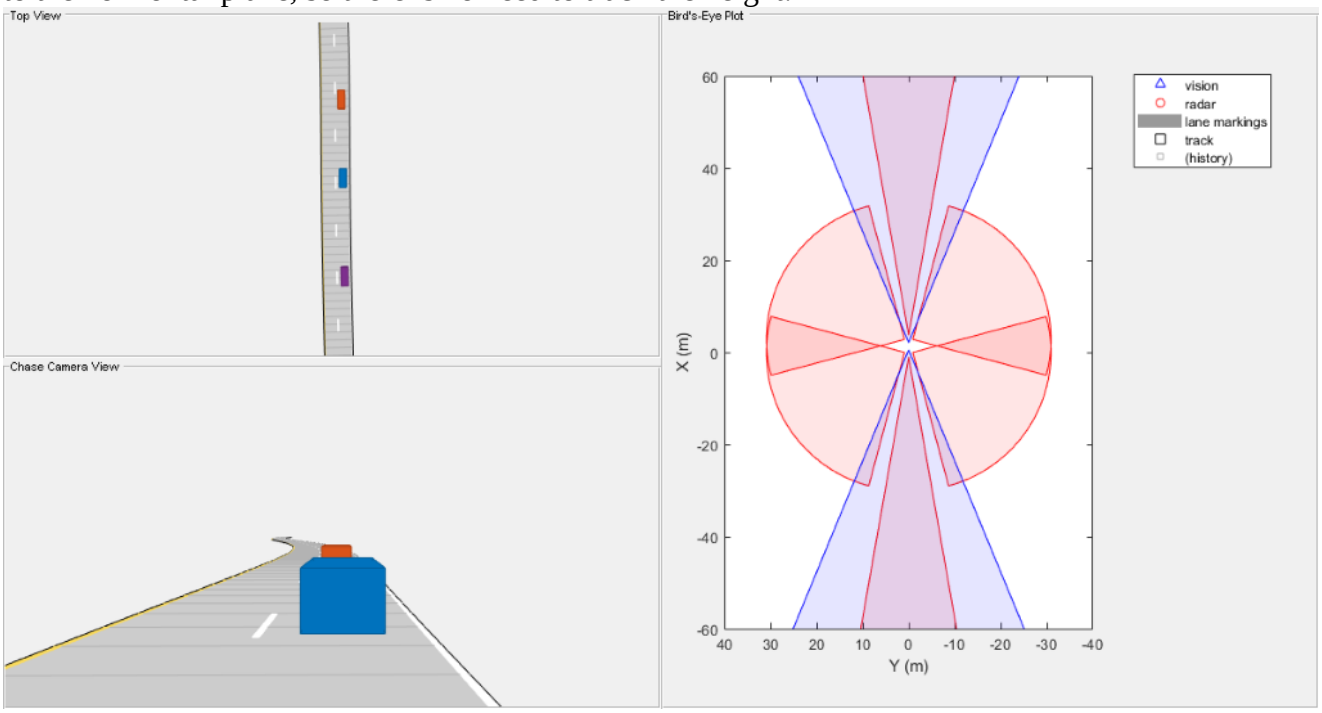

Figure 20 Top view, Chase camera view and Bird's – Eye plot

IMU measurements

An IMU is an electronic device mounted on a platform. The IMU consists of individual sensors that report various information about the platform's motion. IMUs combine multiple sensors, which can include accelerometers, gyroscopes, and magnetometers.

Usually, the data returned by IMUs are fused and interpreted as roll, pitch, and yaw of the platform. Real-world IMU sensors can have different axes for each of the individual sensors. The models provided by Sensor Fusion and Tracking Toolbox assume that the individual sensor axes are aligned. The default IMU model contains an ideal accelerometer and an ideal gyroscope. The accelparams and gyroparams objects define the accelerometer and gyroscope configuration. You can set the properties of these objects to mimic specific hardware and environments. For more information on IMU parameter objects, see accelparams, gyroparams, and magparams.

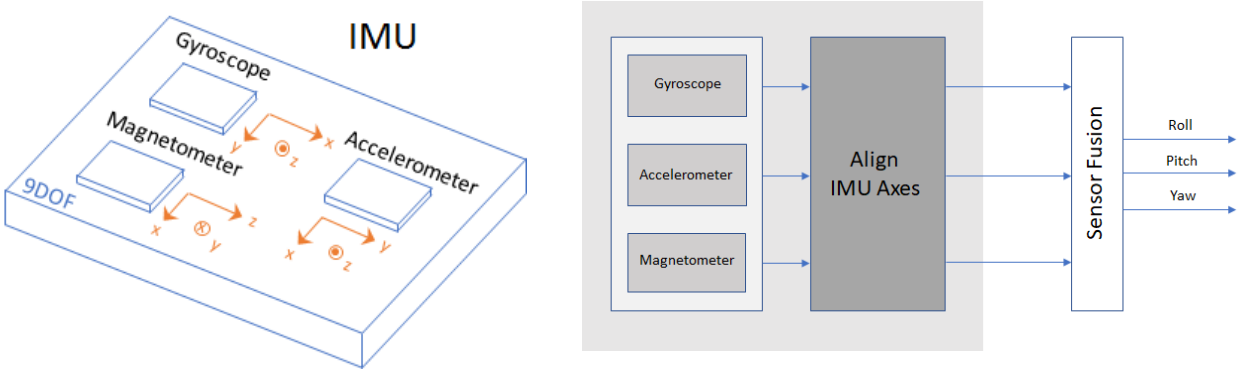

Figure 21 IMU Model

Orientation using an inertial sensor

To fuse data read from an inertial measurement unit (IMU) to estimate the orientation and angular velocity:

1. ecompass — Fuse accelerometer and magnetometer readings
2. imufilter — Fuse accelerometer and gyroscope readings
3. ahrsfilter — Fuse accelerometer, gyroscope, and magnetometer readings

More sensors on an IMU result in a more robust orientation estimation. The sensor data can be cross-validated, and the information the sensors convey is orthogonal. This provides an overview of inertial sensor fusion for IMUs in the Sensor Fusion and Tracking Toolbox.

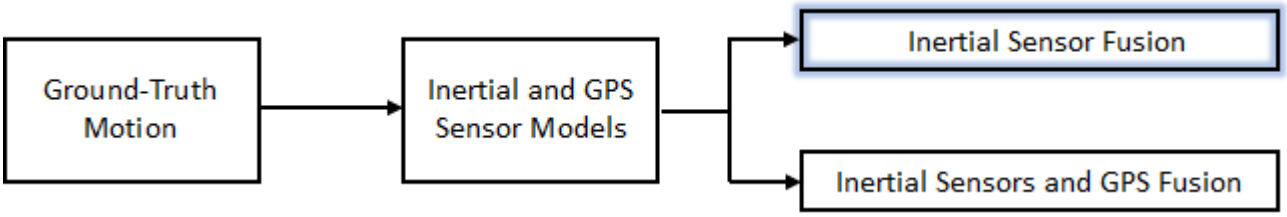

Figure 22 Inertia Sensor fusion for IMU

There are several algorithms to compute orientation from inertial measurement units (IMUs) and magnetic-angular rate-gravity (MARG) units. This example covers the basics of orientation and how to use these algorithms.

An object's orientation describes its rotation relative to some coordinate system, sometimes called a parent coordinate system, in three dimensions.

For the following algorithms, the fixed, parent coordinate system used is North-East-Down (NED). NED is sometimes referred to as the global coordinate system or reference frame. In the NED reference frame, the X-axis points north, the Y-axis points east, and the Z-axis points downward. The X-Y plane of NED is considered to be the local tangent plane of the Earth. Depending on the algorithm, the north may be either magnetic north or true north. The algorithm here uses magnetic north. An object can be thought of as having its coordinate system, often called the local or child coordinate system. This child coordinate system rotates with the object relative to the parent coordinate system. If there is no translation, the origins of both coordinate systems overlap.

The orientation quantity computed is a rotation that takes quantities from the parent reference frame to the child reference frame. The rotation is represented by a quaternion or rotation matrix.

Three types of sensors are commonly used: accelerometers, gyroscopes, and magnetometers. Accelerometers measure proper acceleration. Gyroscopes measure angular velocity. Magnetometers measure the local magnetic field. Different algorithms are used to fuse different combinations of sensors to estimate orientation.

Sensor Data: Through most of this example, the same set of sensor data is used. Accelerometer, gyroscope, and magnetometer sensor data were recorded while a device rotated around three different axes: first around its local Y-axis, then around its Z-axis, and finally around its X-axis. The device's X-axis was generally pointed southward for the duration of the experiment.

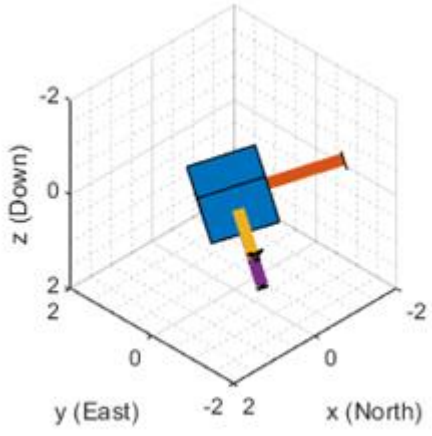

Figure 23 Accelerometer-Magnetometer Fusion:

The ecompass function fuses accelerometer and magnetometer data. This is a memoryless algorithm that requires no parameter tuning, but the algorithm is highly

susceptible to sensor noise. Note that the ecompass algorithm correctly finds the location of the north. However, because the function is memoryless, the estimated motion is not smooth. The algorithm could be used as an initialization step in an orientation filter or some of the techniques presented in Lowpass Filter Orientation Using Quaternion SLERP could be used to smooth the motion. estimating orientation using either an error-state Kalman filter or a complementary filter. The error-state Kalman filter is the standard estimation filter and allows for many different aspects of the system to be tuned using the corresponding noise parameters. The complementary filter can be used as a substitute for systems with memory constraints and has minimal tunable parameters, which allows for easier configuration at the cost of finer tuning. The imufilter and complementaryFilter System objects use accelerometer and gyroscope data. The imufilter uses an internal error-state Kalman filter and the complementaryFilter uses a complementary filter. The filters are capable of removing the gyroscope's bias noise, which drifts over time. Although the imufilter and complementaryFilter algorithms produce significantly smoother estimates of the motion, compared to the ecompass, they do not correctly estimate the direction of north. The imufilter does not process magnetometer data, so it simply assumes the device's X-axis is initially pointing northward. The motion estimate given by imufilter is relative to the initially estimated orientation. The complementaryFilter makes the same assumption when the HasMagnetometer property is set to false. An attitude and heading reference system (AHRS) consist of a 9-axis system that uses an accelerometer, gyroscope, and magnetometer to compute orientation. The ahrsfilter and complementaryFilter System objects combine the best of the previous algorithms to produce a smoothly changing estimate of the device orientation, while correctly estimating the direction of north. The complementaryFilter uses the same complementary filter algorithm as before, with an extra step to include the magnetometer and improve the orientation estimate. Like imufilter, ahrsfilter algorithm also uses an error-state Kalman filter. In addition to gyroscope bias removal, the ahrsfilter has some ability to detect and reject mild magnetic jamming.

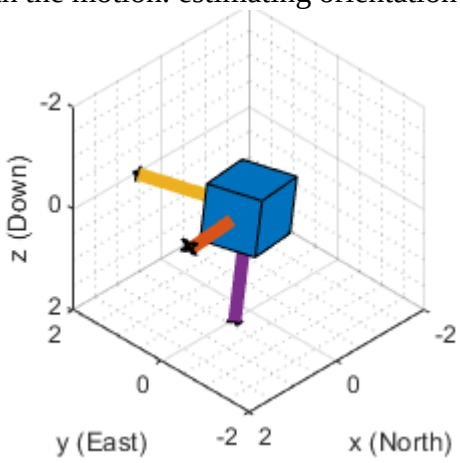

Figure 24 Accelerometer-Gyroscope Fusion

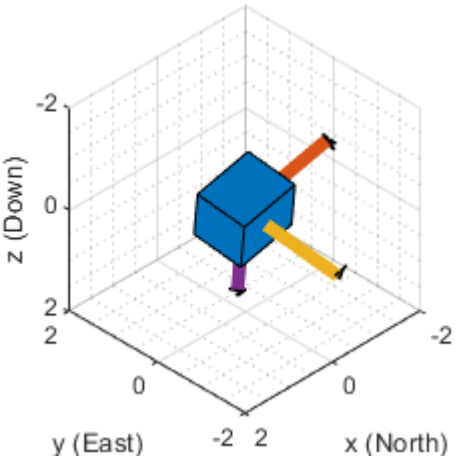

Figure 25 Accelerometer-Gyroscope-Magnetometer Fusion

Visual-inertial odometry using synthetic data

To estimate the pose (position and orientation) of a ground vehicle using an inertial measurement unit (IMU) and a monocular camera. Create a driving scenario containing the ground truth trajectory of the vehicle. Use an IMU and visual odometry model to generate measurements.

Fuse these measurements to estimate the pose of the vehicle and then display the results.

Visual-inertial odometry estimates pose by fusing the visual odometry pose estimate from the monocular camera and the pose estimate from the IMU. The IMU returns an accurate pose estimate for small time intervals but suffers from large drift due to integrating the inertial sensor measurements. The monocular camera returns an accurate pose estimate over a larger time interval but suffers from a scale ambiguity. Given these complementary strengths and weaknesses, the fusion of these sensors using visual-inertial odometry is a suitable choice. This method can be used in scenarios where GPS readings

are unavailable, such as in an urban canyon. Below steps mentioned are to be followed for estimating the pose of a ground vehicle.

1. Create a driving scenario object that contains:
2. The road the vehicle travels on
3. The buildings surrounding either side of the road
4. The ground truth poses of the vehicle
5. The estimated pose of the vehicle

The ground truth pose of the vehicle is shown as a solid blue cuboid. The estimated pose is shown as a transparent blue cuboid. Note that the estimated pose does not appear in the initial visualization because the ground truth and estimated poses overlap. Generate the baseline trajectory for the ground vehicle using the waypointTrajectory System object. Note that the waypointTrajectory is used in place of drivingScenario/trajectory since the acceleration of the vehicle is needed. The trajectory is generated at a specified sampling rate using a set of waypoints, times of arrival, and velocities.

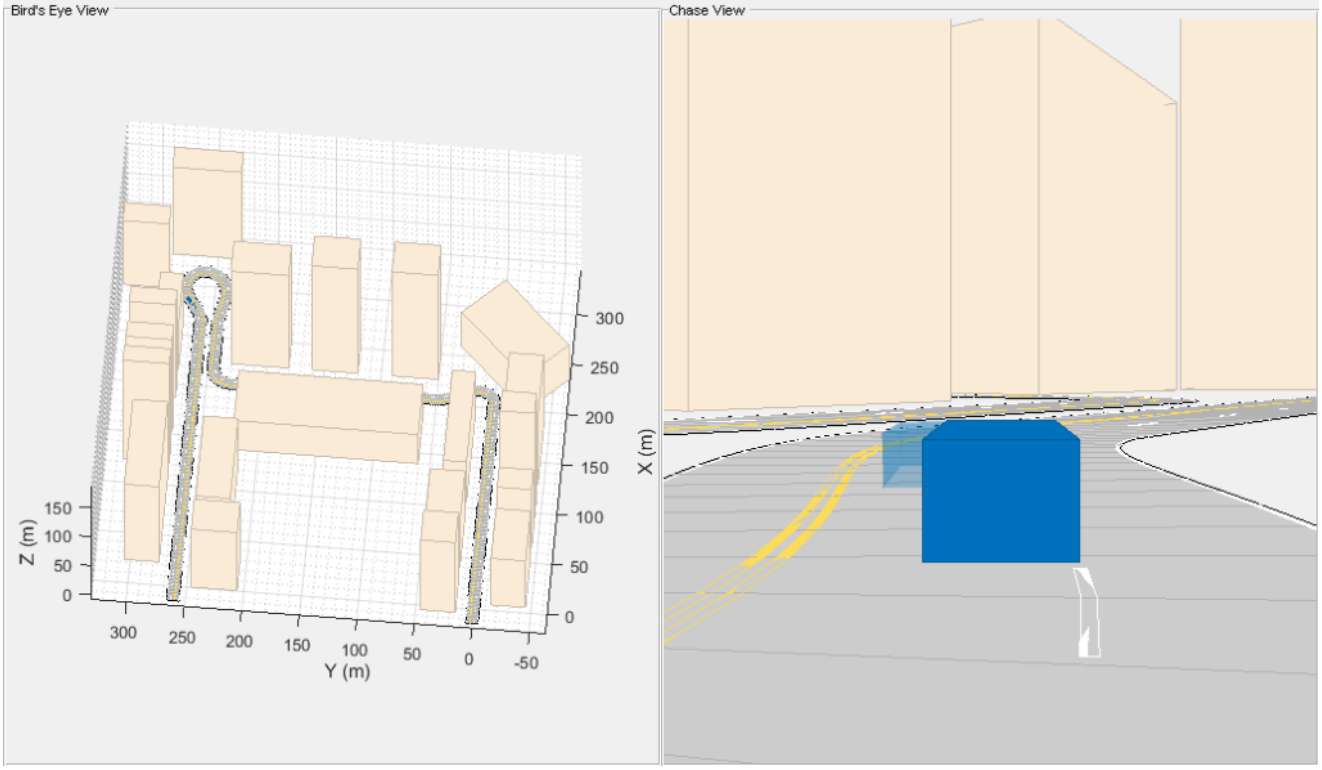

Figure 26 Visual-inertial odometry

Create the filter to fuse IMU and visual odometry measurements. This example uses a loosely coupled method to fuse the measurements. While the results are not as accurate as a tightly coupled method, the amount of processing required is significantly less and the results are adequate. The fusion filter uses an error-state Kalman filter to track the orientation (as a quaternion), position, velocity, and sensor biases.

The insfilterErrorState object has the following functions to process sensor data: predict and fusemvo.

The predict function takes the accelerometer and gyroscope measurements from the IMU as inputs. Call the predict function each time the accelerometer and gyroscope are sampled. This function predicts the state forward by a one-time step based on the accelerometer and gyroscope measurements and updates the error state covariance of the filter.

The fusemvo function takes the visual odometry pose estimates as input. This function updates the error states based on the visual odometry pose estimates by computing a Kalman gain that weighs the various inputs according to their uncertainty. As with the predict function, this function also updates the error

state covariance, this time taking the Kalman gain into account. The state is then updated using the new error state and the error state is reset.
The visual odometry model parameters. These parameters model a feature matching and tracking-based visual odometry system using a monocular camera. The scale parameter accounts for the unknown scale of subsequent vision frames of the monocular camera. The other parameters model the drift in the visual odometry reading like a combination of white noise and a first-order Gauss-Markov process. An IMU sensor model containing an accelerometer and gyroscope using the imuSensor System object. The sensor model contains properties to model both deterministic and stochastic noise sources. The property values set here are typical for low-cost MEMS sensors.
Simulate the IMU sampling rate. Each IMU sample is used to predict the filter's state forward by a one-time step. Once a new visual odometry reading is available, it is used to correct the current filter state.
The plot shows that the visual odometry estimate is relatively accurate in estimating the shape of the trajectory. The fusion of the IMU and visual odometry measurements removes the scale factor uncertainty from the visual odometry measurements and the drift from the IMU measurements. The plot (Figure 27) shows that the visual odometry estimate is relatively accurate in estimating the shape of the trajectory. The fusion of the IMU and visual odometry measurements removes the scale factor uncertainty from the visual odometry measurements and the drift from the IMU measurements.

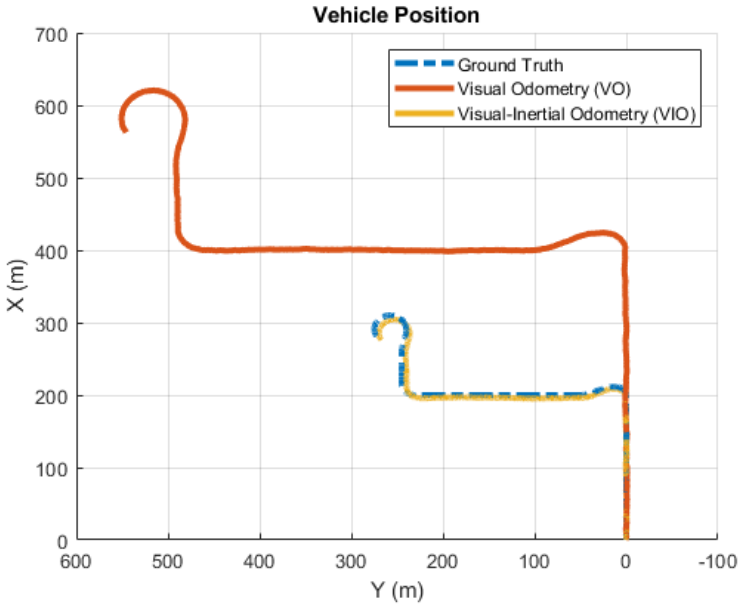

Figure 27 Ground truth vehicle trajectory, the visual odometry estimate, and the fusion filter estimate.

Occupancy grid and semantic segmentation

Estimating free space around a vehicle and create an occupancy grid using semantic segmentation and deep learning includes various configurations to be followed, which are listed below. Free space estimation identifies areas in the environment where the ego vehicle can drive without hitting any obstacles such as pedestrians, curbs, or other vehicles. A vehicle can use a variety of sensors to estimate free space such as radar, lidar, or cameras. Here it is mainly focused on estimating free space from an image sensor using semantic segmentation. A pretrained semantic segmentation network should be used, which can classify pixels into 11 different classes, including Road, Pedestrian, Car, and Sky. The free space in an image can be estimated by defining image pixels classified as Road as free space. All other classes are defined as non-free space or obstacles. For this, the pretrained network is downloaded. First, estimate the free space by processing the image using a downloaded semantic segmentation network. The network returns classifications for each image pixel in the image. The free space is identified as image pixels that have been classified as Road.

Figure 28 Single frame segmented visual

Here the image used is a single frame from an image sequence in the CamVid data set. It took about 1 second to process each frame. Therefore, for expediency, it processed a single frame.

To understand the confidence in the free space estimate, the output score for the Road class for every pixel is displayed. The confidence values can be used to inform downstream algorithms of the estimate's validity. Although the initial segmentation result for Road pixels showed most pixels on the road were classified correctly, visualizing the scores provides richer detail on the classifier's confidence in those classifications. Create the birds-eye-view image, first define the camera sensor configuration. The supporting function listed at the end of camvidMonoCameraSensor returns a mono camera object representing the monocular camera used to collect the CamVid data. Configuring the monoCamera requires the camera intrinsic and extrinsic, which were estimated using data provided in the CamVid data set. To estimate the camera intrinsics, the function uses CamVid checkerboard calibration images and the Camera Calibrator app. Estimates of the camera extrinsic, such as height and pitch, were derived from the extrinsic data estimated by the authors of the CamVid data set. Occupancy grids are used to represent a vehicle's surroundings as a discrete grid in vehicle coordinates and are used for path planning. The procedure to fill the occupancy grid using the free space estimate is as follows:

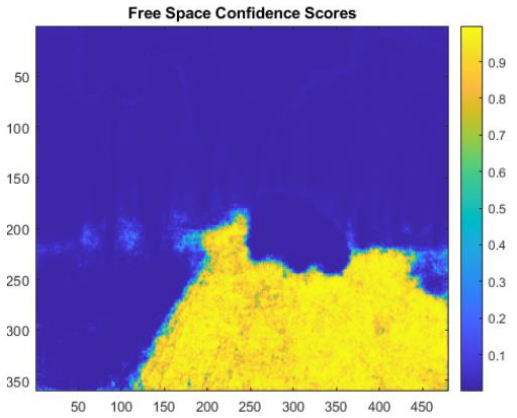

Figure 29 Free space confidence scores

1. Define the dimensions of the occupancy grid in vehicle coordinates.
2. Generate a set of (X, Y) points for each grid cell. These points are in the vehicle's coordinate system.
3. Transform the points from the vehicle coordinate space (X, Y) into the birds-eye-view image coordinate space (x,y) using the vehicleToImage transform.
4. Sample the free space confidence values at (x,y) locations using griddedInterpolant to interpolate free space confidence values that are not exactly at pixel centers in the image.
5. Fill the occupancy grid cell with the average free space confidence value for all (x,y) points that correspond to that grid cell.

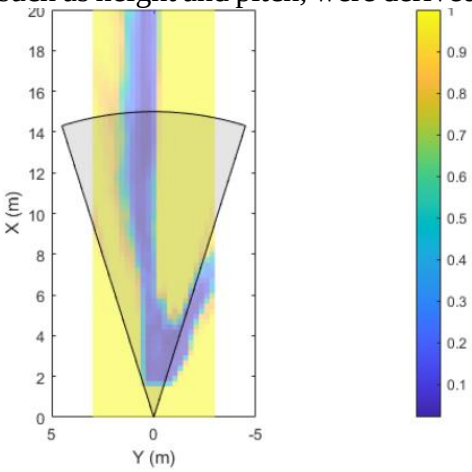

Figure 30 Occupancy grid(probability

Displaying data from multiple sensors is useful for diagnosing and debugging decisions made by autonomous vehicles. The vehicleCostmap provides functionality to check if locations, in-vehicle or world coordinates, are occupied or free. This check is required for any path-planning or decision-making algorithm. Create the vehicleCostmap using the generated occupancyGrid.

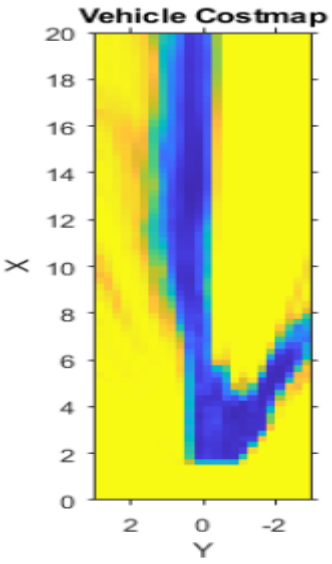

Figure 31 Vehicle costmap

Automate ground-truth labeling of lane boundaries

Good ground truth data is crucial for developing driving algorithms and evaluating their performances. However, creating a rich and diverse set of annotated driving data requires significant time and

resources. The Ground Truth Labeller app makes this process efficient. Create a lane detection algorithm. Then use this algorithm on a single video frame to detect the left ego lane boundary. The lane detected in the previous step is a model and must be converted to a set of discrete points. These points are similar to what a user might manually place on the image. In the camera view, parts of the lane boundary closer to the vehicle will span more pixels than the further parts. Consequently, a user would place more points with higher confidence in the lower parts of the camera image. To replicate this behavior, determine the lane boundary locations from the boundary model more densely at points closer to the vehicle in the next step. To incorporate this lane detection algorithm into the automation workflow of the app, construct a class that inherits from the abstract base class vision. labeler, AutomationAlgorithm. This base class defines properties and signatures for methods that the app uses for configuring and running the custom algorithm. The Ground Truth Labeler app provides a convenient way to obtain an initial automation class template. automation class for lane detection.

Figure 33 Detected left lane boundary

Figure 32 Automatically marked lane boundary points

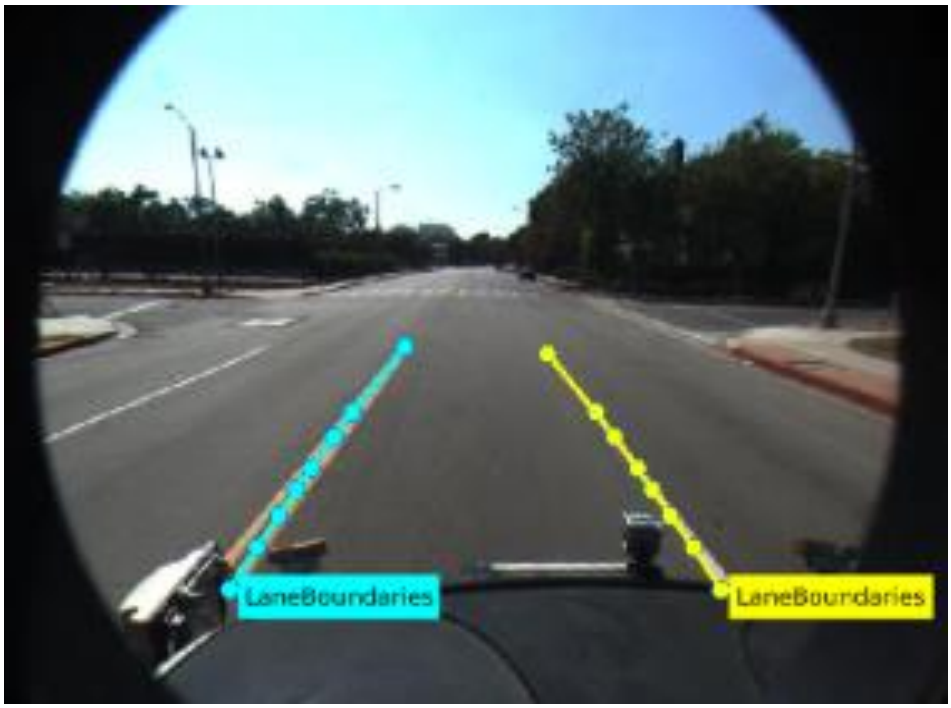

Figure 34 Automatically marked lane boundary

Forward collision warning using sensor fusion

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. The process of providing a forward collision warning comprises the following steps:

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.

Issue warnings based on the tracks and 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. For the visualization here used are monoCamera and birdsEyePlot. 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 outputs of setupTracker are as follow:

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

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. Steps involved in 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.

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

1. First filter 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.
2. 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.
3. Radar objects are considered nonclutter if they are either stationary in front of the car or moving in its vicinity.
4. Format the detections as input to the tracker, i.e., an array of objectDetection elements. See the processVideo and processRadar supporting functions at the end of these steps.

To update the tracker, call the updateTracks method with the following inputs:

1. tracker - The multiObjectTracker that was configured earlier. See the 'Create the Multi-Object Tracker' section.
2. detections - A list of objectDetection objects that were created by processDetections.

3. time - The current scenario time.
4. The output from the tracker is a struct array of tracks.

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, consider only confirmed tracks. Once found the MIO, calculate the relative speed between the car and MIO. 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.

Estimating the distance of vehicles

Estimation for position and orientation for a ground vehicle is to estimate the pose of ground vehicles by fusing data from an inertial measurement unit (IMU) and a global positioning system (GPS) receiver. In a typical system, the accelerometer and gyroscope in the IMU run at relatively high sample rates. The complexity of processing data from those sensors in the fusion algorithm is relatively low. Conversely, the GPS runs at a relatively low sample rate, and the complexity associated with processing it is high. In this fusion algorithm, the GPS samples are processed at a low rate, and the accelerometer and gyroscope samples are processed together at the same high rate. To simulate this configuration, the IMU (accelerometer and gyroscope) is sampled at 100 Hz, and the GPS is sampled at 10 Hz. The fusion filter uses an extended Kalman filter to track the orientation (as a quaternion), position, velocity, and sensor biases. The insfilterNonholonomic object that has two main methods: predict and fusegps. The 'predict' method takes the accelerometer and gyroscope samples from the IMU as input. Call the predict method each time the accelerometer and gyroscope are sampled. This method predicts the states forward one-time step based on the accelerometer and gyroscope. The error covariance of the extended Kalman filter is updated in this step.

The fusegps method takes the GPS samples as input. This method updates the filter states based on the GPS sample by computing a Kalman gain that weights the various sensor inputs according to their uncertainty. An error covariance is also updated in this step, this time using the Kalman gain as well.

The insfilterNonholonomic object has two main properties: IMUSampleRate and DecimationFactor. The ground vehicle has two velocity constraints that assume it does not bounce off the ground or slide on the ground. These constraints are applied using the extended Kalman filter update equations. These updates are applied to the filter states at a rate of IMUSampleRate/DecimationFactor Hz.

The waypointTrajectory object calculates pose based on the specified sampling rate, waypoints, times of arrival, and orientation. Specify the parameters of a circular trajectory for the ground vehicle. Set up the GPS at the specified sample rate and the reference location. The other parameters control the nature of the noise in the output signal. Typically, ground vehicles use a 6-axis IMU sensor for pose estimation. To model an IMU sensor, define an IMU sensor model containing an accelerometer and gyroscope. In a real-world application, the two sensors could come from a single integrated circuit or separate ones. The property values set here are typical for low-cost MEMS sensors. Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly.

The states are:

States	units	index
Orientation (quaternion parts)	-	1:4
Gyroscope Bias (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Accelerometer Bias (XYZ)	m/s^2	14:16

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly. The measurement noises describe how much noise is corrupting the GPS reading based on the gpsSensor parameters and how much uncertainty is in the vehicle dynamic model.
The process noises describe how well the filter equations describe the state evolution. Process noises are determined empirically using parameter sweeping to jointly optimize position and orientation estimates from the filter.
The HelperScrollingPlotter scope enables the plotting of variables over time. It is used here to track errors in a pose. The HelperPoseViewer scope allows 3-D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to false. The main simulation loop is a while loop with a nested for loop. The while loop executes at the gpsFs, which is the GPS measurement rate. The nested for loop executes at the imuFs, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

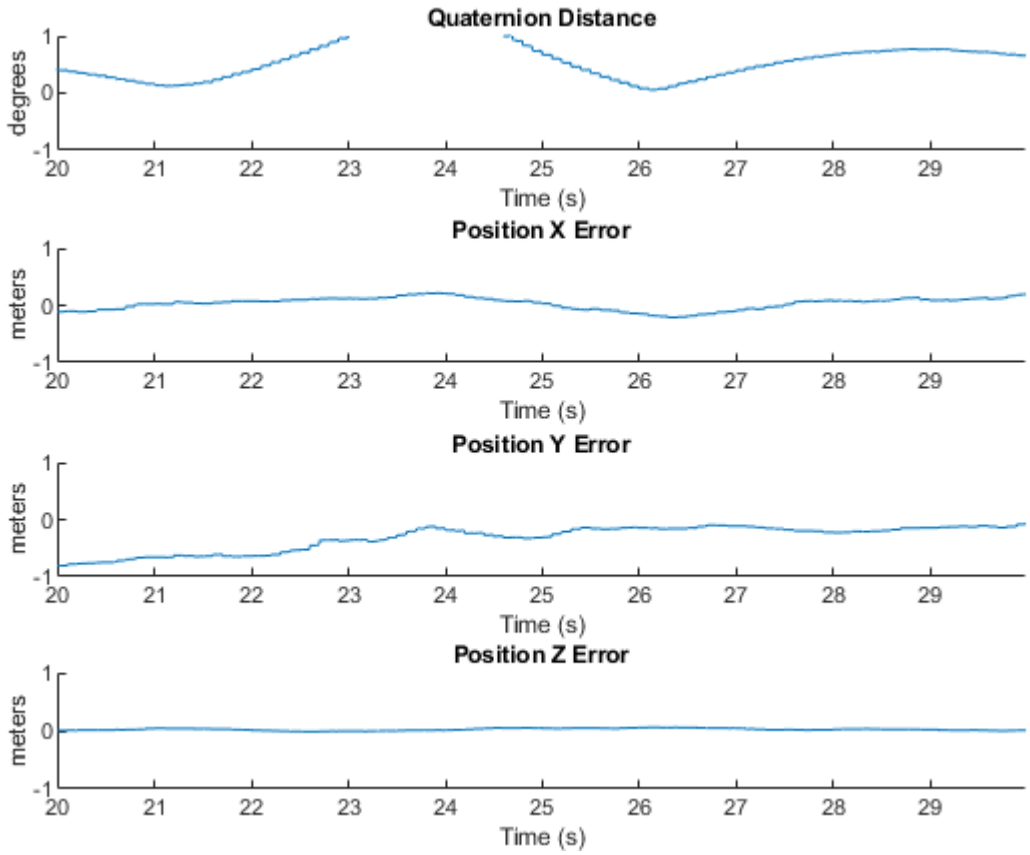

Figure 35 scope output updated at the IMU sample rate

Position and orientation were logged throughout the simulation. Now compute an end-to-end root mean squared error for both position and orientation.
End-to-End Simulation Position RMS Error = X: 1.16, Y: 0.98, Z: 0.03 (meters)
End-to-End Quaternion Distance RMS Error = (degrees) 0.51 (degrees)

Estimation of distance for a ground vehicle is to develop a vehicle detection and distance estimation algorithm and use it to automate labeling using the Ground Truth Labeler app, develop a computer vision algorithm to detect vehicles in a video and use the monocular camera configuration to estimate distances to the detected vehicles and Use the AutomationAlgorithm API to create an automation algorithm. Good ground truth data is crucial for developing driving algorithms and evaluating their performances. However, creating a rich and diverse set of annotated driving data requires significant effort. The Ground Truth Labeler app makes this process efficient. However, manual labeling requires

a significant amount of time and resources. As an alternative, this app provides a framework for creating algorithms to extend and automate the labeling process. construct a monocular camera sensor simulation capable of lane boundary and vehicle detections. The sensor will report these detections in the vehicle coordinate system. Vehicles that contain ADAS features or are designed to be fully autonomous rely on multiple sensors. These sensors can include sonar, radar, lidar, and cameras. Subsequently, the readings returned by a monocular camera sensor can be used to issue lane departure warnings, collision warnings, or to design a lane keep assist control system. In conjunction with other sensors, it can also be used to implement an emergency braking system and other safety-critical features. Knowing the camera's intrinsic and extrinsic calibration parameters is critical to accurate conversion between pixel and vehicle coordinates.

Start by defining the camera's intrinsic parameters. The parameters below were determined earlier using a camera calibration procedure that used a checkerboard calibration pattern. You can use the Camera Calibrator app to obtain them for your camera.

```
focalLength    = [309.4362, 344.2161]; % [fx, fy] in pixel units
principalPoint = [318.9034, 257.5352]; % [cx, cy] optical center in pixel coordinates
imageSize      = [480, 640];           % [nrows, mcols]
```

Note that the lens distortion coefficients were ignored because there is little distortion in the data. The parameters are stored in a cameraIntrinsics object. Next, define the camera orientation concerning the vehicle's chassis.

```
height = 2.1798;    % mounting height in meters from the ground
pitch  = 14;        % pitch of the camera in degrees
```

The above quantities can be derived from the rotation and translation matrices returned by the extrinsic function. Pitch specifies the tilt of the camera from the horizontal position. For the camera used in this example, the roll and yaw of the sensor are both zero. The entire configuration defining the intrinsic and extrinsic are stored in the monoCamera object.

```
sensor = monoCamera(camIntrinsics, height, 'Pitch', pitch);
```

Note that the monoCamera object sets up a very specific vehicle coordinate system, where the X-axis points forward from the vehicle, the Y-axis points to the left of the vehicle, and the Z-axis points up from the ground.
By default, the origin of the coordinate system is on the ground, directly below the camera center defined by the camera's focal point. The origin can be moved by using the SensorLocation property of the monoCamera object. Additionally, monoCamera provides 'imageToVehicle' and vehicleToImage methods for converting between image and vehicle coordinate systems.

Before processing the entire video, process a single video frame to illustrate the concepts involved in the design of a monocular camera sensor.
Start by creating a VideoReader object that opens a video file. To be memory efficient, VideoReader loads one video frame at a time.

Read an interesting frame that contains lane markers and a vehicle. Load an aggregate channel features (ACF) detector that is pretrained to detect the front and rear of vehicles. A detector like this can handle scenarios were issuing a collision warning is important. It is not sufficient, for example, for detecting a vehicle traveling across a road in front of the ego vehicle. Use the configureDetectorMonoCamera function to specialize the generic ACF detector to take into account the geometry of the typical automotive application. Bypassing in this camera configuration, this new detector searches only for vehicles along the road's surface, because there is no point searching for vehicles high above the vanishing point. This saves computational time and reduces the number of false positives. convert vehicle detections to vehicle coordinates. The computeVehicleLocations function calculates the location of a vehicle in vehicle coordinates given a bounding box returned by a detection algorithm in image coordinates. It returns the center location of the bottom of the bounding box in vehicle coordinates. Because monocular camera sensor is used and a simple homography, only distances along the surface of the road can be computed accurately. Computation of an arbitrary location in 3-D space requires the use of a stereo camera or another sensor capable of triangulation. Create a pretrained vehicle detector and configure it to detect vehicle bounding boxes using the calibrated monocular camera configuration. To detect vehicles, try out the algorithm on a single video frame and retrain it.

Figure 36 Monocular camera detecting vehicles and coordinates concerning camera as an origin

Now that vehicles have been detected (Figure 36), estimate distances to the detected vehicles from the camera in world c oordinates. monoCamera provides an imageToVehicle method to convert points from image coordinates to vehicle coordinates. This can be used to estimate the distance along the ground from the camera to the detected vehicles.

Incorporate the vehicle detection and distance estimation automation class into the automation workflow of the app. Start with the existing ACF Vehicle Detection automation algorithm to perform vehicle detection with a calibrated monocular

Figure 37 Vehicles detected by a camera and represented as bounding boxes

camera. Then modify the algorithm to perform attribute automation, use the distance of the vehicle from the camera as an attribute of the detected vehicle.

Figure 38 Vehicles detected by a camera with estimating distance

Steps to be followed:

The packaged version of the vehicle distance computation algorithm is available in the VehicleDetectionAndDistanceEstimation class. To use this class in the app:

Create the folder structure required under the current folder, and copy the automation class into it.

```
    mkdir('+vision/+labeler');
copyfile(fullfile(matlabroot,'examples','driving','main','VehicleDetectionAndDistanceEstimation.m'),'+
vision/+labeler');
```

Load the monoCamera information into the workspace. This camera sensor information is suitable for the camera used in the video used in this example, 05_highway_lanechange_25s.mp4. If you load a different video, use the sensor information appropriate for that video.

```
load('FCWDemoMonoCameraSensor.mat', 'sensor')
 Open the groundTruthLabeler app.
groundTruthLabeler 05_highway_lanechange_25s.mp4
```

1. In the ROI Label Definition pane on the left, click Label. Define a label with the name Vehicle and type Rectangle. Optionally, add a label description. Then click OK.
2. In the ROI Label Definition pane on the left, click Attribute. Define an attribute with name Distance, type Numeric Value, and default value 0. Optionally, add an attribute description. Then click OK.
3. Select Algorithm > Select Algorithm > Refresh list.
4. Select Algorithm > Vehicle Detection and Distance Estimation. If you do not see this option, ensure that the current working folder has a folder called +vision/+labeler, with a file named VehicleDetectionAndDistanceEstimation.m in it.
5. Click Automate. A new tab opens, displaying directions for using the algorithm.
6. Click Settings, and in the dialog box that opens, enter the sensor in the first text box. Modify other parameters if needed before clicking OK.
7. Click Run. The vehicle detection and distance computation algorithm progress through the video. Notice that the results are not satisfactory in some of the frames.
8. After the run is completed, use the slider or arrow keys to scroll across the video to locate the frames where the algorithm failed.
9. Manually tweak the results by either moving the vehicle bounding box or by changing the distance value. You can also delete the bounding boxes and the associated distance values.
10. Once you are satisfied with the vehicle bounding boxes and their distances for the entire video, click Accept.

Automated parking of a vehicle

Automatically parking a car that is left in front of a parking lot is a challenging problem. The vehicle's automated systems are expected to take over control and steer the vehicle to an available parking spot. Such a function makes use of multiple onboard sensors. For example:

Front and side cameras for detecting lane markings, road signs (stop signs, exit markings, etc.), other vehicles, and pedestrians

Lidar and ultrasound sensors for detecting obstacles and calculating accurate distance measurements

Ultrasound sensors for obstacle detection

IMU and wheel encoders for dead reckoning

On-board sensors are used to perceive the environment around the vehicle. The perceived environment includes an understanding of road markings to interpret road rules and infer drivable regions, recognition of obstacles, and the detection of available parking spots.

As the vehicle sensors perceive the world, the vehicle must plan a path through the environment towards a free parking spot and execute a sequence of control actions needed to drive to it. While doing so, it must respond to dynamic changes in the environment, such as pedestrians crossing its path, and readjust its plan.

The environment model represents a map of the environment. For a parking valet system, this map includes available and occupied parking spots, road markings, and obstacles such as pedestrians or other vehicles. Occupancy maps are a common representation of this form of environment model. Such a map is typically built using Simultaneous Localization and Mapping (SLAM) by integrating observations from lidar and camera sensors. This example concentrates on a simpler scenario, where a map is already provided, for example, by a vehicle-to-infrastructure (V2X) system or a camera overlooking the entire parking space. It uses a static map of a parking lot and assumes that the self-localization of the vehicle is accurate.

The parking lot example used in this example is composed of three occupancy grid layers.

Stationary obstacles: This layer contains stationary obstacles like walls, barriers, and bounds of the parking lot.

Road markings: This layer contains occupancy information about road markings, including road markings for parking spaces.

Parked cars: This layer contains information about which parking spots are already occupied.

Each map layer contains different kinds of obstacles that represent different levels of danger for a car navigating through it. With this structure, each layer can be handled, updated, and maintained independently.

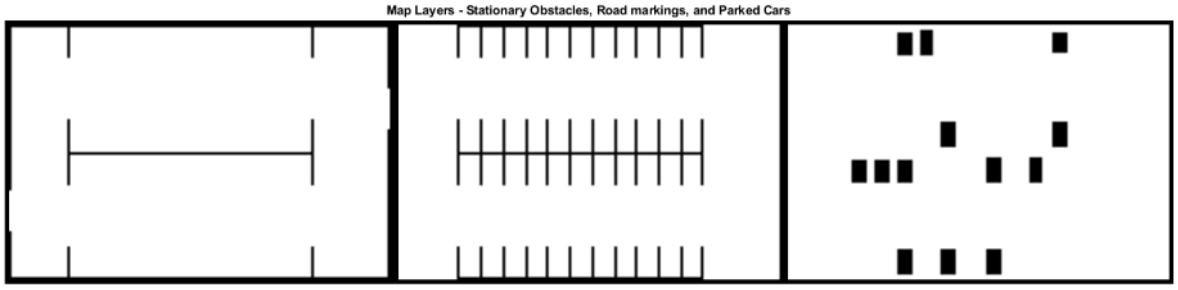

Figure 39 occupancy grid layers of a parking lot

The combined costmap is a vehicleCostmap object, which represents the vehicle environment as a 2-D occupancy grid. Each grid in the cell has values between 0 and 1, representing the cost of navigating through the cell. Obstacles have a higher cost, while free space has a lower cost. A cell is considered an obstacle if its cost is higher than the OccupiedThreshold property and free if its cost is lower than the

FreeThreshold property. The costmap covers the entire 75m-by-50m parking lot area, divided into 0.5m-by-0.5m square cells.

Create a vehicleDimensions object for storing the dimensions of the vehicle that will park automatically. Also, define the maximum steering angle of the vehicle. This value determines the limits on the turning radius during motion planning and control. Create a vehicleDimensions object for storing the dimensions of the vehicle that will park automatically. Also, define the maximum steering angle of the vehicle. This value determines the limits on the turning radius during motion planning and control. Planning involves organizing all pertinent information into hierarchical layers. Each successive layer is responsible for a more fine-grained task. The behavioral layer [1] sits at the top of this stack. It is responsible for activating and managing the different parts of the mission by supplying a sequence of navigation tasks. The behavioral layer assembles information from all relevant parts of the system, including:

1. Localization: The behavioral layer inspects the localization module for an estimate of the current location of the vehicle.
2. Environment model: Perception and sensor fusion systems report a map of the environment around the vehicle.
3. Determining a parking spot: The behavioral layer analyzes the map to determine the closest available parking spot.
4. Finding a global route: A routing module calculates a global route through the road network obtained either from a mapping service or from a V2X infrastructure.

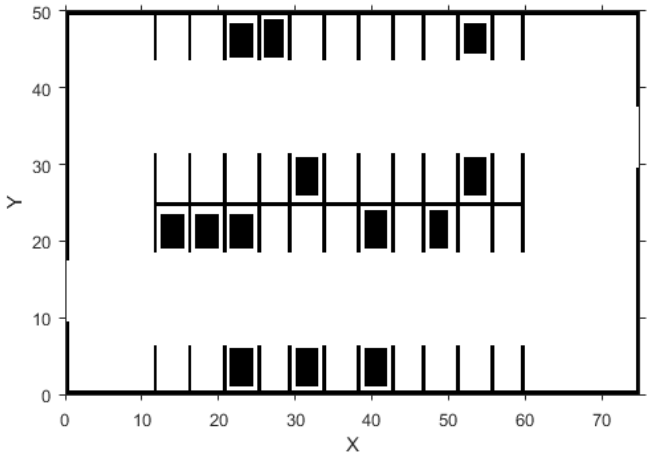

Figure 40 combined costmap

Decomposing the global route as a series of road links allows the trajectory for each link to be planned differently. For example, the final parking maneuver requires a different speed profile than the approach to the parking spot. In a more general setting, this becomes crucial for navigating through streets that involve different speed limits, numbers of lanes, and road signs. Rather than rely upon on-vehicle sensors to build a map of the environment, this example uses a map that comes from a smart parking lot via V2X communication. For simplicity, assume that the map is in the form of an occupancy grid, with road links and locations of available parking spots provided by V2X. The HelperBehavioralPlanner class mimics an interface of a behavioral planning layer. The HelperBehavioralPlanner is created using the map and the global route plan. The MAT-file containing a route plan that is stored in a table. The table has three variables: StartPose, EndPose, and Attributes. StartPose and EndPose specify the start and end poses of the segment, expressed as $[x,y,\theta]$. Attributes specify properties of the segment such as the speed limit. Plot a vehicle at the current pose, and along with each goal in the route plan. Create the behavioral planner helper object. The 'requestManeuver' method requests a stream of navigation tasks from the behavioral planner until the destination is reached. The vehicle navigates each path segment using these steps:

1. Motion Planning: Plan a feasible path through the environment map using the optimal rapidly exploring random tree (RRT*) algorithm (pathPlannerRRT).
2. Path Smoothing: Smooth the reference path by fitting splines to it using smoothPathSpline.
3. Trajectory Generation: Convert the smoothed path into a trajectory by generating a speed profile using helperGenerateVelocityProfile.

4. Vehicle Control: Given the smoothed reference path, HelperPathAnalyzer calculates the reference pose and velocity based on the current pose and velocity of the vehicle. Provided with the reference values, lateralControllerStanley computes the steering angle to control the heading of the vehicle. HelperLongitudinalController computes the acceleration and deceleration commands to maintain the desired vehicle velocity.
5. Goal Checking: Check if the vehicle has reached the final pose of the segment using helperGoalChecker.

Given a global route, motion planning can be used to plan a path through the environment to reach each intermediate waypoint, until the vehicle reaches the final destination. The planned path for each link must be feasible and collision-free. A feasible path is one that can be realized by the vehicle given the motion and dynamic constraints imposed on it. A parking valet system involves low velocities and low accelerations. This allows us to safely ignore dynamic constraints arising from inertial effects.

Create a pathPlannerRRT object to configure a path planner using an optimal rapidly exploring random tree (RRT*) approach. The RRT family of planning algorithms find a path by constructing a tree of connected, collision-free vehicle poses. Poses are connected using Dubins or Reeds-Shepp steering, ensuring that the generated path is kinematically feasible. Plan a path from the current pose to the first goal by using the plan function. The returned driving. Path object, refPath, is a feasible and collision-free reference path. The reference path consists of a sequence of path segments. Each path segment describes the set of Dubins or Reeds-Shepp maneuvers used to connect to the next segment. Inspect the path segments. Use smoothPathSpline to fit a parametric cubic spline that passes through all the transition points in the reference path. The spline approximately matches the starting and ending directions with the starting and ending heading angle of the vehicle. Specify initial, maximum, and terminal speeds so that the vehicle starts stationary, accelerates to a speed of 5 meters/second, and comes to a stop. The reference speeds, together with the smoothed path, comprise a feasible trajectory that the vehicle can follow. A feedback controller is used to follow this trajectory. The controller corrects errors in tracking the trajectory that arises from tire slippage and other sources of noise, such as inaccuracies in localization. In particular, the controller consists of two components:

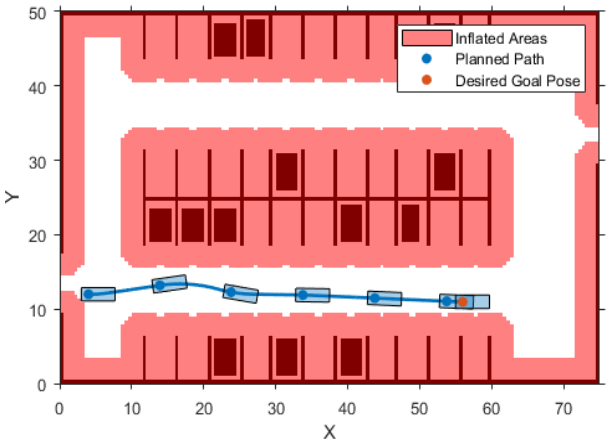

Figure 41 Transition poses and directions from the planned path

1. Lateral control: Adjust the steering angle such that the vehicle follows the reference path.
2. Longitudinal control: While following the reference path, maintain the desired speed by controlling the throttle and the brake.

Since this scenario involves slow speeds, you can simplify the controller to take into account only a kinematic model. In this example, lateral control is realized by the lateralControllerStanley function. The

longitudinal control is realized by a helper System object™ HelperLongitudinalController, that computes acceleration and deceleration commands based on the Proportional-Integral law.
The feedback controller requires a simulator that can execute the desired controller commands using a suitable vehicle model. The HelperVehicleSimulator class simulates such a vehicle using the following kinematic bicycle model:

$$\dot{x}_r = v_r * \cos(\theta) \quad \dot{y}_r = v_r * \sin(\theta) \quad \dot{\theta} = \frac{v_r}{l} * \tan(\delta) \qquad \dot{v}_r = a_r$$

In the above equations, (x_r, y_r, θ) represents the vehicle pose in world coordinates $v_r a_r l$, and δ represent the rear-wheel speed, rear-wheel acceleration, wheelbase, and steering angle, respectively. The position and speed of the front wheel can be obtained by:

$$x_f = x_r + l\cos(\theta) \qquad y_f = y_r + l\sin(\theta) \quad v_f = \frac{v_r}{\cos(\delta)}$$

Until the goal is reached, do the following:

1. Compute steering and acceleration/deceleration commands required to track the planned trajectory.
2. Feed control commands to the simulator.
3. Record the returned vehicle pose and velocity to feed into the controller in the next iteration.
4. Now combine all the previous steps in the planning process and simulate the complete route plan. This process involves incorporating the behavioral planner.

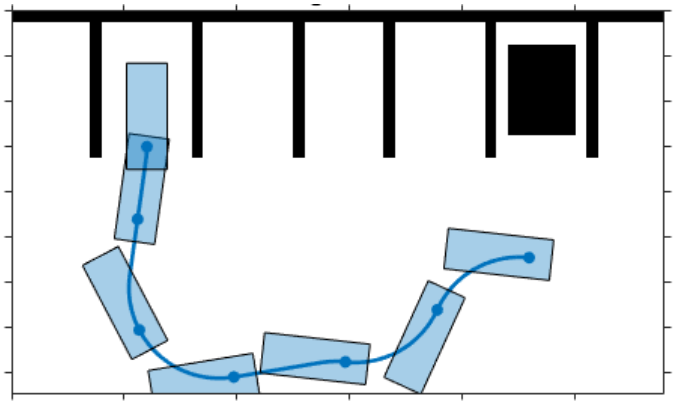

Figure 42 Planning maneuver

An alternative way to park the vehicle is to back into the parking spot. When the vehicle needs to back up into a spot, the motion planner needs to use the Reeds-Shepp connection method to search for a feasible path. The Reeds-Shepp connection allows for reverse motions during planning.

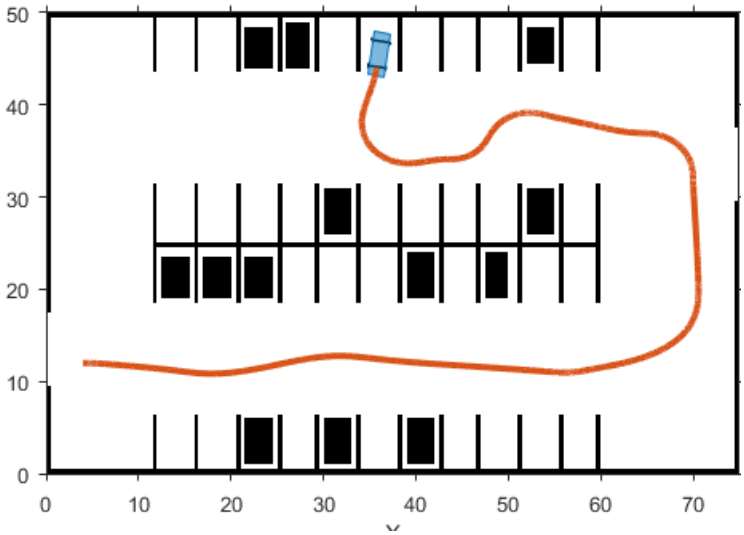

Figure 43 Actual Path Traced bt Ego-Vehicle

Conclusions

Autonomous systems require sensing, perception, planning, and execution. While sensing is usually done by readily available sensors, and execution is controlled by well-known and mature control algorithms, the stages of perception and planning pose the core of research and development efforts for autonomous systems.

This project has introduced the terminology, design considerations, and safety assessment of self-driving cars. Below mentioned are the learnings of this project:

1. Understand commonly used hardware used for self-driving cars
2. Identify the main components of the self-driving software stack
3. Program vehicle modeling and control
4. Analyze the safety frameworks and current industry practices for vehicle development
5. Understand the key methods for parameter and state estimation used for autonomous driving, such as the method of least-squares
6. Develop a model for typical vehicle localization sensors, including GPS and IMUs
7. Apply extended and unscented Kalman Filters to a vehicle state estimation problem
8. Understand LIDAR scan matching and the Iterative Closest Point algorithm

Apply these tools to fuse multiple sensor streams into a single state estimate for a self-driving car and could learn the main perception tasks in autonomous driving, static and dynamic object detection, and will survey common computer vision methods for robotic perception. Methods to visual odometry, object detection, and tracking, and semantic segmentation for drivable surface estimation. These techniques represent the main building blocks of the perception system for self-driving cars. find the shortest path over a graph or road network using Dijkstra's and the A* algorithm, use finite state machines to select safe behaviors to execute, and design optimal, smooth paths and velocity profiles to navigate safely around obstacles while obeying traffic laws. Build occupancy grid maps of static elements in the environment and learn how to use them for efficient collision checking. The ability to construct a full self-driving planning solution, to take you from home to work while behaving like a typical driving and keeping the vehicle safe at all times.

Bibliography

1. IEEE. Standard for Distributed Interactive Simulation – Application Protocols. IEEE P1278.1/D16 Rev 18, May 2012.
2. Falcone, P. et al., "Predictive Active Steering Control for Autonomous Vehicle Systems", IEEE (2007).
3. Buehler, Martin, Karl Iagnemma, and Sanjiv Singh. The DARPA Urban Challenge: Autonomous Vehicles in City Traffic (1st ed.). Springer Publishing Company, Incorporated, 2009.
4. Lepetic, Marko, Gregor Klancar, Igor Skrjanc, Drago Matko, Bostjan Potocnik, "Time Optimal Path Planning Considering Acceleration Limits." Robotics and Autonomous Systems. Volume 45, Issues 3-4, 2003, pp. 199-210.
5. J. Jiang and A. Astolfi, "Lateral Control of an Autonomous Vehicle," in IEEE Transactions on Intelligent Vehicles, vol. 3, no. 2, pp. 228-237, June 2018.
6. Wang, H., Kearney, J., & Atkinson, K. (2002, June). Robust and efficient computation of the closest point on a spline curve. In Proceedings of the 5th International Conference on Curves and Surfaces (pp. 397-406).
7. Snider, J. M., "Automatic Steering Methods for Autonomous Automobile Path Tracking", Robotics Institute, Carnegie Mellon University, Pittsburg (February 2009).
8. Hoffmann, G. et al., "Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing", Stanford University, (2007).
9. Roland Siegwart, Illah R. Nourbakhsh, Davide Scaramuzza, Introduction to Autonomous Mobile Robots (2nd ed., 2011).
10. Timothy D. Barfoot, State Estimation for Robotics (2017)