2021

DEVELOPMENT OF AUTONOMOUS VEHICLE MOTION PLANNING AND CONTROL ALGORITHM WITH D* PLANNER AND MODEL PREDICTIVE CONTROL IN A DYNAMIC ENVIRONMENT

Somnath Mondal
Michigan Technological University, somnathm@mtu.edu

DEVELOPMENT OF AUTONOMOUS VEHICLE MOTION PLANNING AND CONTROL ALGORITHM WITH D* PLANNER AND MODEL PREDICTIVE CONTROL IN A DYNAMIC ENVIRONMENT

By

Somnath Mondal

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Mechanical Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2021

This report has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Mechanical Engineering.

Department of Mechanical Engineering-Engineering Mechanics

Report Advisor: *Dr. Bo Chen*

Committee Member: *Dr. Jung Yun Bae*

Committee Member: *Dr. Ye Sun*

Department Chair: *Dr. William W. Predebon*

Contents

Table of Figures

Acknowledgements

I would like to express my utmost gratitude to my advisor Dr. Bo Chen for her continuous support and guidance throughout this research work and completing my Master's degree. I would also like to thank my committee members Dr. Jungyun Bae and Dr. Ye Sun for their time in reviewing my work and their valuable suggestions.

I am also thankful to all my colleagues at Intelligent Mechatronics and Embedded System Lab at Michigan Technological University for their time and help in guiding me through choosing my research topic.

I am grateful to my family and friends for their constant support and encouragement in my toughest times. This would have not been possible without them.

Abstract

The research in this report incorporates the improvement in the autonomous driving capability of self-driving cars in a dynamic environment. Global and local path planning are implemented using the D Star (Dynamic A Star) path planning algorithm with a combined Cubic B-Spline trajectory generator, which generates an optimal obstacle free trajectory for the vehicle to follow and avoid collision. Model Predictive Control (MPC) is used for the longitudinal and the lateral control of the vehicle. The presented motion planning and control algorithm is tested using Model-In-the-Loop (MIL) method with the help of MATLAB® Driving Scenario Designer and Unreal Engine® Simulator by Epic Games®. Different traffic scenarios are built, and a camera sensor is configured to simulate the sensory data and feed it to the controller for further processing and vehicle motion planning. Simulation results of vehicle motion control with global and local path planning for dynamic obstacle avoidance are presented. The simulation results show that an autonomous vehicle (ego vehicle) follows a commanded velocity when the relative distance between the ego vehicle and an obstacle is greater than a calculated safe distance. When the relative distance is close to the safe distance, the ego vehicle maintains the headway. When an obstacle is detected by the ego vehicle and the ego vehicle wants to pass the obstacle, the ego vehicle performs obstacle avoidance maneuver by tracking desired lateral positions.

1. Introduction

Self-driving is currently one of the main research areas of automotive industry and academia because of its advantages including greater safety, reduced emissions, and improved mobility [1]. Recent advancements in both the sensing and computing technologies have fueled the developments in driverless vehicle technology as a huge societal benefit has been perceived [2].

National Highway Traffic Safety Administration (NHTSA) of the United States Federal Government has recently published collected results from National Motor Vehicle Crash Causation Survey to determine the critical reason for the pre-crash events from the on-scene post-crash data. It concludes that most of the critical reasons for crashes are because of the drivers [3].

Critical reasons leading up to crashes, July 3, 2005 to December 31, 2007

Critical reasons	Number of crashes	Percent of crashes
Drivers	2,046,000	94%
Vehicles	44,000	2%
Environment	52,000	2%
Unknown Critical Reasons	47,000	2%
Total	2,189,000	100%

Source: Singh, S. (2018, March). Critical reasons for crashes investigated in the National Motor Vehicle Crash Causation Survey. (Traffic Safety Facts Crash Stats. Report No. DOT HS 812 506). Washington, DC: National Highway Traffic Safety Administration.

Figure 1-1: Critical reasons leading to crashes in the United States [3]

Again, the crashes in the survey above have been split into four different driver error categories, where it was found that 41% of the total crashes were caused by the recognition error and 33% were caused by decision error. The categories of the errors are listed as recognition error, decision error, performance error and non-performance error. Recognition error includes drivers'

inattention, distractions (both internal and external), and inadequacy of surveillance. Decision error includes illegal maneuvers, misjudgment of speed and distance of other vehicles. Performance error includes poor control maneuvers for both lateral and longitudinal control. And non-performance error involves the instances when drivers fall asleep [3].

Critical driver-related reasons leading up to crashes, July 3, 2005 to December 31, 2007

Critical reasons	Number of crashes	Percent of crashes
Recognition error	845,000	41%
Decision error	684,000	33%
Performance error	210,000	11%
Non-performance error	145,000	7%
other	162,000	8%
Total	2,046,000	100%

Source: Singh, S. (2018, March). Critical reasons for crashes investigated in the National Motor Vehicle Crash Causation Survey. (Traffic Safety Facts Crash Stats. Report No. DOT HS 812 506). Washington, DC: National Highway Traffic Safety Administration.

Figure 1-2: Driver related errors that lead to crashing in the U.S. [3]

The Fatality Analysis Reporting System (FARS) has shared the NHTSA data regarding fatal crashes where it was indicated that the speeding was a main factor in a fatal crash. The analysis by National Safety Council (NSC) has been shown in Figure 1-3.

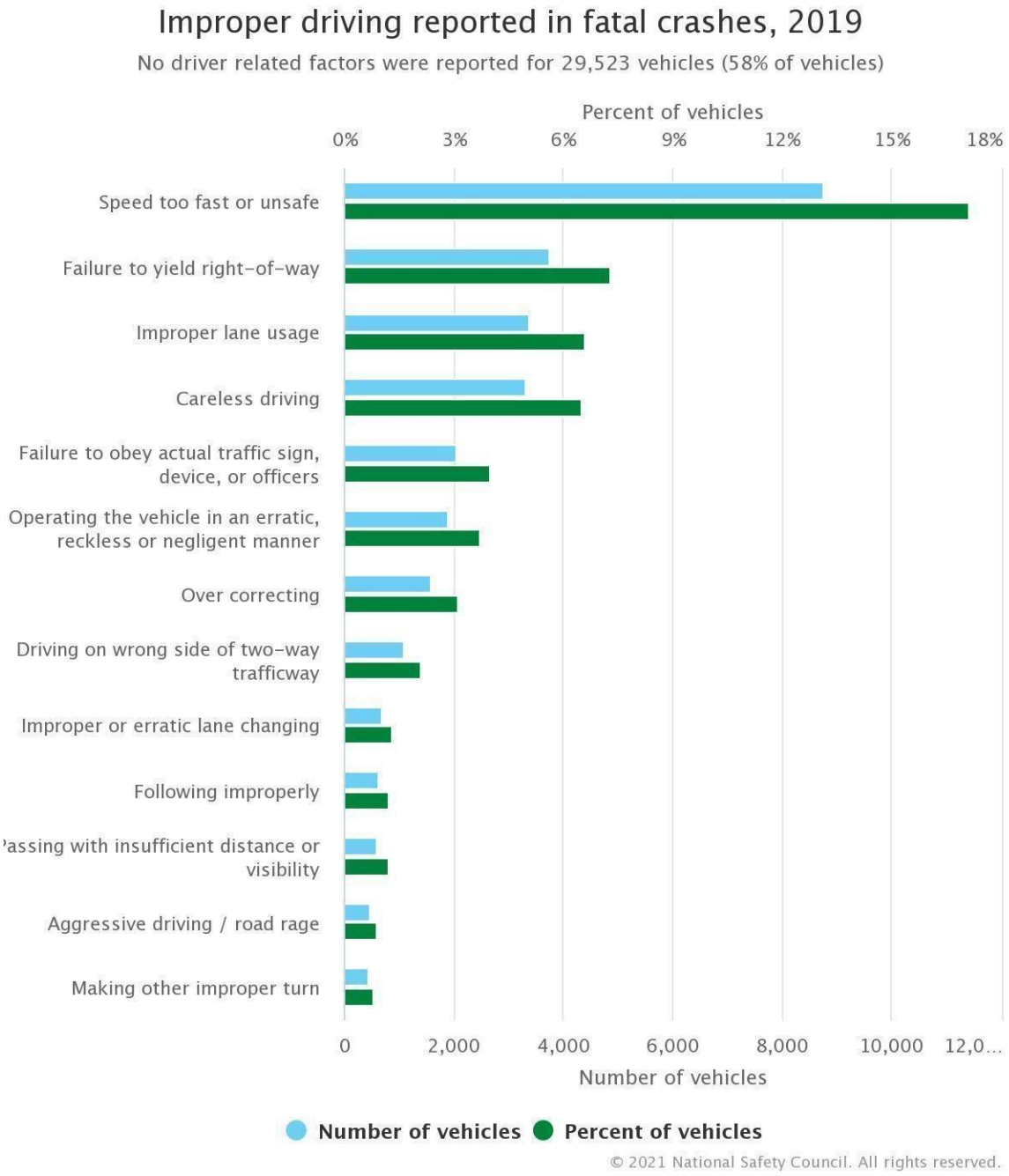

Figure 1-3: Report of Improper Driving Involved in Fatal Crashes in 2019 in the U.S. [3]

This is a clear take away that if human driving is supplanted with a much more sophisticated computation methodologies of autonomous vehicles, then the number of vehicle collisions caused by driver error and negligence will drastically reduce [2]. The key to autonomy is critical decision

making and achieved through planning algorithms while considering vehicle dynamics and its maneuver capabilities in different driving scenarios [1].

Most of the new cars in the United States are equipped with such technology called Advanced Driver Assistance Systems (ADAS) [4]. An extension to this ADAS is fully automated driving capabilities of vehicles i.e., vehicles can drive themselves without human intervention [5]. With rapidly evolving sensor and on-board computational systems to meet the demands of expanded self-driving vehicle operations, most global auto manufacturers including General Motors, Ford, Volkswagen, Toyota, Honda, Tesla, Volvo, and BMW are exploring to operate the vehicles at five increasingly sophisticated autonomy levels (as defined by SAE International [6]) [4]. The DARPA 2007 Urban Challenge winner, Boss, a full-size autonomous vehicle, demonstrated its capability of driving with simulated moving traffic in an urban environment with a speed up to 30 miles per hour (mph). Hence, developing a mechanism to survive the rigors of the traffic was not the challenge, instead, the challenge was in developing the decision-making algorithms and software that could robustly take care of driving, even in unexpected circumstances [7].

1.1 An Overview of the Decision-Making Hierarchy:

The decision-making hierarchy in autonomous driving systems of level 3 and above are constructed into route planning, behavioral decision making, local motion planning and feedback control as shown in Figure 1-4. In this section, the different layers of the decision-making hierarchy are described, and the responsibilities of each layer are discussed.

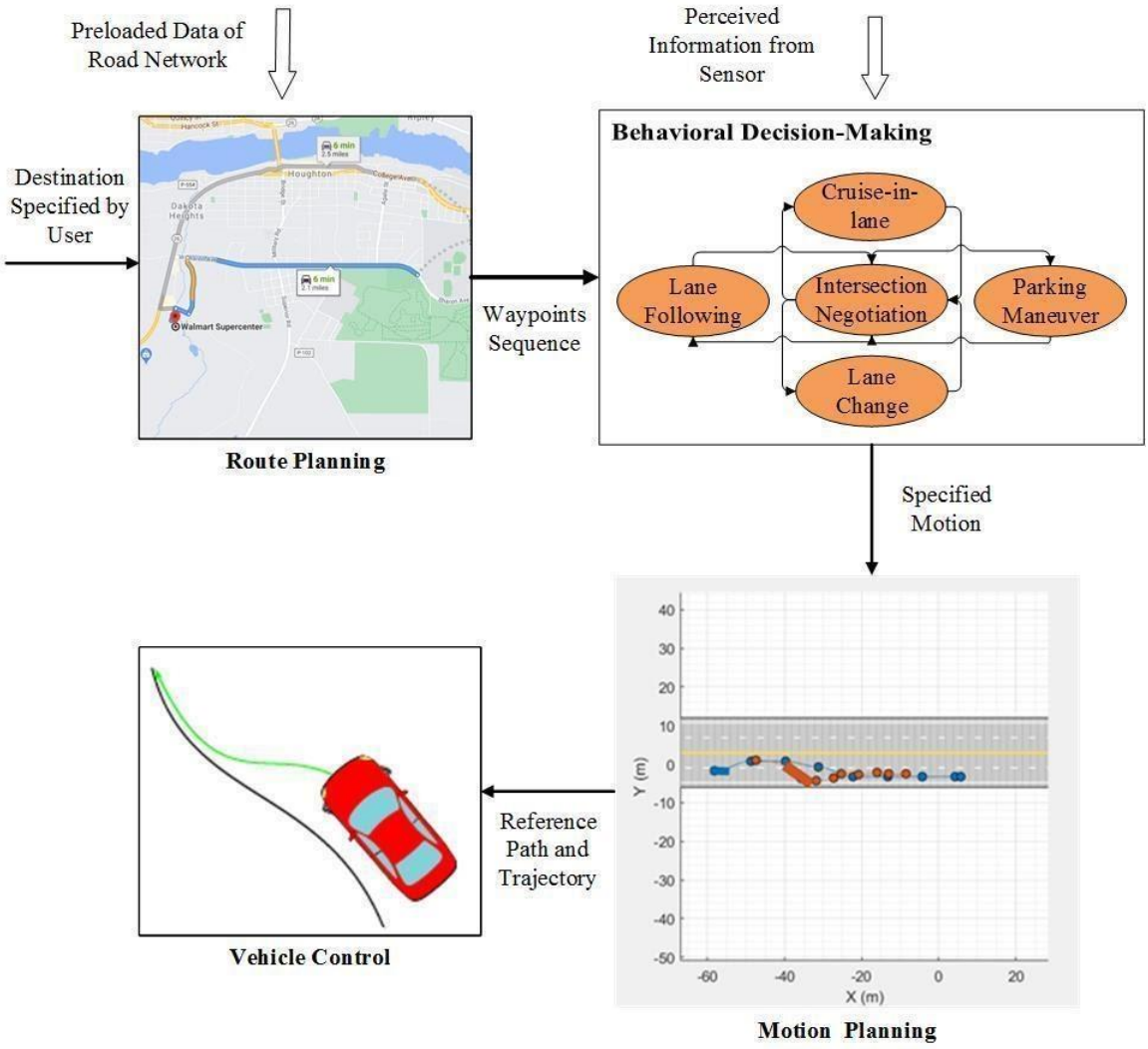

Figure 1-4: Decision-Making Hierarchy in Autonomous Vehicles

The autonomous decision-making system in the self-driving cars uses on-board sensors like cameras, radars, LiDARs, Global Positioning System (GPS), Inertial Measurement Unit (IMU), odometry to gather and process the observations about the vehicle and the surroundings. Then these observations along with previous information about road networks, road rules, sensor model and vehicle dynamics are used to derive control parameters for a desired motion of the vehicle. Advancing toward autonomy requires the vehicle system to hierarchically organize the perception and decision-making tasks. The perception system uses the previous information and observed

data collected from the sensors to provide an estimation of the vehicle state and its surroundings. The decision-making system then uses these estimates to control the vehicle for driving objective accomplishment.

1.1.1 Route Planning

The highest level of decision-making requires the vehicle to select a path through road networks from the current position of the vehicle to the desired destination. A path is a geometric trace that a vehicle needs to follow for successfully reaching the goal position avoiding collision with any obstacle. Route planning or path planning is therefore a problem of determining a geometric path from an initial set of states to a given terminating set of states and configuration where each state and configuration on the path is feasible. Feasible states and configurations do not result in a collision and stick to motion constraints like lane and road boundaries, traffic rules etc. It can also be formulated as finding a path with minimum cost of traversing on a road network by representing the road network as a weight-based graph where weight corresponds to the cost of traversing.

The graphs used to represent the road networks are sometimes consist of million edges which makes the classical path-search algorithms like Dijkstra [8] or A* [9] to be impractical. Efficient route planning depends on algorithms which return an optimal route in milliseconds [10], [11] after one-time pre-processing.

1.1.2 Behavioral Decision Making

Once a route has been determined, the self-driving vehicle must navigate through the route and interact with the other traffic participants while obeying all the driving rules and conventions of the road. Considering other traffic-participants' perceived behavior, road condition and signals received from road-side infrastructures, the decision-making layer selects an appropriate driving

behavior at any instant [2]. Some of the decision-making examples are ‘turning’, ‘going straight’, ‘overtaking’, following a particular lane, changing a lane, cruise control in a lane, negotiating a signalized intersection, parking in a valet etc. In short, behavioral decision-making helps the vehicle take the best high-level decision while accounting the path specified from route planning.

Driving in real-world especially in urban scenarios is specified by the uncertainty over the unpredicted intentions of the other traffic members. Among the solutions to address the problem of predicting intentions and estimating the future trajectories of other vehicles, pedestrians and bikes, there are techniques which are based on machine learning, such as Gaussian mixture models [12], Gaussian process regression [13], the learning techniques for predicting intentions used by Google self-driving cars [14], and model-based approaches to estimate intentions directly from sensor information [15], [16].

1.1.3 Motion Planning

Once the behavioral decision-making layer determines the driving behavior for the instance, the desired maneuver must be converted to a path or trajectory that the low-level feedback controller can track. The converted path or trajectory has to be feasible dynamically for the vehicle, comfortable as well for the passengers and should avoid colliding with other sensor-detected obstacles. The motion planning system is responsible for finding such a path or trajectory for the vehicle from its current configuration to the termination or goal configuration specified by the behavioral layer.

The motion planning layer collects information about both static and dynamic obstacles in the vehicle surroundings, and then generates an obstacle-free trajectory which satisfies both kinematic

and dynamic vehicle motion constraints. The problem of motion planning can be divided into global [17] and local planning [18]. To address the challenges in driving rules and broad road networks, the planning techniques are classified into four sections according to their utilization in autonomous driving, namely graph search method, sampling-based method, interpolating method, and numerical optimization-based method [19].

In graph search-based planners, Dijkstra's algorithm finds the shortest path in the graph with a single source. In self-driving applications, it was implemented by the Ben Franklin racing team in DARPA Urban Challenge [20], and the VictorTango team [21]. A* algorithm implements a heuristic based fast node search method with the determination of a cost function defining the weights of the nodes [22]. Some improved applications of A* are the dynamic A* (D*) [23], Anytime D* (AD*) [24], Field D* [25]. In DARPA Urban Challenge, Stanford University team Junior implemented hybrid A* [26], and KIT team AnnieWAY implemented A* [27]. The DARPA Urban Challenge winning team of Carnegie Mellon University implemented AD* in their vehicle Boss [28].

In sampling-based planners, algorithms provide suboptimal solutions by planning in high-dimensional spaces and solving time constraints [29]. The Probabilistic Roadmap Method (PRM) and the Rapidly-exploring Random Tree (RRT) are the most commonly used methods, the RRT has been tested extensively in self-driving vehicles. The vehicle developed by MIT at DARPA Urban Challenge implemented a variation of RRT which is called closed-loop RRT with biased sampling [30].

1.1.4 Vehicle Control

A control system is necessary in autonomous navigation for intelligent vehicles. This control system consists of a longitudinal controller and lateral controller. The longitudinal controller regulates the vehicle cruise velocity and maintains a safe distance from the lead vehicle using the throttle and brake command, whereas the lateral controller controls the steering angle of the vehicle for path tracking maneuver [31]. In article [32], a detailed survey on longitudinal control for car following is provided, where vehicle longitudinal control in car following with vision system and fuzzy logic [33] was discussed among many. Article [34] shows the implementation of Model Predictive Control for vehicle longitudinal control in Adaptive Cruise Control (ACC). A broad survey on vehicle lateral control is provided in [2]. Second place holder vehicle Sandstorm in DARPA Grand Challenge used a modified pure-pursuit controller for path tracking [35]. DARPA Grand Challenge winner Stanley, and second place winner Junior in DARPA Urban Challenge both used the vehicle's kinematic model-based steering control [36]. And DARPA Urban Challenge winner Boss used a model predictive control strategy for vehicle lateral control [37].

1.2 Research Objective and Contributions

The center of many problems related to path planning of autonomous vehicles is to find the path with lowest cost of traversing through a graph of states. When the arc costs change during traverse, rapid replanning of the remainder of the path is essential. Therefore, the objective of this research is to develop a motion planning algorithm integrated with a closed-loop controller which is capable of real-time optimal traverse planning, also computationally time and cost efficient.

In this research, a novel motion planning algorithm has been implemented in a dynamic environment using D* path planner and Cubic B Spline trajectory generator. The start and the goal position were determined from the scenario canvas built in simulation. The start position of the vehicle is the current position of the ego vehicle in the scenario. The obstacle information such as distance, orientation, velocity etc. and other scenario information such as road boundaries, lane boundaries etc. were obtained using a simulated monocular camera sensor available in MATALB Driving Scenario Designer. Cubic B-Spline trajectory generator is used to smoothen the planned path for non-holonomic constraints of the vehicle. The path planner combined with trajectory generator generates an optimal obstacle free path and provides the desired longitudinal and lateral position for the vehicle to follow. Afterwards, a novel longitudinal control model was developed and coupled with a lateral control model to use as prediction model in Model Predictive Controller for controlling the ego vehicle velocity, headway, and path tracking maneuver.

The developed motion planning and control algorithm is computationally efficient. The algorithm is tested and validated with MATLAB Driving Scenario Designer using Model-in-the-Loop (MIL) method.

The rest of the report is organized as follows. Section 2 discusses about motion planning and trajectory generation by the autonomous vehicle with the help of a camera sensor. Section 3 presents the development of a prediction model and a model predictive controller for lateral and longitudinal control of the vehicle. Section 4 discusses simulation results, and section 5 concludes the work.

2. Motion Planning: Optimal Path and Trajectory Generation

Path planning of autonomous vehicles requires two distinct but inherent aspects of planning: global path planning and local path planning. Global path planning plans the path from a specified starting point to a specified goal point. The global planning takes a map of the environment and plans the path considering the static obstacles already presented in the environment. Now, if a new obstacle appears in the environment and comes in the planned path of the vehicle, then the vehicle needs to locally plan a path from its current position to the goal and generate a trajectory that avoids the obstacle and maintain the nonholonomic constraints of the vehicle.

In this chapter, the implementation of D* path planning algorithm and trajectory generation using Cubic B-Spline trajectory generator is discussed.

2.1 Binary Occupancy Map

Binary occupancy map is a two-dimensional (2-D) occupancy map object in MATLAB, which is used to represent and visualize the surroundings of an autonomous vehicle, including obstacles. The approximate locations of the obstacles in the surroundings are spatially represented using the integrated sensor data and position estimates. Binary occupancy maps are useful in path planning algorithms and are also used in finding optimal obstacle-free path and collision avoidance. The occupancy status in the occupancy map is represented by a binary value zero or one. Zero (0) or ‘false’ represents a free location, whereas One (1) or ‘true’ represents an occupied location [38]. A thorough description on creation of the scenario canvas and gird map is given in later chapters.

2.2 Extracting Obstacle Information from Simulated Camera Sensor

The obstacles in the environment can be detected by various sensors such as lidar, radar, camera, and ultrasonic sensors. Radar can provide the information about any present obstacle in the environment using radio wave emitting from it and a map of the environment can be generated using that information. Lidar, similar to the radar, uses laser pulse instead of radio wave and gives the information about the environment as a three-dimension (3D) point cloud format and that is used to generate a map of the environment. A camera helps to detect an object presented in the environment and it provides with the obstacle information such as obstacle position, relative distance from the ego vehicle, relative velocity with respect to the ego vehicle, road boundaries, lane boundaries, and road signals. All this information is required to generate a map of the environment and to plan the motion of the ego vehicle accordingly.

In this project, a monocular camera sensor is used to sense the environment and extract the information of the environment. The camera is placed on the top of the ego vehicle as shown in Figure 2-1. The camera properties are given in Figure 2-2.

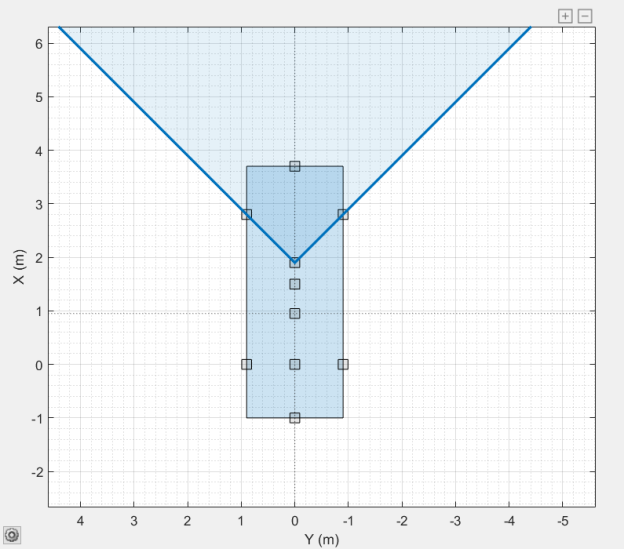

Figure 2-1: Camera sensor mounted on ego vehicle in the scenario

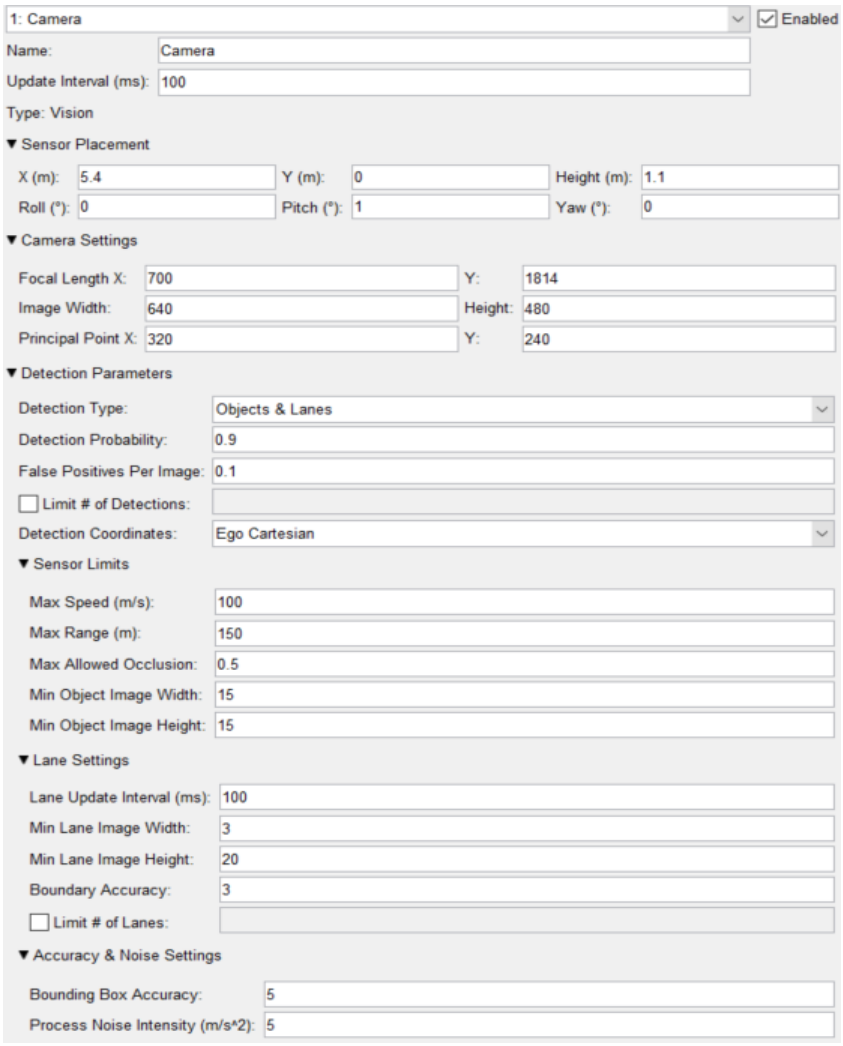

Figure 2-2: Monocular Camera Properties and Configurations

Increasing the focal length of the camera increases the width of the coverage area and therefore more obstacles around the ego vehicle come into the observation range. Update interval is set to 100ms which indicates the refreshing rate of the camera image. Detection probability in the configuration dialogue box is 0.9 which specifies the probability of the camera detecting a detectable object, the value ranges on the interval (0,1]. And, False Positive Per Image specifies the number of false detections generated per image by the camera sensor. Here, the number chosen is 1 per 10 images. This camera detects objects and lanes and sends it to the control system. A MATLAB function visionDetectionGenerator provides a statistical model for simulating

detections with the help of a monocular camera sensor. The velocity measurements are generated by the sensor with the help of a constant velocity Kalman filter [48].

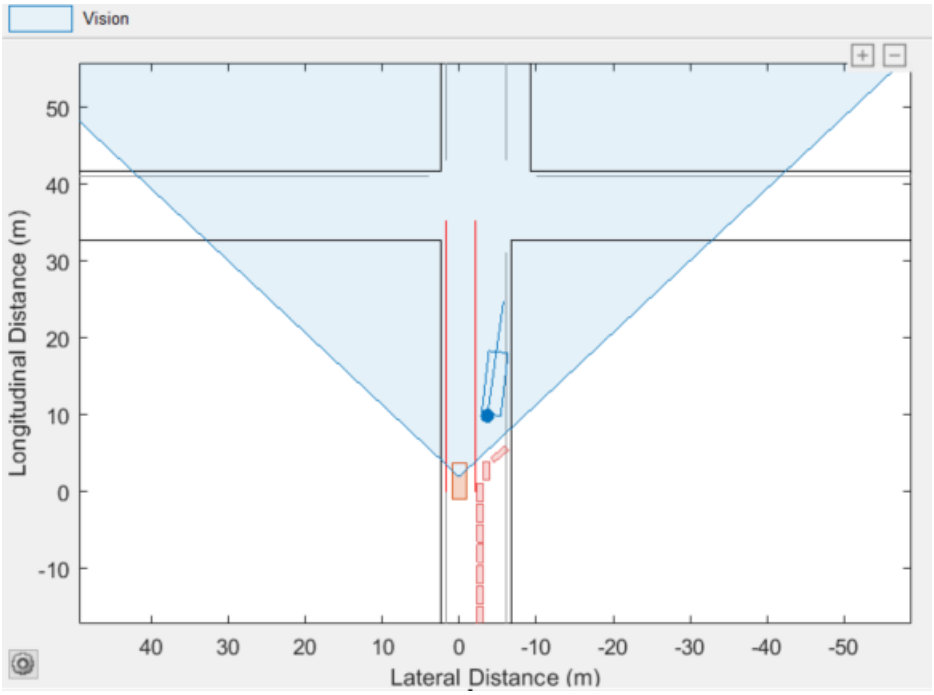

Figure 2-3: Obstacle Information Perceived by Camera Sensor

2.3 Generating Dynamic Map from Environment

Binary occupancy map is an ego centric map where the map is represented with two numbers: zero and one (0 and 1). The obstacle free space in the map is represented as zero and the space occupied by any obstacle is represented as 1. The cost of traversing through the map is then determined by the planner as it takes the map as an input to the planning architecture.

The coordinates of the scenario canvas designed in the MATLAB Driving Scenario Designer are the world coordinates (XY coordinates as shown in Figure 2-4), and this scenario canvas has to be

converted to an occupancy map so that the planner can use that to plan an optimal obstacle free path through the environment.

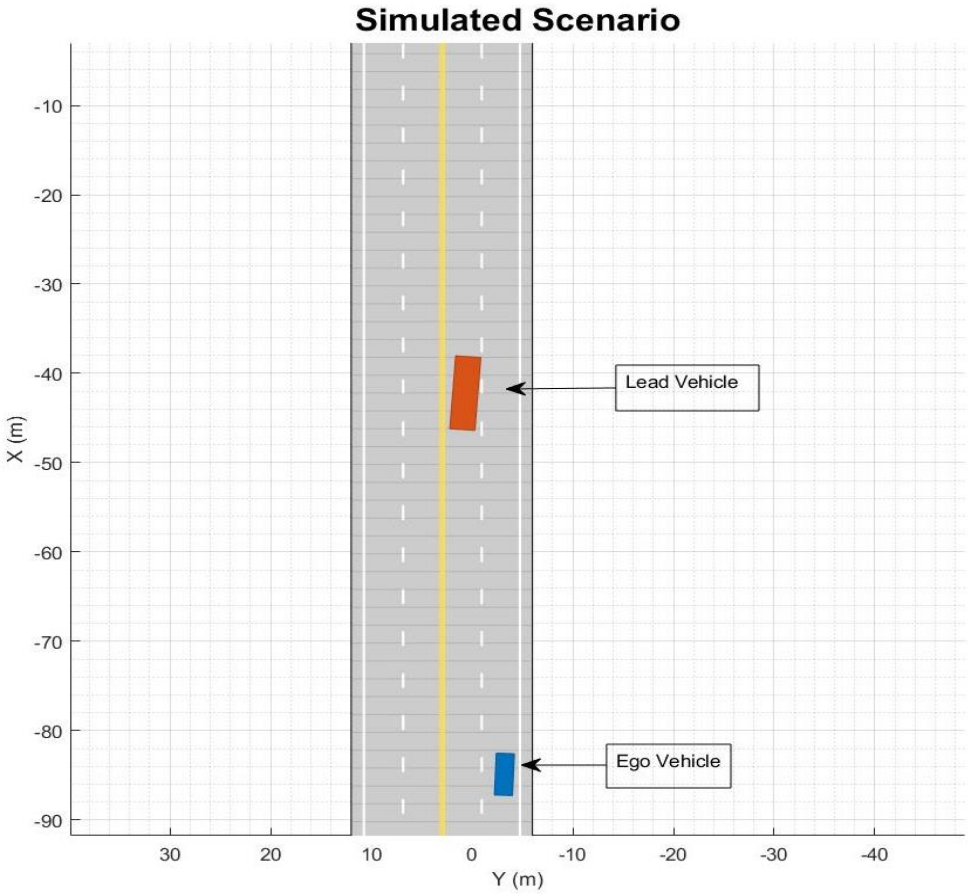

Figure 2-4: Scenario canvas representing world (XY) coordinates

As the project incorporates an environment where it includes dynamic obstacles, the occupancy map generated from the scenario canvas should be dynamically updated with the information about the changing environment. For that, we need to generate and update the occupancy map in each time step of the simulation when the vehicle is on the go. This way the binary occupancy map gets

updated with all the obstacle information that are coming on the way while the vehicle is moving and sets its occupancy accordingly.

The binary occupancy map (shown in Figure 2-5) is a 200X200 sized map which represents the scenario canvas in Figure 2-4, but only limited to the height and width defined during conversion (200x200) and keeping the ego vehicle position in the middle. Meaning that the occupancy map will only cover 200 meters multiply 200 meters of the actual scenario canvas when the ego vehicle position will be in the middle of the occupancy grid map.

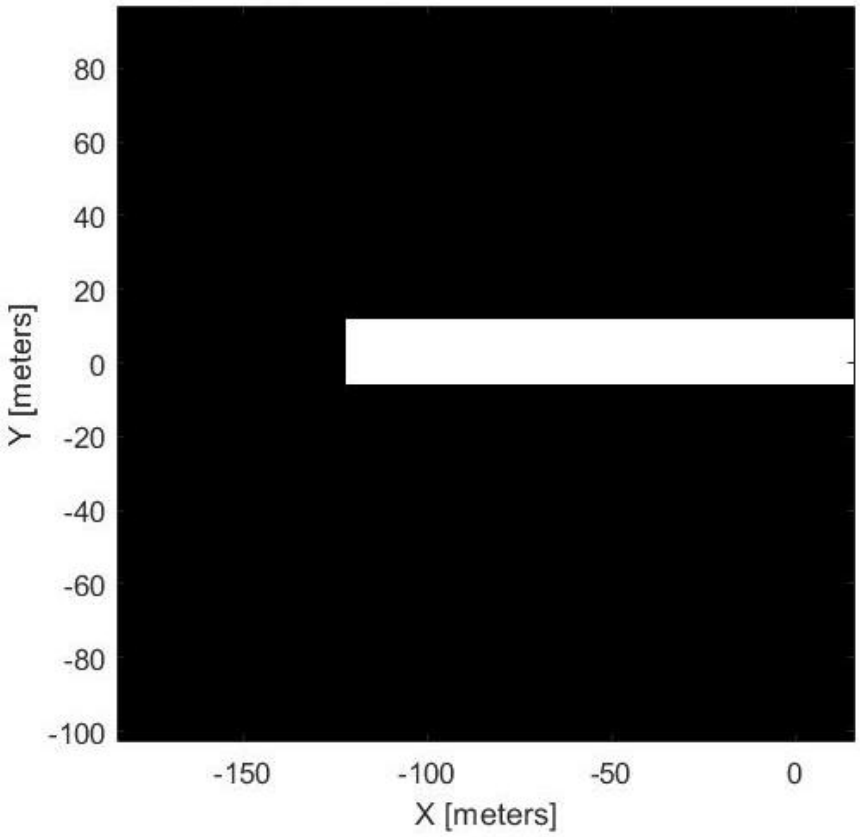

Figure 2-5: Binary occupancy map

The function ‘binaryOccupancyMap()’ in MATLAB helps create an occupancy map from a scenario canvas. The generated occupancy map from the scenario canvas is rotated 90 degrees to

the right compared to the actual orientation of the scenario canvas. The ego vehicle position is exactly same in both the scenario canvas and binary occupancy map. This binary occupancy map has exactly same coordinate system as of the scenario (XY). When it comes to path planning, the D* planner considers this map in terms of cells that are presented in the grid of this occupancy map. The number of cells in the occupancy grid depends on the resolution (discussed later) of the occupancy map.

Cells in the occupancy grid are the evenly spaced fields containing binary values (0 or 1) and represent free space or obstacle location in the environment. If the user defined height and width of the occupancy grid map is 200x200, and the resolution is 1, then the number of cells present in the grid or the size of the raw matrix containing the grid cell values (0 and 1) will be of 200x200. The occupancy map that was generated by the MATLAB function 'binaryOccupancyMap()' has a user defined resolution of 2 (default resolution is 1), meaning that the raw matrix containing the grid cell values will be of size 400x400. To increase the resolution of the map, the number of cells in the grid containing the occupancy values (0 or 1) has been increased. By increasing the resolution to 2 from 1, a single occupancy cell will be increased to 4 cells to define same occupancy.

This actual raw matrix of size 400x400 containing the grid cell values (0 and 1) is used as a grid map by the D* planner to plan a path. Any point in this 400x400 grid map is placed in a different coordinate system (true grid coordinates, IJ) (Figure 2-6). Where, 'I' (ranges from 0 to 400) corresponds to 'Y' coordinate in world coordinates and 'J' (ranges from 0 to 400) corresponds to 'X' coordinate in world coordinates. Each point in this grid map has a value of either 0 or 1.

The only difference between the binary occupancy map generated by the MATLAB function 'binaryOccupancyMap()', and the grid map used by the D* planner is that the former one is a 200x200 occupancy map representation of the actual scenario canvas with actual world coordinate system (XY) unchanged. Whereas the later one is nothing but the true grid representation of the occupancy map. Based on the resolution (which is 2 in this case) of the occupancy map, the raw matrix (400x400 in this case) containing the grid cell values (0 or 1) is used by the D* planner as a grid map (with IJ coordinates) for path planning.

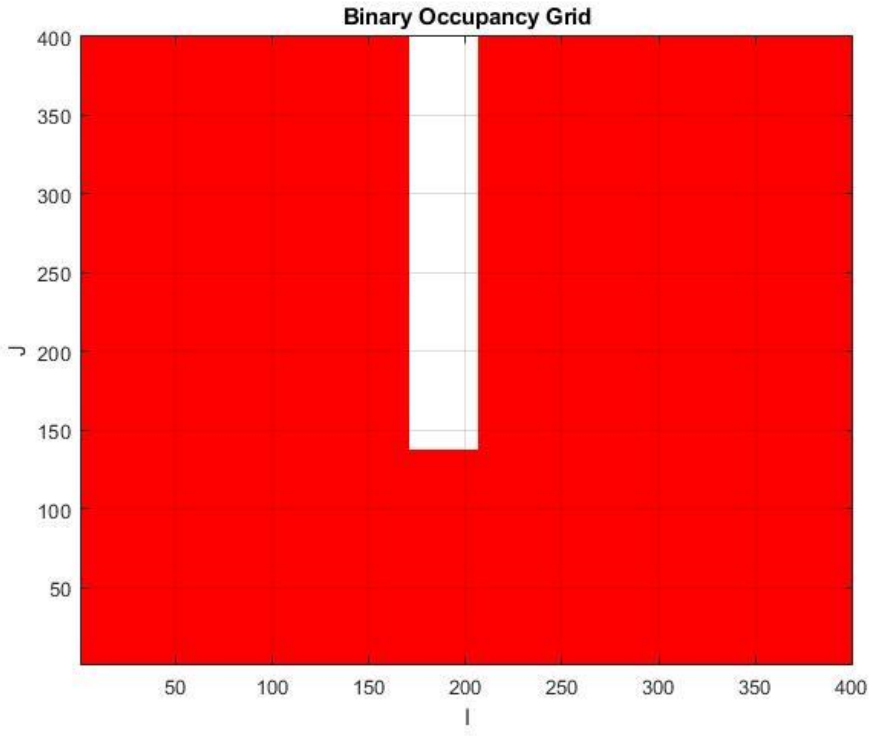

Figure 2-6: Grid map representing (IJ) coordinates

As the D* path planner used in this project assumes the ego vehicle as a point mass and does not account the vehicle geometry, to compensate for the ego vehicle dimension and to avoid unnecessary collision, an inflated binary occupancy map is used. By inflating the map, the obstacles are inflated, which makes the obstacle appear larger and reduces free space. The map should not be over-inflated so that the planner can find a path between start and goal points. The inflation formula that has been followed in this project is that the obstacle size is increased by a radius of half of the ego vehicle width. The grid map only increases the resolution of the binary occupancy map by using 4 cells instead of 1 cell in binary occupancy map for an occupied or unoccupied location. Inflating the size of the obstacle in binary occupancy map does not require the grid size to be increased or decreased.

The binary occupancy map is an 'ego-centric' map meaning that the ego vehicle will always be at the center of this map, so as the ego vehicle moves forward in the scenario at each time step, the map also moves forward in the environment keeping the ego vehicle in the middle of this map. Thus, the ego-centric binary occupancy map represents only 200x200 meters of the scenario canvas, and the center point of this occupancy map will be the ego vehicle position. As the ego vehicle moves forward in the scenario and the camera sensor senses any obstacle in the environment, the occupancy map gets updated with the obstacle information. The occupancy of the map changes to 1 for occupied space and 0 for unoccupied or free space. Every time the occupancy is changed, the obstacle points are saved in a matrix in the form of true grid coordinates (IJ) to be used later in the path planning algorithm for arc cost update during planning.

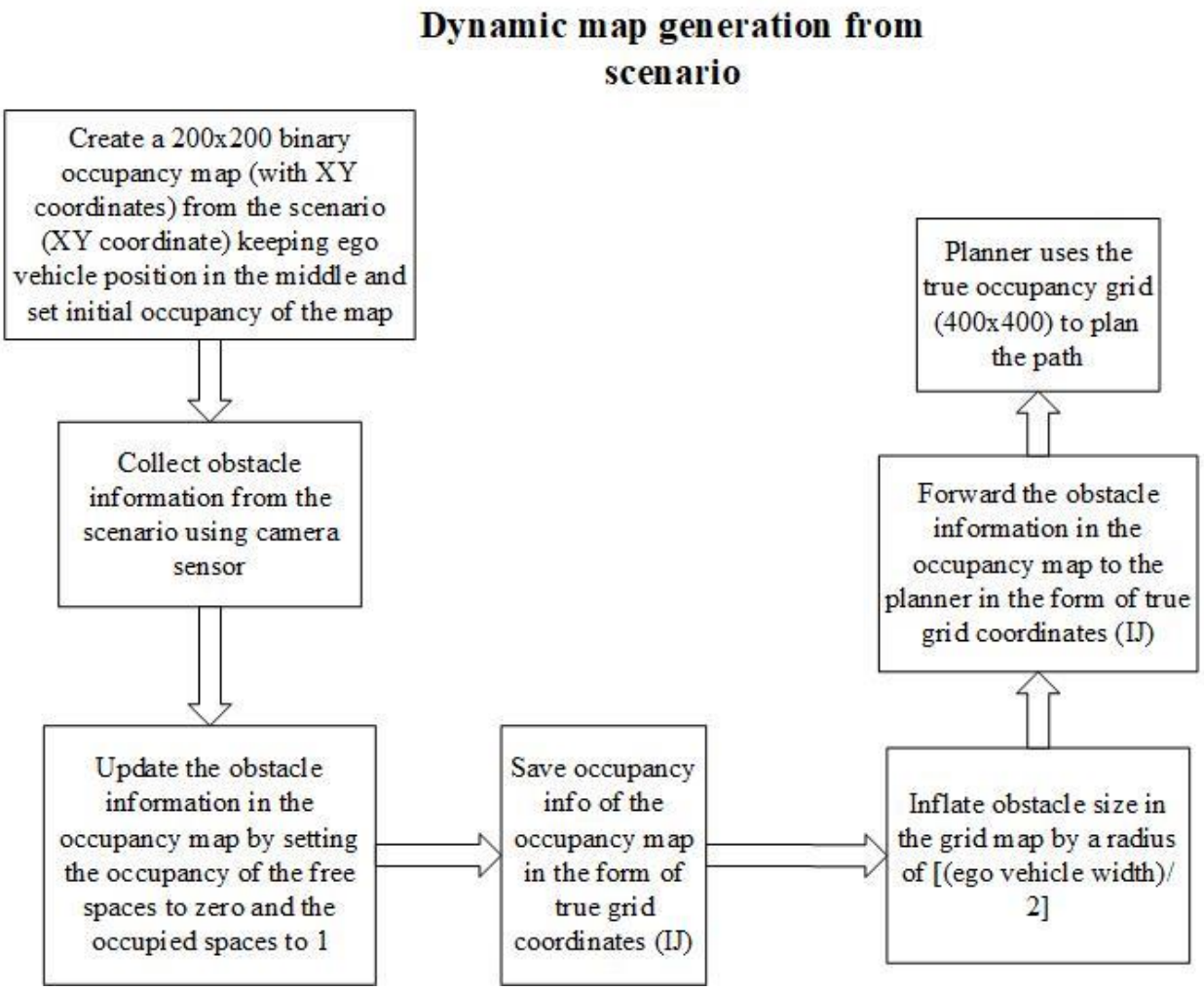

Figure 2-7: The flowchart of dynamic map generation

In Table 1, the dynamic map generation pseudocode is given. First, a binary occupancy map called *egoMap* is created with the help of a MATLAB function *binaryOccupancyMap* by defining the height (in meters), width (in meters) and resolution (cells per meter, every cell in the binary occupancy map contains a value that represents the occupancy of that particular cell, for occupied space the value is 1 and for free space the value is 0. If the resolution is 2, the grid containing the cells will be of size [(height*2) X (width*2)]). To get started with the grid map, it is required to define the occupancy of the cells initially with 0 or 1 for all the cells present (in this case 400*400)

in the grid, which is done by the MATLAB function *setOccupancy*. Then the scenario information (obstacle information, road borders) is collected by the ego vehicle from the scenario canvas with the help of a camera sensor using a function *exampleHelperGetObstacleDataFromSensor* from Navigation Toolbox in MATLAB. The function *'visionDetectionGenerator'* helps configure the camera sensor according to the specifications mentioned in previous chapter (Figure 2-2) and saves it in a variable called *sensor*. After the scenario information is available, the obstacle positions and the unoccupied positions in the occupancy map are saved in variables $obstaclePoints$ and $unoccupiedSpace$ respectively with the help of function $exampleHelperFilterObstacles$ available in MATLAB's Navigation Toolbox. After that, the occupancy of the map is updated according to the obstacle position and free space again with the help of the function $setOccupancy$. The obstacle position is also saved in the form of true grid co-ordinates (IJ) in a variable $obstaclePoints\_IJ$ using a MATLAB function called *world2grid* which helps transform the world (XY) coordinates to true grid (IJ) coordinates (function *grid2world* is used to convert grid coordinates to world coordinates). Then the obstacle positions in the grid map are inflated using the MATLAB function $inflate$.

Table 1: Pseudocode for dynamic map generation:

Table 1: Dynamic Map generation from Scenario
Initialization with Scenario Information and Camera Configuration $egoMap \rightarrow$ Grid map (with IJ coordinates) of the scenario $sensor \rightarrow$ Configuration of the sensor to detect lanes and obstacles $egoVehicle.Width \rightarrow$ Ego vehicle width $obsatcleInfo \rightarrow$ Obstacle information perceived by camera sensor $obstaclePoints \rightarrow$ Position of the obstacle in the world $unoccupiedSpace \rightarrow$ Unoccupied space in the world $obstaclePoints_{IJ} \rightarrow$ Obstacle position in grid coordinates **<u>Creating and Updating the Grid Map</u>** **While advancing scenario** **for every time step 0.1 second** **egoMap** → *binaryOccupancyMap* (200,200,2) *setOccupancy*(egoMap, ones(400*400)) **sensor** → *visionDetectionGenerator* (DetectionProbability, FalsePositivePerImage, UpdateInterval) **obstacleInfo** → *exampleHelperGetObstacleDataFromSensor* (sensor, egoMap) [obstaclePoints, unoccupiedSpace] → exampleHelperFilterObstacles (egoMap, egoVehicle, obstacleInfo) **if** *unoccupiedSpace* is not empty *setOccupancy*(egoMap, unoccupiedSpace, 0) obstaclePoints_IJ = world2grid (egoMap, unoccupiedSpace) **end** **if** obstaclePoints is not empty *setOccupancy*(egoMap, obstaclePoints, 1) obstaclePoints_IJ = world2grid (egoMap, obstaclePoints) **end** Inflate the obstacle size in $egoMap$ by a radius of $(egoVehicle.Width/2)$ **end** **end**

2.4 D* Path Planning Algorithm

D star (D*) path planning algorithm is used for generating real-time optimal traverses through a graph with updated or changing arc costs. The algorithm is named D* (Dynamic A*) because it has resemblance with A*, except that it is dynamic in nature, i.e., the arc cost can change or update while traversing the solution path. The D* planner is advantageous and computationally fast because most of the arc cost corrections take place in the ego vehicle vicinity and only a portion of the path is replanned out to the current ego vehicle location. In this project the D* algorithm is not written from scratch, but several functions are used from MATLAB toolbox 'Robotics Toolbox for MATLAB' to plan and construct path with modified arc cost.

At any point of time the arc cost corrections (as perceived by sensors) can be made, and the arc values which are known, measured, and estimated comprise the environment cost map by updating the map with proper cost. A* algorithm could be used for recomputing the cost map, but it is not efficient for large environments or if the goal location is very far away. D* marks affected states invalid, and also uses another approach of incremental and re-entrant repair where the optimal paths are repaired and then traversing through the optimal path is continued. Invalidated states are only processed when the ego vehicle is forced to enter that previously affected area.

D* algorithm, for expansion, maintains an OPEN and a CLOSED list of states. Where, in OPEN list the states are of two types: RAISE and LOWER. The RAISE states (costs of these states are higher than previous time they were on the OPEN list) are responsible for transmitting path cost increments due to an increased value of arc cost. Whereas the LOWER states (costs of these states are lower than previous time they were on the OPEN list) are responsible for reducing costs and redirecting the arrows (or 'Backpointer' referring to the neighbor node leading to the goal) so that new optimal paths can be computed. D* starts computing path by searching backward from the

target (goal) node. The RAISE states propagate the increment of arc cost through all the invalidated states and activate neighboring LOWER states to reduce the costs of traversing and redirect the backpointers. LOWER states compute new optimal paths to the previously raised states [39].

Regardless of cost correction order and propagation length, RAISE and LOWER states propagate cost increments and cost reductions respectively from the lowest to highest value of path cost in the map. In this process LOWER states can become RAISE states and vice versa. If the ego vehicle proceeds into a region which was previously invalidated, the D* algorithm propagates all the cost updates logged until that point of time into this region for computing a new optimal path to the goal position [40].

The notations used below are for the sake of understanding the D* algorithm, and have no contribution to the rest of the notations used in other chapters for development of this project [49]:

M, N → States of the robot vehicle

b(M) = N → Backpointer of a state M to next state N

c (M, N) → Arc-cost of a path from state M to N

t (M) → Tag of a state M [NEW (never been put on OPEN list), OPEN (currently present in the OPEN list) or CLOSED (no longer present in the OPEN list)]

h (M) → Cost of path to the goal from state M

k (M) → The least value of h(M) from the time the state M was placed in OPEN list

The robot vehicle can move in eight different directions (Figure 2-8). The values of the arc-cost c(M, N) are lesser for clear or obstacle free cells and larger for the grey-shaded cells or the obstacle occupied cells (N7 and N8). N in c(M, N) represents the cells N1, N2, N3, N4, N5, N6, N7 and N8 in Figure 2-8.

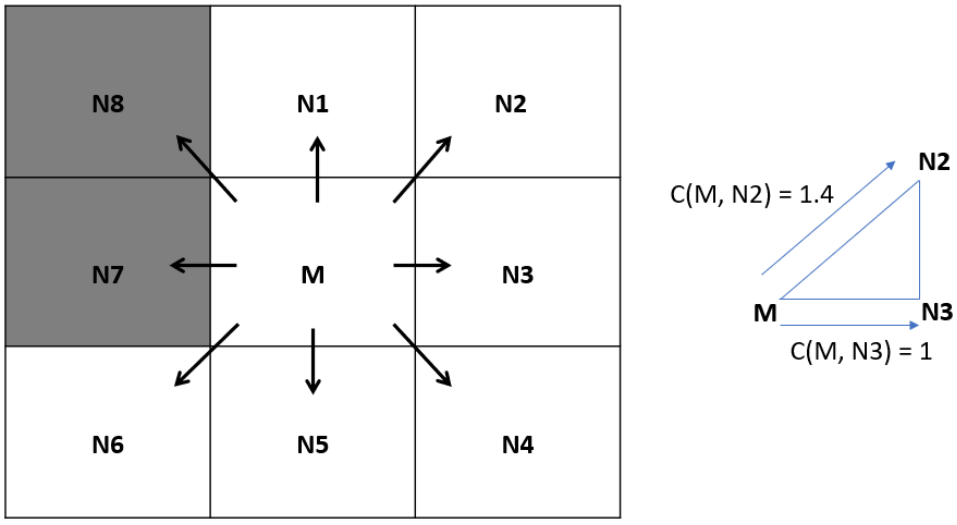

Figure 2-8: Directions that the robot vehicle can move towards

Cost of traversing in horizontal or vertical direction from one cell to another neighboring cell is considered as 1. Whereas cost of traversing to another cell in any diagonal direction is considered as 1.4 (determined from $\sqrt{1^2 + 1^2} = 1.4,$). The cost of traversing to any neighboring cell occupied with obstacle is considered as a very high number (for instance 10000).

	Cost of traversing in horizontal or vertical direction	Cost of traversing in diagonal direction
Obstacle free cells (states)	c(M, N2) = 1	c(M, N3) = 1.4
Obstacle occupied cells (states)	c(M, N7) = 10000	c(M, N8) = 10000

An overview of the D* algorithm [46]:

- The value of k(M) determines the priority of the state in the OPEN list to be expanded or evaluated.
- LOWER State:
    - k(M) = h(M)
    - Propagates information to the neighbor states when path cost is reduced (for instance, due to reduction of an arc cost or a new path to the goal)
    - For every neighbor N of LOWER state M:

      if *t(N) = NEW or h(N) > h(M) + c(M, N)* then
        - Set *h(N) = h(M) + c(M, N)*
        - Set *b(N) = M*
        - Inserts state N to OPEN list with k(N) = h(N) for it to propagate change of costs to neighbors.
- RAISE State:
    - k(M) < h(M)
    - Propagates information to neighbors about increment of path cost (for instance, due to an arc cost increment)
    - For every neighbor N of a RAISE state M:
        - If t(N)=NEW or [b(N) = M and h(N) ≠ h(M) + c(M, N)] then:

          insert N to OPEN list with k(N) = h(M) + c(M,N)
        - Elseif [b(N) ≠ M and h(N) > h(M) + c(M, N)] then:

          insert M to OPEN list with k(M) = h(M)

- Elseif [b(N) ≠ M and h(M) > h(N) + c(M, N)] then:

  insert N to OPEN list with k(N) = h(N)

- Expansion or evaluation of a state in the OPEN list:
  - An optimal path to the goal is computed through expansion of the states.
  - h(G) = 0 [G = Goal] is set in the beginning and h(G) is inserted it to OPEN list
  - Expansion happens with repetition until the state M of the robot vehicle is removed from the OPEN list

- Obstacle Handling:
  - This stage works immediately when an error in the arc-cost-function is detected by the robot vehicle (for instance, a new obstacle has been discovered)
  - This makes changes to the arc-cost function and enter affected states in the OPEN list

The below example shows how D* algorithm plans a path through an environment and handles an obstacle if found in the path [46][49]. In Figure 2-9, cell (2,1) is the Start state and cell (7,6) is the Goal state, and the expansion starts from the goal state (a ‘state’ is also called a ‘node’ or a ‘cell’) until the start node is reached.

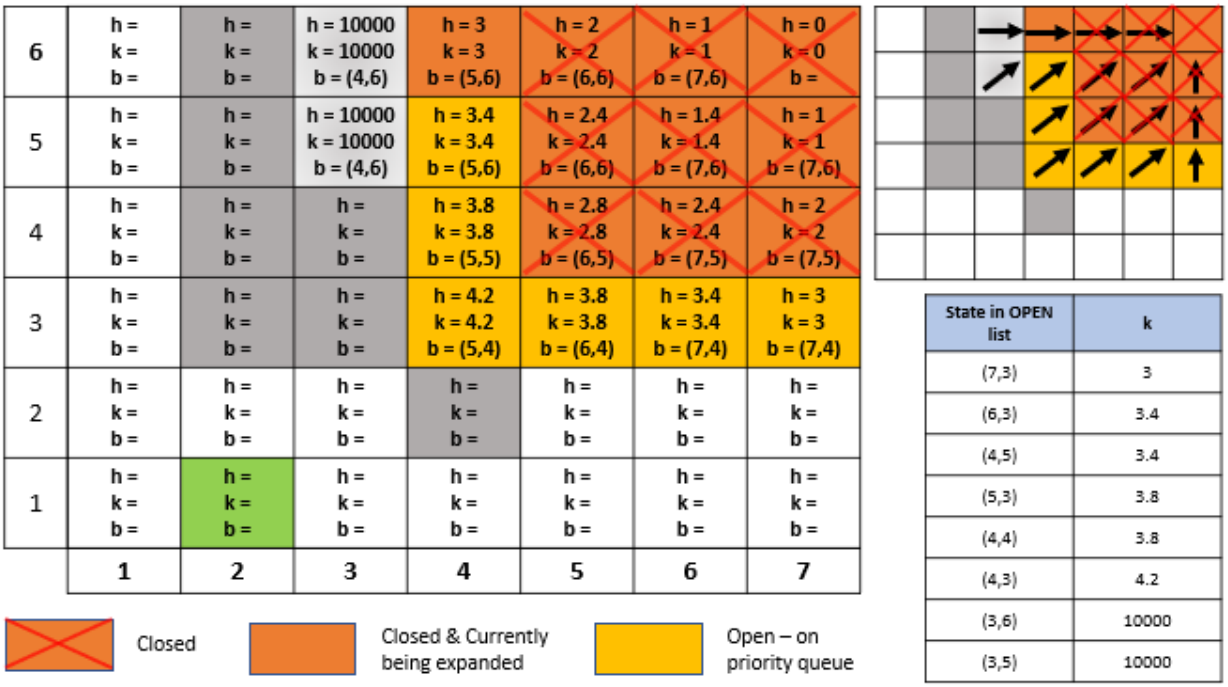

Figure 2-9: Expansion in D* starts from goal location

Each state in Figure 2-9 has a cost (h) associated with it, the minimum cost counted from the goal node to a particular state is k, and h is the path cost, which can change if any obstacle appears on the path. A state with the smallest value of 'k' in the OPEN list gets priority to be evaluated first among the other states with greater 'k' values in the OPEN list. 'b' is the backpointer referring to the state leading to the goal. Expansion ends when the start node (2,1) has been expanded (Figure 2-10). Then the optimal path is determined by following h value gradient (Figure 2-11).

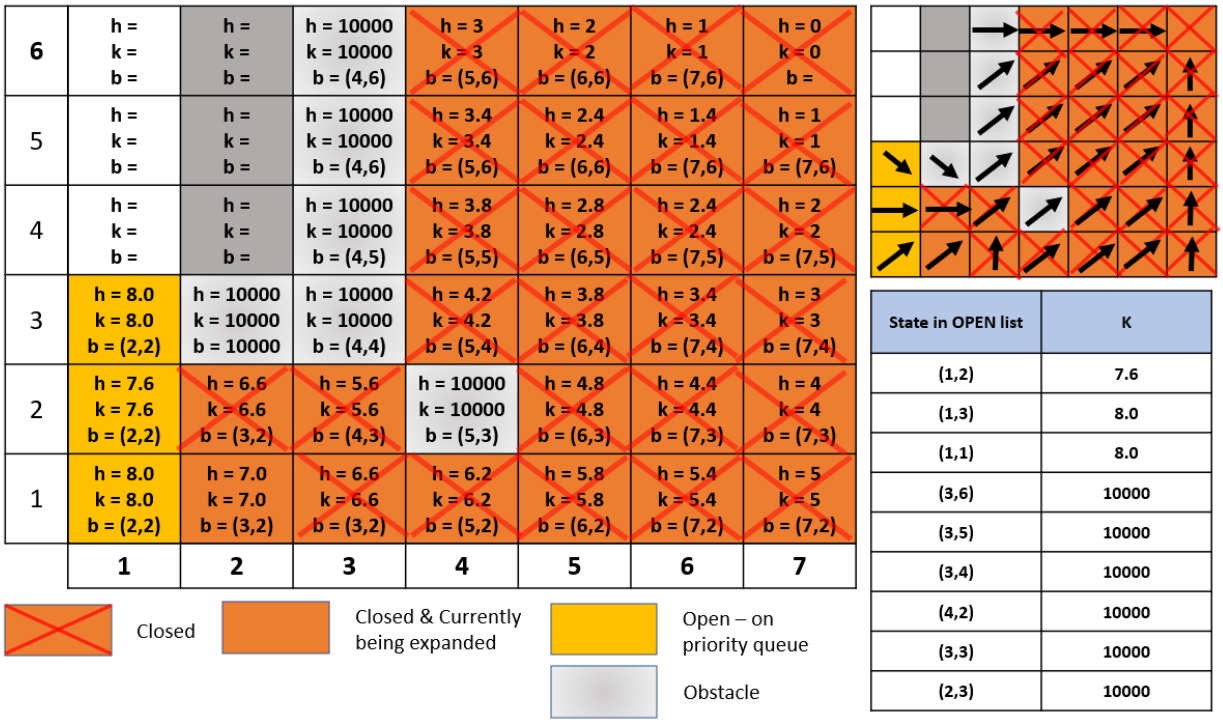

Figure 2-10: Expansion ends in D* when the start node is expanded

In Figure 2-10, cell (2,1) is expanded, and search ends here, some nodes ((1,1), (1,2), (1,3)) are still remaining in the OPEN list, and some ((1,4), (1,5), (1,6)) are not touched. Figure 2-11 shows that the final path is computed following the 'h' value gradient and backpointers from the start node to the goal node.

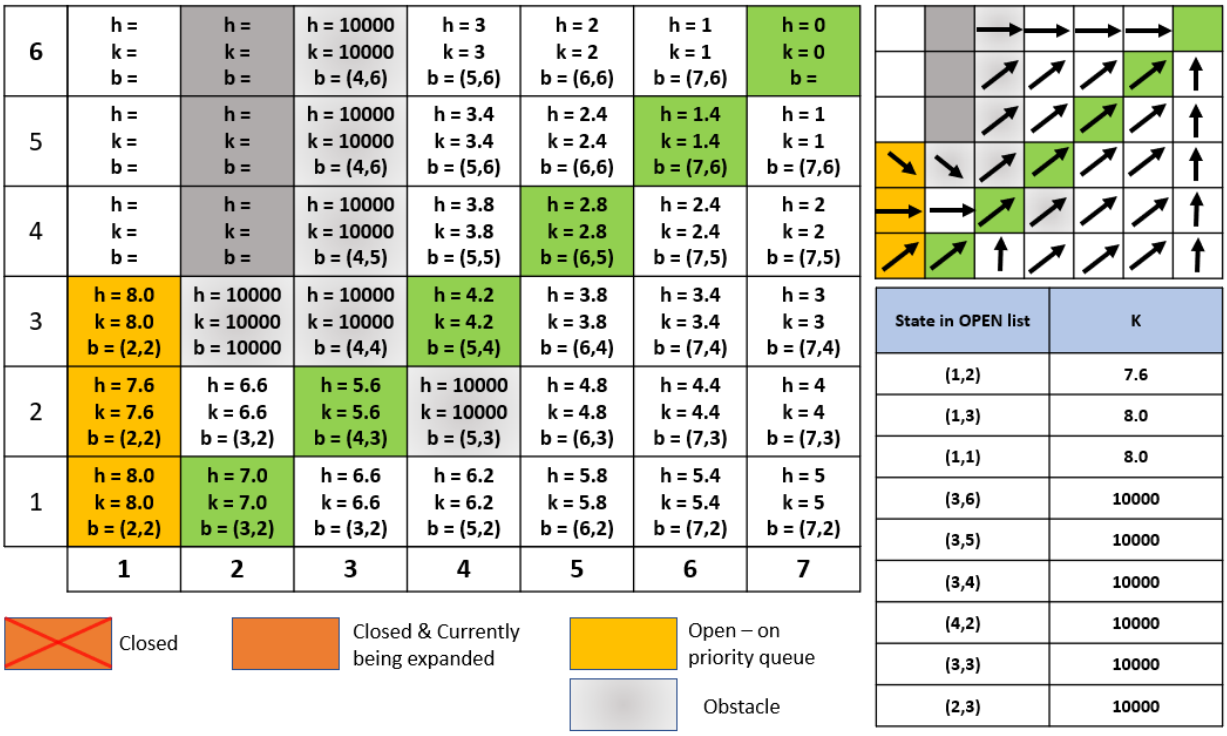

Figure 2-11: Optimal path to the goal is computed according to the gradient of 'h'

Now, an obstacle is detected by the ego vehicle in the way to the goal while following the optimal path (Figure 2-12).

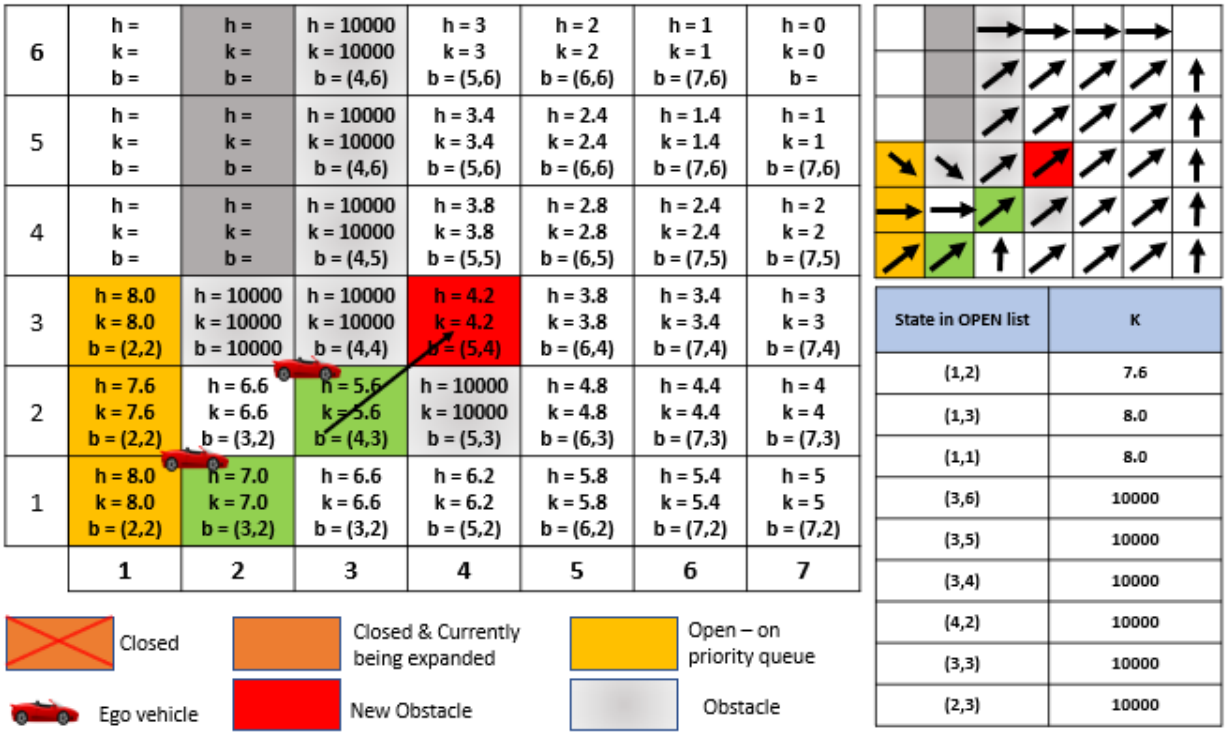

Figure 2-12: Obstacle detected by the robot vehicle while following the optimal path to the goal

When an obstacle is detected (in state (4,3) in Figure 2-12) on the optimal path, the 'h' value associated to the node containing the obstacle is raised, and all the nodes (all eight neighboring nodes) that are affected by the obstacle node are again put on OPEN list including the obstacle-node (Figure 2-13) for re-evaluation.

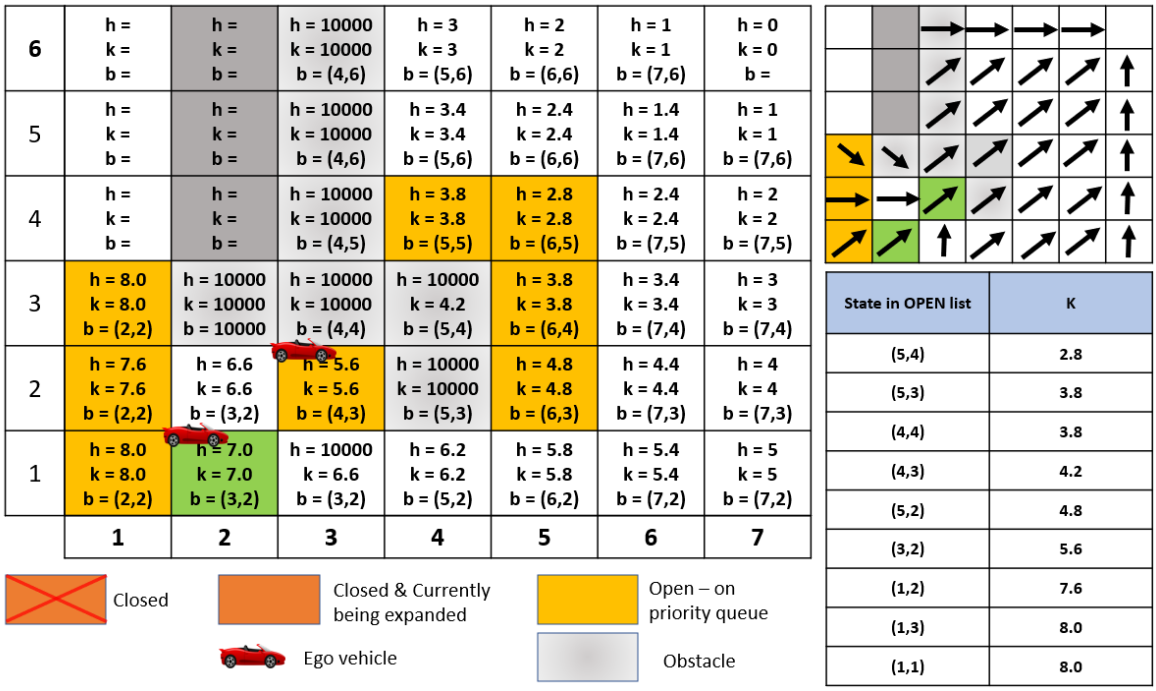

Figure 2-13: Obstacle node and neighboring nodes are put on OPEN list for re-evaluation

(5,4) is evaluated first because it has the lowest ‘k’ value (Figure 2-14). Since (4,3) is neighbor of (5,4) with a backpointer to (5,4), and the ‘h’ value of (4,3) is recently raised due to the presence of an obstacle (now (4,3) has h>k), (4,3) is put on the OPEN list as a RAISED state. Next, (5,3) and (4,4) are expanded respectively, but nothing will be changed because no surrounding states are new and path-costs (h) of those surrounding states are correct (same as previous). Next, (4,3) is expanded and the states (in this case (3,2)) whose backpointers lead to the goal through (4,3) are placed on the OPEN list with updated high path-cost (‘h’ value). After that, (5,2) is expanded and it does not affect anything, because no surrounding state of (5,2) is new and path-costs (‘h’ values) of those surrounding states are correct.

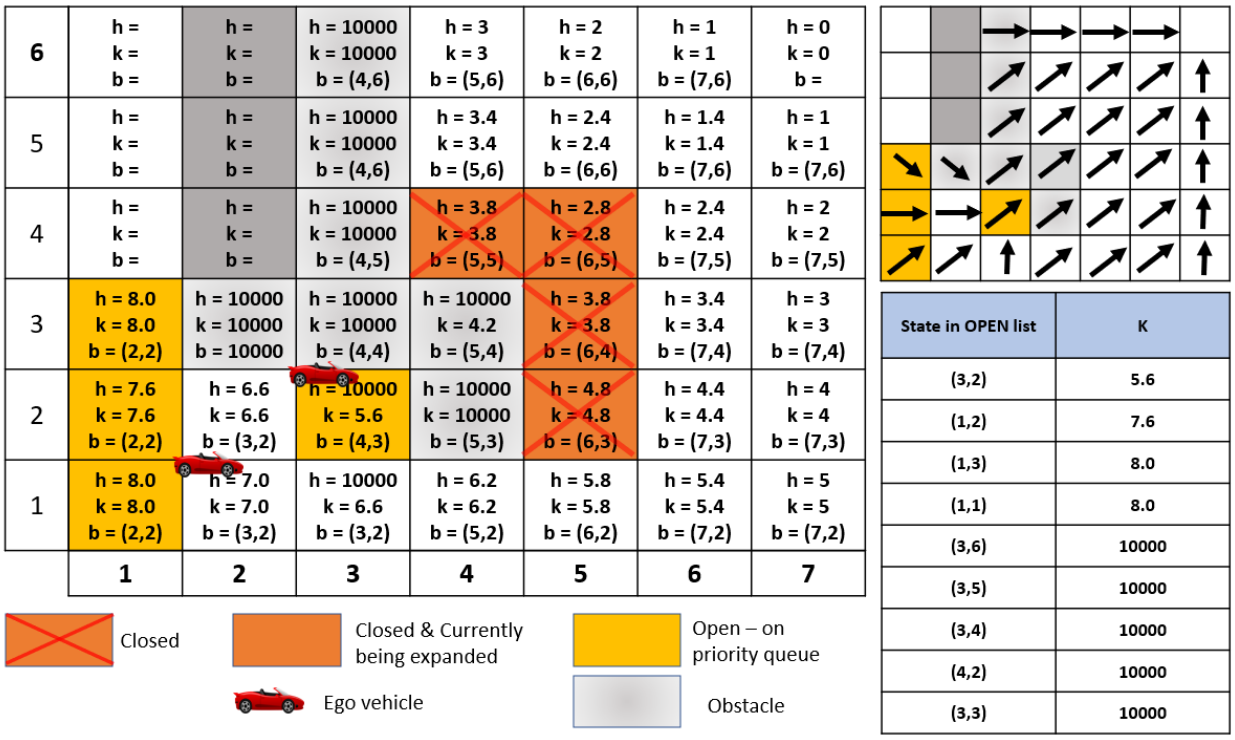

Figure 2-14: Evaluation of neighboring nodes of obstacle node

Next, the state (3,2) is expanded because its ‘k’ value is the smallest in the OPEN list (Figure 2-15). But now, the ‘h’ value of state (3,2) is higher than its ‘k’ value ((3,2) is now a RAISED state). When a state becomes a RAISED state, its backpointer may no longer point to an optimal path leading to the goal. After (3,2) is expanded, states (2,1), (2,2), (3,1), and (4,1) are put on the OPEN list (Figure 2-15). States (2,1), (2,2), (3,1) have backpointers pointing toward the state (3,2) i.e., optimal path from these states leading to the goal passes through the new-found obstacle, so these states receive the raised path-cost (h) from state (3,2) and are put on the OPEN list as RAISED states. The optimal path from the state (4,1) does not pass through the new-found obstacle, therefore the ‘h’ value does not change for state (4,1).

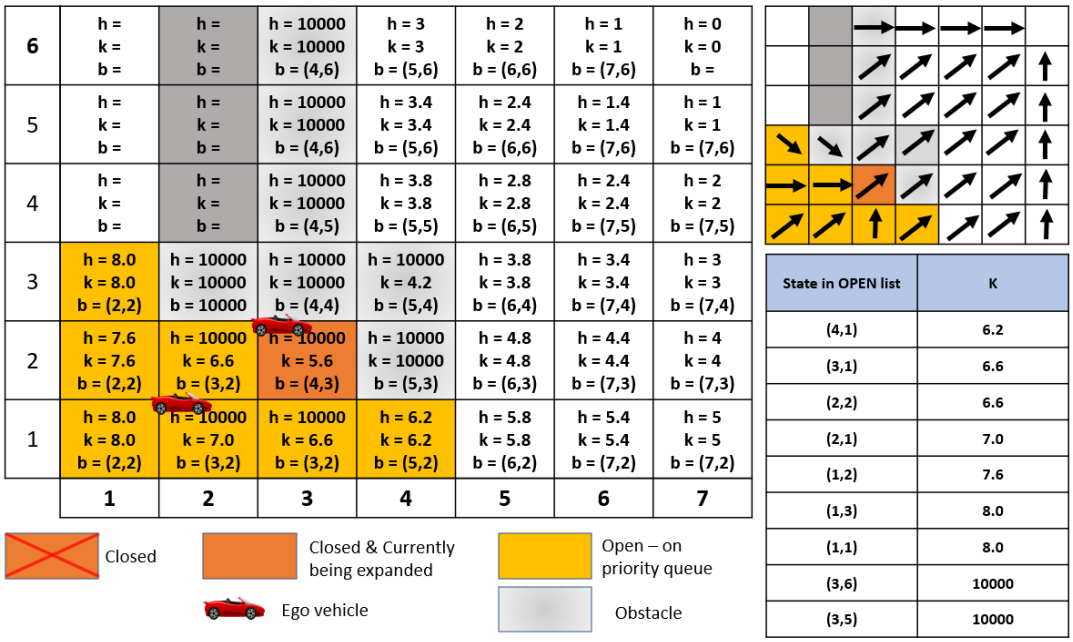

Figure 2-15: Obstacle handling and cost propagation

Next, the state (4,1) is expanded as its 'k' value is the smallest in the OPEN list (Figure 2-16), and the states (3,1), (3,2), (5,1), (5,2) are put on the OPEN list (priority queue). The path-cost (h) for the states (5 ,1) and (5,2) does not change. But, for the state (3,2), the goal state now can be reached from (3,2) with a path cost of [h(4,1) + c((4,1), (3,2)) = 6.2 + 1.4 = 7.6], where h(4,1) is the path-cost of state (4,1) from the goal state and c((4,1),(3,2)) is the arc-cost from state (4,1) to state (3,2).

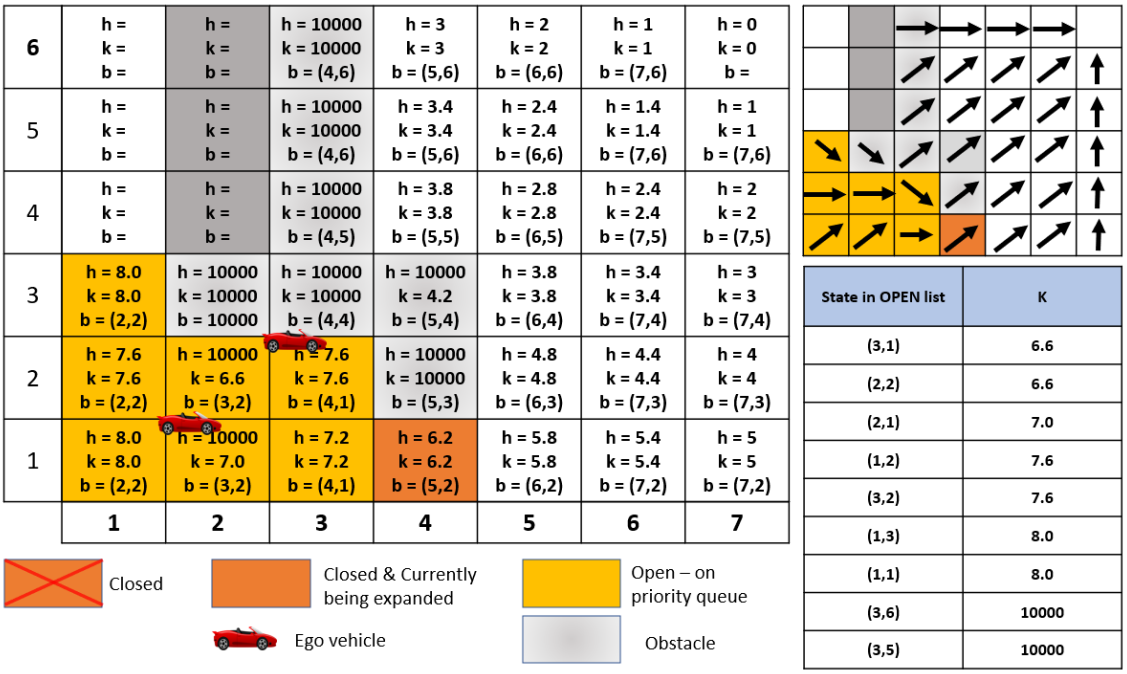

Figure 2-16: Obstacle handling and back pointer update

Now, in Figure 2-16, the state (3,2) has 'h' value of 7.6 and new 'k' value is of 7.6 (as $k_{new} = min\ (h_{old}, h_{new})$, where h_{old} is 10000 and h_{new} is 7.6). As 'h' value and 'k' value of the state (3,2) are now same, the state (3,2) is now a LOWER state (previously (3,2) was RAISE state because 'h' value was more than 'k'). Now the backpointer of (3,2) is set toward the state (4,1) because (3,2) became a LOWER state after (4,1) changed the path-cost (h) of (3,2). The optimal path to the goal starting from the state (4,1) does not pass through the state (4,3), therefore it is still an optimal path even after the new obstacle is found in state (4,3). Thus, an optimal path to the goal is found from the robot vehicle's current state (3,2) through state (4,1), and the expansion process ends here. The optimal path to the goal is determined by following the gradient of path-cost (h) (Figure 2-17).

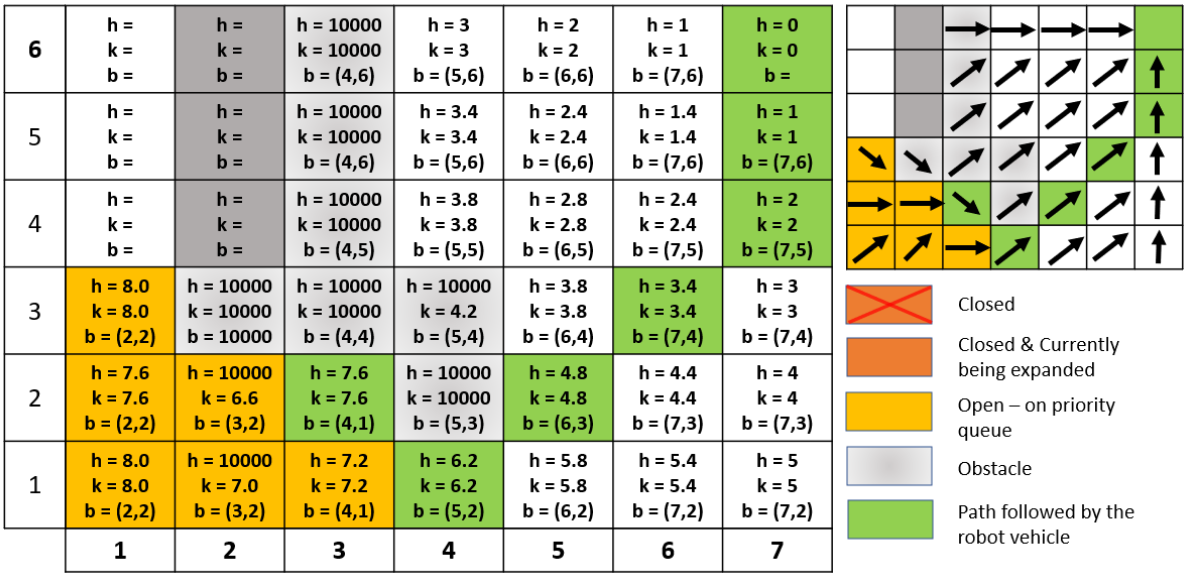

Figure 2-17: Obstacle handling and optimal path computing

Now robot vehicle (ego vehicle) in the figure below (Figure 2-18) traverses in the optimal path continuing from cell (3,2).

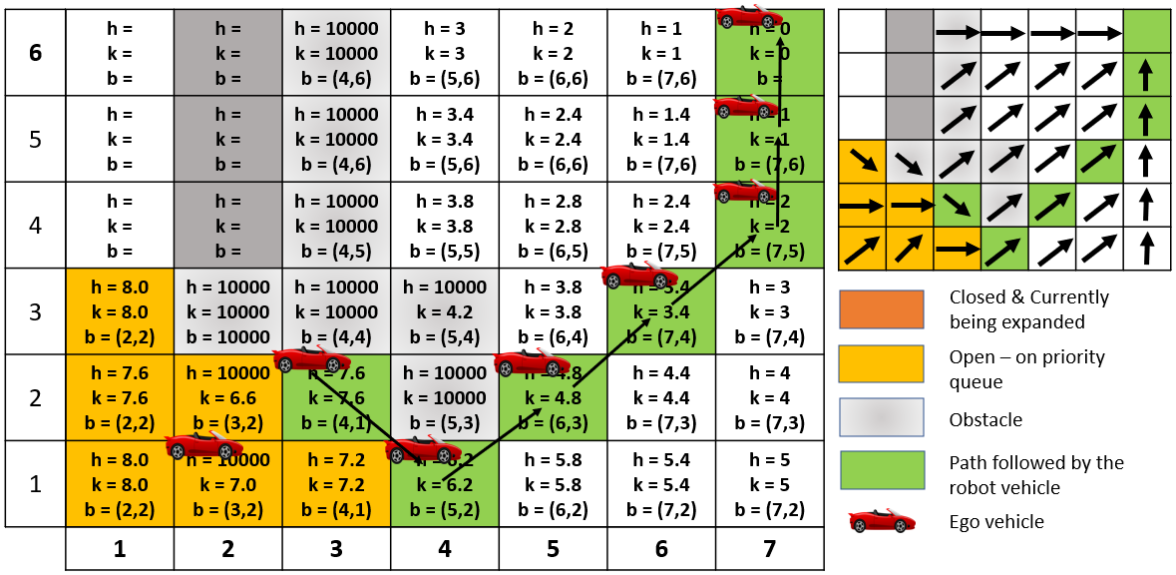

Figure 2-18: Optimal path traversing continues after avoiding the obstacle

In this project, instead of writing the D* algorithm from scratch, a MATLAB toolbox (Robotics Toolbox for MATLAB) is used to implement D* algorithm with the help of a number of functions: *plan* (computes the cost of traversing to cells in any direction for a given goal location and an environment map), *query* (finds an optimal path to the goal).

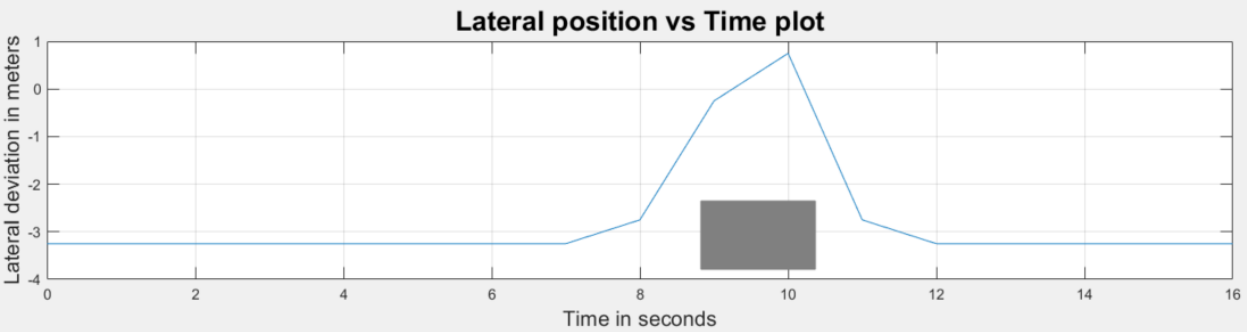

Figure 2-19: Sample reference path generated by D* algorithm

2.5 Trajectory Generation:

The path generated by the planner is a set of waypoints which creates a path that a vehicle cannot follow because it has sharp turns. A vehicle cannot move directly towards sideways because of its rolling constraints which limits the possible direction of motion. Such constraints are called nonholonomic constraints of vehicles. To address this issue and for the vehicle to follow the planned path considering the nonholonomic constraints, the planned path is then smoothened using a polynomial trajectory generator.

The D* planner generates an optimal path in the form of waypoints consisting of desired longitudinal and lateral positions of the vehicle. Cubic B spline trajectory generator takes these waypoints and then converts it to a continuous smoothened curve that the vehicle can follow. The desired lateral position generated from the trajectory generator is then fed to the MPC controller

for the lateral control of the vehicle. Longitudinal Control of the vehicle is done by calculating a safe distance from the lead vehicle and then feeding it to the MPC controller to maintain it accordingly.

A piecewise cubic B-spline trajectory is generated with this trajectory generation method. This trajectory falls in the control polygon which is defined by control points. The output coordinate points for the newly generated trajectory are sampled uniformly in a time interval, decreasing this interval will result in a smoother curve [40]. It is a local control-based polynomial where if a control point changes its position the respective segment of the curve will change its shape, not the entire curve.

The definition of a B-spline curve of degree p is given as:

$$B(t) = \sum_{i=0}^{n} N_{i,p}(t)\, P_i \tag{2.1}$$

Where control points are represented as P_i , and a normalized B-Spline $N_{i,p}(t)$ is defined over the knots (t) [47].

In this project, a MATLAB function *bsplinepolytraj* is used to generate a smooth trajectory which takes the control points and the interval for sampling the trajectory points as input arguments.

As at each time step the algorithm plans the path and generates the trajectory, the start position should be the current ego vehicle position. And at each time step the planner plans the path, the trajectory gets generated from the current ego vehicle position. The number of waypoints generated for the trajectory are same for each time step. After the MPC computes the required

velocity for the vehicle to run at, MATLAB function *trajectory* helps the ego vehicle travel along the given trajectory and with the controlled speed in the driving scenario simulation.

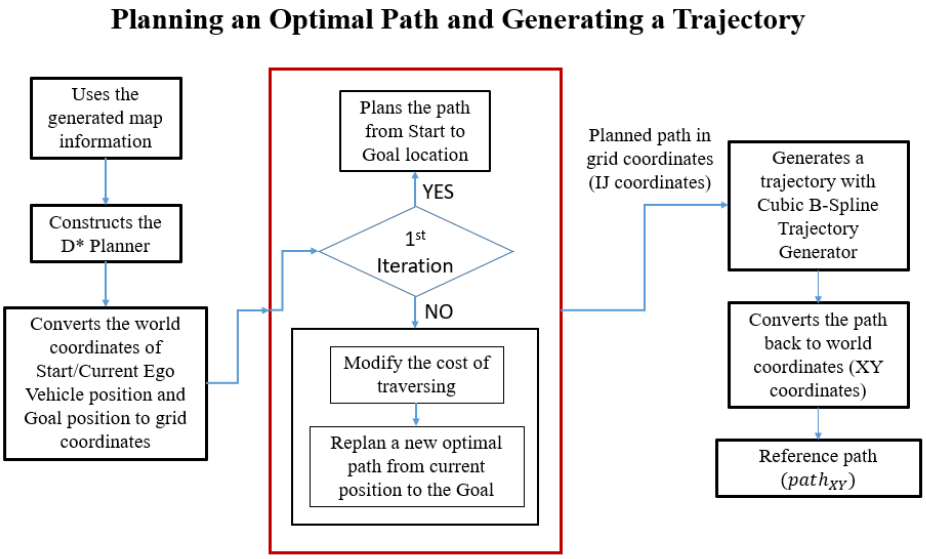

Figure 2-20: Optimal path and trajectory generation flowchart

In Table 2, the pseudocode for the path planning and trajectory generation is given. The variable 'mapData' contains the occupancy information of the 'egoMap'. And the function 'Dstar' available from 'Robotics Toolbox for MATLAB' in MATLAB constructs the planning space for D* planner using the egoMap information from 'mapData' and saves it in the variable 'dstar'. As the planner uses the grid map for path planning, the start and goal locations of the ego vehicle are converted to the grid coordinates (IJ coordinates) from world coordinates (XY coordinates). On the first iteration of the algorithm, the D* planner plans an optimal path to the goal from start position. From the next iteration the D* planner modifies the cost of traversing based on the obstacle information perceived form the grid map and replans a new optimal path to the goal. Now the function 'query' available from 'Robotics Toolbox for MATLAB' in MATLAB queries a route

from current ego vehicle position to the goal position. The path generated from this planner is in grid coordinates (IJ coordinates) and is saved in variable '$path_{IJ}$' in the form of waypoints. The waypoints in $path_{IJ}$ are used by the trajectory generator as control points. The function *bsplinepolytraj* available in MATLAB generates a smooth trajectory with the control points at a specified sampling rate. Increasing this sampling rate (or, decreasing the sampling time) will result in a smoother trajectory. Now, the generated trajectory is converted to the world coordinate form (XY coordinates) so that the ego vehicle can follow the waypoints in the scenario.

Table 2: Path Planning and Trajectory Generation

$mapData \rightarrow$ Occupancy information of the egoMap
$dstar \rightarrow$ D* planner construction
$goal_{XY} \rightarrow$ Ego vehicle goal position on world coordinate
$q \rightarrow$ positions of the trajectory at the given sample times
$egoPose \rightarrow$ Ego vehicle position
$k \rightarrow$ variable to count the instances of running a loop

Path Planning

While advancing scenario

k = 1

for every time step 0.1 seconds

$mapData \rightarrow$ Occupancy information of the egoMap

$dstar \rightarrow$ Construct the D* planner

Convert world coordinates (XY) of start & goal locations to grid location

$goal_{IJ} \rightarrow$ Goal location in grid coordinates

$start_{XY} \rightarrow$ Current vehicle position in world coordinate

$start_{IJ} \rightarrow start_{XY}$ in grid coordinates

if (k = = 1):

$plan\left(dstar, goal_{IJ}\right) \rightarrow$ Plan all routes to the goal in grid coordinates

else:

Modify the cost of traversing

Replan a new optimal path to the goal

end

Query a route from $start\_IJ$ in grid coordinates

$path_{IJ} \rightarrow query(dstar, start_{IJ})$

end

k = k + 1

end

return $path_{IJ}$

Trajectory Generation

While advancing scenario
for every time step 0.1 seconds
$m \rightarrow transpose\left(path_{IJ}(:,1)\right)$
$n \rightarrow transpose\left(path_{IJ}(:,2)\right)$

$cpts \rightarrow [m, n] \rightarrow$ Points for control polygon of B-Spline trajectory
$tpts \rightarrow [0\ 5] \rightarrow$ Start and end times for the trajectory points to be sampled for
$tvec \rightarrow$ Sample rate for trajectory points, outputs are sampled at this interval
$[q] \rightarrow bsplinepolytraj(cpts, tpts, tvec)$
$path_{IJ} \rightarrow round\left(transpose(q)\right) \rightarrow$ Waypoints in the grid coordinates
$path_{XY} \rightarrow$ Convert the path back to the world coordinates
$Desired_{Lat} \rightarrow$ Y coordinates of the $path_{XY}$
end
end

return $path_{XY}$

As the trajectory information obtained from the above algorithm, it is then used to calculate the desired yaw angle and extract desired lateral position of the ego vehicle, which are then fed to the Model Predictive Controller for lateral control of the vehicle. The longitudinal control of the vehicle requires the value of safe distance which is obtained from the obstacle information and ego vehicle information in the scenario.

3. Model Predictive Control for Vehicle Longitudinal and Lateral Control

In this chapter, designing of Model Predictive Control for vehicle longitudinal and lateral control is discussed. The prediction model is used to control the throttle, brake, and steering command to track the commanded vehicle velocity, maintain a safe headway distance from the lead vehicle, and execute the planned path. The Model Predictive Control is used because of its capability to maintain constraints on the states or plant outputs and control inputs as well. This is very crucial for vehicle control as a vehicle is constrained by its own mechanics and the environment.

3.1 Model Predictive Control

The Model Predictive Control is a discrete-time control method and the general objective of MPC is to determine a trajectory of future manipulated variables or control inputs to optimize the future performance of the plant outputs. To perform the optimization within a limited time window, the plant information must be provided at the start of the time window. Designing of Model Predictive Control systems is based on a mathematical model of the plant, which is a state-space model [41]. The state-space model is used explicitly for predicting the plant outputs at future instants (prediction horizon) of time. MPC calculates sequence of control inputs after solving a quadratic programming (QP) optimization problem and minimizing an objective function. MPC also follows receding horizon strategy where the horizon is moved toward the future at each instant, and the first control signal of the sequence calculated at each time step is applied to the plant [42].

Some important objectives of an MPC controller are preventing violation of input and output constraints, optimizing some output variables while keeping others in specified ranges, controlling a major number of system variables upon unavailability of a sensor or an actuator. The three critical

steps involved in in the process of a Model Predictive Controller are given as prediction model, optimization, and feedback correction [43].

3.2 Prediction Model

The cornerstone of a Model Predictive Controller is the prediction model. All the necessary mechanisms must be included in a design to obtain a best possible model. This model must be complete enough so that it completely captures the system dynamics and allows the prediction calculation. The necessity of calculating the predicted output at future instants determines the use of the plant model. Various models can be used in different strategies of Model Predictive Control to represent how the outputs and the measured inputs are related. In these measured inputs some are manipulated variables others can be considered as measured disturbances. A disturbance model can also be used for describing the behavior what the plant model does not reflect. These behaviors include the effects of unmeasured inputs, noise, and model errors [44].

The prediction model discussed below is a combination of longitudinal and lateral control model of the ego vehicle. The longitudinal control model is used to calculate the desired tractive force and the lateral control model is responsible for calculating the steering angle to execute a planned path.

3.2.1 Longitudinal Control Model

The longitudinal control model of the ego vehicle is used to express the relationship between the traction force and the vehicle acceleration which are responsible for influencing longitudinal motion of the vehicle. In longitudinal control, the manipulated variable tractive force (F_x) is required for achieving the desired acceleration or deceleration ($\ddot{x}_{desired}$) as determined by the controller.

The longitudinal control system of the vehicle consists of an upper-level controller and a lower-level controller. The upper-level controller is responsible for determining the desired acceleration for the ego vehicle, whereas the lower-level controller is responsible for determining the throttle input required to track the desired acceleration. As the lower-level controller is associated with the finite bandwidth, it is expected that the vehicle would not track the desired acceleration perfectly, which is why the lower-level controller has a first-order lag in its performance [34].

$$F_x = m\ddot{x}_{desired} \tag{3.1}$$

$$\ddot{x}_{actual} = \frac{1}{\tau s+1}\, \ddot{x}_{desired} \tag{3.2}$$

$$or,\ \tau\dddot{x}_{actual} = \ddot{x}_{desired} - \ddot{x}_{actual} \tag{3.3}$$

In equation (3.1) and (3.2), x_{actual} is the vehicle longitudinal position measured from a reference frame which is the initial position of the ego vehicle in the driving scenario canvas. It is assumed that the actual acceleration of the ego vehicle tracks the desired acceleration with a time constant τ. In this case, a lag of $\tau = 0.5$ is assumed for simulation and analysis.

Now, in the longitudinal vehicle dynamics, the opposing forces should also be considered, where it is assumed that the road the ego vehicle is moving on is not inclined and the wind velocity is zero. The two opposing forces considered are aerodynamic force and the rolling friction. The frontal area of the vehicle is given by A_f, mass density of the air is ρ, aerodynamic drag coefficient is C_d, mass of the vehicle is m, the rolling friction is μ. $\ddot{x}_{ego}$ is ego vehicle acceleration and $\dot{x}_{ego}$ is the velocity of ego vehicle.

If m is known and is between the range of 800 to 2000 kg, then the relationship between the vehicle mass and the frontal area is given by [34]:

$$A_f = 1.6 + 0.00056(m - 765) \quad (3.4)$$

The vehicle dynamics is then given as:

$$m\ddot{x}_{ego} = m\ddot{x}_{actual} - \left(F_{ad} + F_f\right) \quad (3.5)$$

$$m\ddot{x}_{ego} = m\ddot{x}_{actual} - F_{loss} \quad (3.6)$$

Where, F_{ad} is the aerodynamic drag force and F_f is the rolling friction force. These two forces together create loss (F_{loss}) to actual force.

$$F_{ad} = \frac{1}{2}\rho C_d A_f \left(\dot{x}_{ego} + V_{wind}\right)^2 \quad (3.7)$$

and $F_f = \mu mg$ (where g is gravitational acceleration)

The velocity $\dot{x}_{ego}$ of the ego vehicle is obtained from the MATLAB Driving Scenario Designer. The responsibility of the controller is to track the ego vehicle velocity to a commanded reference velocity. In an adaptive cruise control system, the ego vehicle tracks a commanded reference speed when no lead vehicle is in front, or it is far away and maintains a safe distance from a lead vehicle if there is one. The model designed until this point is capable of tracking a commanded velocity with the help of required tractive force. Now, to maintain a safe distance from a lead static or dynamic obstacle, a control strategy needs to be implemented. To maintain a safe distance from the lead vehicle, the position of the lead vehicle and the relative distance between the ego vehicle and the lead vehicle are required. The position of the lead vehicle and lead vehicle velocity (v_{lead}) can be obtained from the monocular camera sensor. A constant time gap approach is used to control the spacing between the ego and lead vehicles. The controller maintains a time gap

(t_{gap}) from the lead vehicle. The Safe Distance is the distance required to maintain the time gap, and is given by:

$$D_{safe} = D_{initial} + (t_{gap} * \dot{x}_{ego}) \quad (3.8)$$

In equation (3.8), $D_{initial}$ is the default spacing when the vehicles are standstill, i.e., when the velocity of both vehicles are zero. In this design as shown in Figure 3-1, the controller tries to track the error (e_s) between the relative distance (D_{rel}) and the Safe Distance (D_{safe}) to zero, where, $D_{rel} - D_{safe} = e_s$.

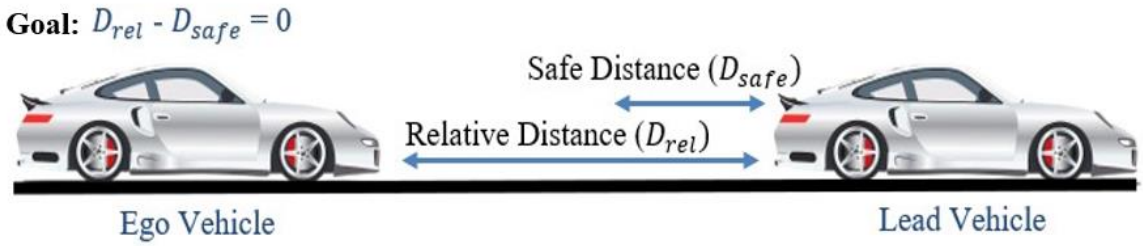

Figure 3-1: Spacing Control Representation of Ego Vehicle

$$\dot{\boldsymbol{e}}_{\boldsymbol{s}} = v_{lead} - \dot{x}_{ego} - [t_{gap} * \ddot{x}_{ego}] = v_{lead} - \dot{x}_{ego} - t_{gap} * \left[\ddot{x}_{act} - \frac{F_{loss}}{m}\right] \quad (3.9)$$

$$\ddot{\boldsymbol{x}}_{\boldsymbol{ego}} = \ddot{x}_{act} - \frac{F_{loss}}{m} \quad (3.10)$$

$$\dddot{\boldsymbol{x}}_{\boldsymbol{act}} = -\frac{1}{\tau}\ddot{x}_{act} + \frac{1}{\tau * m}(F_x) \quad (3.11)$$

Combining the equations (3.3) (3.6), (3.8), (3.9), (3.10) and (3.11), the state-space form of the prediction model for the longitudinal control of the ego vehicle can be represented with states $x_1 =$

$\begin{bmatrix} e_s & \dot{x}_{ego} & \ddot{x}_{act} \end{bmatrix}^T$ and the control input to the plant is given by $u_1 = [F_x]$. The resistance force (F_{loss}) and the velocity of the lead vehicle (v_{lead}) are considered as measured disturbances which update at every sample time and are integrated into the system model. The disturbance vector is given by $w = [F_{loss} \quad v_{lead}]^T$. And the output vector of the longitudinal model is given by $y_1 = \begin{bmatrix} e_s & \dot{x}_{ego} \end{bmatrix}^T$. The final state-space model of the prediction model for the vehicle longitudinal control is given by equations (3.12) and (3.13).

$$\frac{\mathrm{d}}{\mathrm{dt}}\begin{bmatrix} e_s \\ \dot{x}_{ego} \\ \ddot{x}_{act} \end{bmatrix} = \begin{bmatrix} 0 & -1 & -t_{gap} \\ 0 & 0 & 1 \\ 0 & 0 & -1/\tau \end{bmatrix} * \begin{bmatrix} e_s \\ \dot{x}_{ego} \\ \ddot{x}_{act} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\tau m} \end{bmatrix} * [F_x] + \begin{bmatrix} \frac{t_{gap}}{m} & 1 \\ -\frac{1}{m} & 0 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} F_{loss} \\ v_{lead} \end{bmatrix} \quad (3.12)$$

$$y_1 = \begin{bmatrix} e_s \\ \dot{x}_{ego} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} e_s \\ \dot{x}_{ego} \\ \ddot{x}_{act} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} * [F_x] \quad (3.13)$$

3.2.2 Lateral Control Model

In this section the prediction model for lateral control of the vehicle is designed with a linear bicycle model obtained from a nonlinear model [34]. The bicycle model of the ego vehicle has two degrees of freedom which are represented as the lateral position y and yaw angle ψ of the vehicle, which are the outputs of the system as well. The control input is steering angle which is given by δ. The distances of the front and rear tire from the center of gravity (c.g.) of the vehicle are represented as l_f and l_r respectively. The corning stiffness of each of the front tires is given by C_f. And the corning stiffness of each of the rear tires of the vehicle is given by C_r. I_z represents the vehicle moment of inertia with respect to the yaw axis. $\dot{y}$ and $\dot{\psi}$ represent the lateral velocity and the yaw angle rate of the vehicle, respectively. The prediction model for the lateral vehicle control

has states $x_2 = \begin{bmatrix} y & \dot{y} & \psi & \dot{\psi} \end{bmatrix}^T$ and input $u_2 = [\delta]$. The output from the system is $y_2 = [y \quad \psi]^T$. The state-space model is shown in equations (3.14) and (3.15).

$$\frac{\mathrm{d}}{\mathrm{dt}}\begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & a_{22} & 0 & a_{24} \\ 0 & 0 & 0 & 1 \\ 0 & a_{42} & 0 & a_{44} \end{bmatrix} * \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 \\ b_{21} \\ 0 \\ b_{41} \end{bmatrix} * [\delta] \tag{3.14}$$

$$y_2 = \begin{bmatrix} y \\ \psi \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} * [\delta] \tag{3.15}$$

Where,

$$a_{22} = -\frac{2 * C_f + 2 * C_r}{m * v_x}$$

$$a_{24} = -v_x \ - \frac{2 * C_f * l_f \ - 2 * C_r * l_r}{m v_x}$$

$$a_{42} = -\frac{2 * l_f * C_f \ - 2 * l_r * C_r}{I_z v_x}$$

$$a_{44} = -\frac{2 l_f^2 * C_f + 2 l_r^2 * C_r}{I_z v_x}$$

$$b_{21} = \frac{2 C_f}{m}$$

$$b_{41} = \frac{2 l_f C_f}{I_z}$$

The longitudinal and lateral control models in equations (3.9), (3.10), (3.11) & (3.12) are combined to one state-space model to use as a single prediction model in Model Predictive Controller for vehicle control maneuver. This prediction model has states $x = \begin{bmatrix} e_s & \dot{x}_{ego} & \ddot{x}_{act} & y & \dot{y} & \psi & \dot{\psi} \end{bmatrix}^T$,

control input $u = [F_x \ \ \delta]^T$, measured disturbances to the system $w = [F_{loss} \ \ v_{lead}]^T$, and output of the system $Y = \begin{bmatrix} e_s & \dot{x}_{ego} & y & \psi \end{bmatrix}^T$. The combined state-space model is shown in equations (3.16) & (3.17).

$$\frac{\mathrm{d}}{\mathrm{dt}}\begin{bmatrix} e_s \\ \dot{x}_{ego} \\ \ddot{x}_{act} \\ y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} = A * \begin{bmatrix} e_s \\ \dot{x}_{ego} \\ \ddot{x}_{act} \\ y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} + B * \begin{bmatrix} F_x \\ \delta \end{bmatrix} + B_w * \begin{bmatrix} F_{loss} \\ v_{lead} \end{bmatrix} \tag{3.16}$$

$$A = \begin{bmatrix} 0 & -1 & -t_{gap} & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{\tau} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{22} & 0 & a_{24} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & a_{42} & 0 & a_{44} \end{bmatrix}; B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{1}{\tau m} & 0 \\ 0 & 0 \\ 0 & b_{21} \\ 0 & 0 \\ 0 & b_{41} \end{bmatrix}; B_w = \begin{bmatrix} \frac{t_{gap}}{m} & 1 \\ -\frac{1}{m} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$Y = \begin{bmatrix} e_s \\ \dot{x}_{ego} \\ y \\ \psi \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} e_s \\ \dot{x}_{ego} \\ \ddot{x}_{act} \\ y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} F_x \\ \delta \end{bmatrix} \tag{3.17}$$

3.3 MPC Formulation

Figure 3-2 shows an overview of the designed model predictive control system for the motion control of autonomous vehicles. MPC takes reference e_s (which is 0), commanded velocity, reference lateral position and reference yaw angle as input. These values are determined by the sensor and the planner. Disturbances to the vehicle (like loss of force due to aerodynamic drag and rolling friction) are also sent to MPC by the sensor for better estimation of system states. MPC calculates the control inputs to the vehicle: traction force and steering angle. Vehicle system acts according to the control inputs and feedbacks the system outputs (e_s, ego vehicle velocity, current ego vehicle lateral position and current yaw angle of the ego vehicle) to MPC.

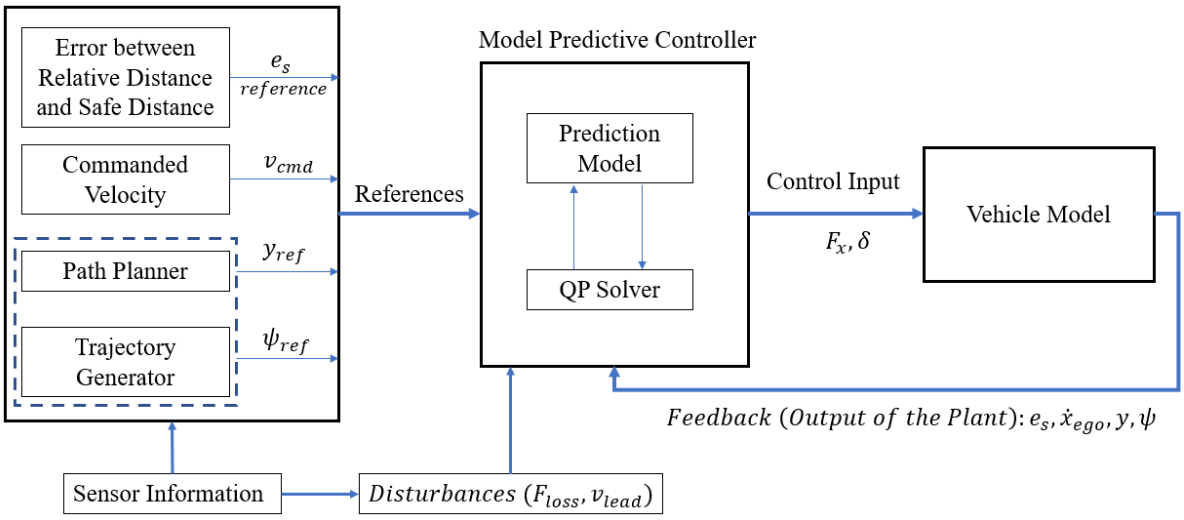

Figure 3-2: Model predictive control system

The lateral reference y_{ref} is the output of the trajectory generator. The reference yaw angle is calculated using the generated trajectory as shown in Table 3. The value of calculated safe distance D_{safe} is not used anywhere in the algorithm, but the equation is shown below in Table 3 because it is used in defining the prediction model.

Table 3: Safe Distance, e_s and Reference Yaw Angle Calculation

$t_{gap} \rightarrow 3$ (Time gap between the vehicles)

$D_{default} \rightarrow 5$ (default standstill spacing)

$egoVelX \rightarrow$ Ego vehicle velocity

$path_{XY} \rightarrow$ The array of generated trajectory waypoints in the world coordinates

$D_{safe} \rightarrow$ Safe distance between ego vehicle and obstacle

$D_{rel} \rightarrow$ Relative distance between ego vehicle and lead vehicle

$\psi_{ref} \rightarrow$ Reference yaw angles of ego vehicle

$e_s \rightarrow$ Error between relative distance and safe distance

$N_y \rightarrow$ length of prediction horizon (less than number of waypoints in $path_{XY}$)

$i \rightarrow an\ integer, i \in [1, N_y]$

$x_i \rightarrow$ X-coordinate of i^{th} waypoint in the array $path_{XY}$

$y_i \rightarrow$ Y-coordinate of i^{th} waypoint in the array $path_{XY}$

While advancing scenario

for every time step 0.1 seconds

$$D_{safe} \rightarrow D_{default} + (t_{gap} * egoVelX)$$

$$\boldsymbol{for}\ i = 1{:}\ N_y$$

$$[\psi_{ref}] \rightarrow atan\frac{(y_{i+1} - y_i)}{(x_{i+1} - x_i)}$$

$$\boldsymbol{end}$$

$$D_{rel} \rightarrow RelDist$$

$$e_s \rightarrow D_{rel} - D_{safe}$$

end

end

return D_{safe}

return ψ_{ref}

A zero-order hold method is used to convert the continuous-time prediction model (equation 3.16 and equation 3.17) to a discrete-time model via exact discretization with sample time Ts. The discrete-time model can be represented as equation (3.18).

$$x(k+1) = A_1 x(k) + B_1 u(k) + B_2\, w(k) \tag{3.18}$$
$$y(k) = C_1 x(k) + D_1 u(k)$$

Where signals are considered at the sampling times: $t = k * Ts$. And $k \,\epsilon\, N =$ discrete time steps.

MPC estimates the controller states and predicts the future plant outputs using the plant model and disturbances. The predicted plant outputs are used by the controller to solve an optimization problem to minimize a cost function (Quadratic Criterion) and determine the control inputs: tractive force (F_x) and steering angle (δ). The cost function is defined in the equation (3.19).

$$J = \sum_{n=1}^{N_y} \left[Y(k+n|k) - Y_{ref}(k+n)\right]^T Q \left[Y(k+n|k) - Y_{ref}(k+n)\right]$$
$$+ \sum_{n=0}^{N_u-1} \delta u^T(k+n)\, R\, \delta u(k+n) \tag{3.19}$$

Where,

$\mathrm{Y}(k) = Output\ at\ timestep\ (k+n),\ estimated\ at\ time\ step\ k.$

$Y_{ref}(k+n) =$ Reference of the measured outputs.

$\delta u =$ Change of control inputs.

$Q =$ Weight matrix of objectives = $diag\left\{q_{e_s},\, q_{\dot{x}_{ego}},\, q_y,\, q_\psi\right\}$

$R =$ Weight matrix for change of control inputs = $diag\ \{q_{F_x},\ q_\delta\}$

$N_y =$ Prediction horizon length

$N_u =$ Control horizon length

The constraints of the optimization problem are shown in equation (3.17).

$$e_s \geq 0$$

$$F_{x_{min}} \leq F_x \leq F_{x_{max}}$$

$$\Delta F_{x_{min}} \leq \Delta F_{x_k} \leq \Delta F_{x_{max}} \tag{3.20}$$

$$\delta_{min} \leq \delta \leq \delta_{max}$$

$$\Delta\delta_{min} \leq \Delta\delta_k \leq \Delta\delta_{max}$$

The constraints for the plant outputs and control inputs are determined based on the vehicle dynamics. The limits for rate of change of control inputs are determined in a way so that passenger discomfort can be avoided.

In the next chapter the motion planning and control algorithms are validated with MATLAB Driving Scenario Designer where the vehicle performance can be visualized.

4 Model-in-the-Loop Validation for the Path Planning and Control Strategy

To validate the path planning and control algorithms, two scenarios have been built using MATLAB Driving Scenario Designer and exported to MATLAB as a function. Upon running the MATLAB script, we can validate the path planning and control strategy that have been applied. The simulation results are presented in this chapter.

4.1 Simulation Setup

Two scenarios have been created using MATLAB Driving Scenario Designer. In the first scenario, there is an ego vehicle and in front of it there is a moving lead vehicle, which changes its lane to the ego vehicle's lane. The lead vehicle has a definite set of waypoints to follow which is known solely to the lead vehicle to give it a movement while the script is running and not known to the ego vehicle. The scenario is set up in a way that the random movement of the lead vehicle is a disturbance to the ego vehicle and the ego vehicle is controlled dynamically based on the change of the lead vehicle position. In the second scenario, there is an ego vehicle and there is a moving lead vehicle in the same lane as of the ego vehicle. The sole purpose of the algorithm implemented here is to avoid the dynamic obstacle by planning a different path along its way and apply control maneuver accordingly. The scenario is shown in Figure 4-1.

The 3D view in Figure 4-1 is achieved by Unreal Engine. However, the validation part will only use the box view (Figure 4-2) and a 2D top view. Figure 4-3 is a scenario canvas and the changes in the scenario will be reflected upon implementing the above-mentioned algorithm and as the actors in the scenario (ego vehicle and lead vehicle) start moving.

Figure 4-1: 3D Visualization of a Driving Scenario Designed in MATLAB

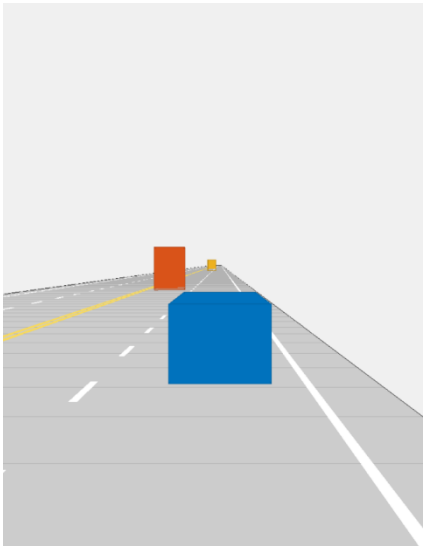

Figure 4-2: 2D Visualization of a Driving Scenario designed in MATLAB.

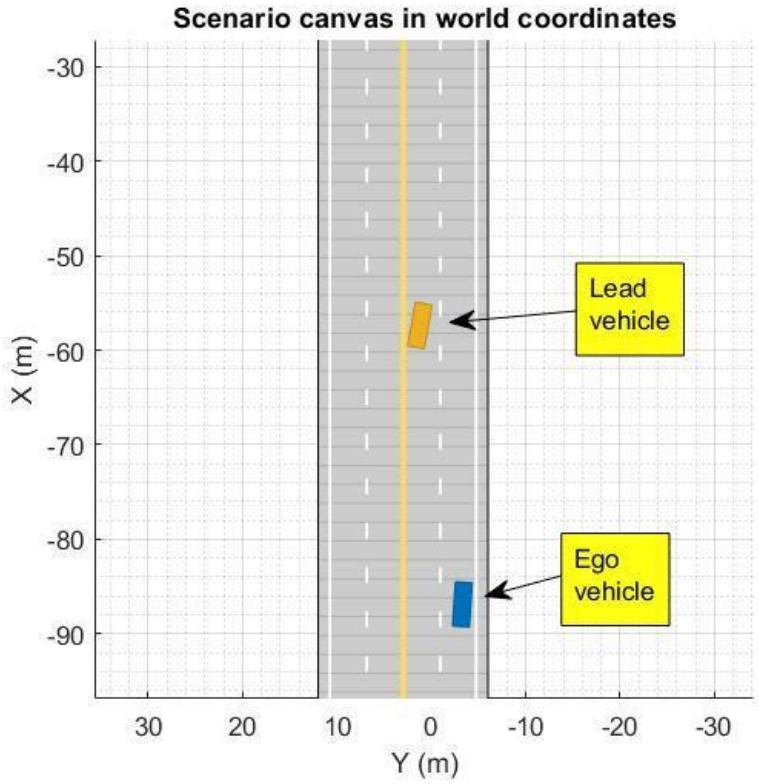

Figure 4-3: 2D Top View of Scenario Canvas with Actors

The algorithms described in the earlier sections will first allow the ego vehicle to search for an optimal path to the goal based on the scenario information available in the map. Then a control algorithm is applied to follow the path accurately and maintain a safe distance from the lead vehicle while controlling ego vehicle's velocity.

4.2 Simulation Results

4.2.1 First Scenario – the obstacle changes lane

The goal position of the ego vehicle is chosen by the user from the scenario environment in world coordinates (XY coordinates), later the goal position is converted to grid coordinates (IJ

coordinates) so that D* path planner can use it for path planning. When the vehicle will first start moving, an optimal path will first be generated from the current ego vehicle position to the goal position based on the ego map information. This also can be considered as route planning as the ego vehicle was not provided with any predefined set of waypoints to follow to reach the goal position in the beginning, and it generates its shortest path to the goal at the start point. As the sensor (monocular camera) configuration does not allow the ego vehicle to detect an obstacle further than a certain distance, it still did not find an obstacle in front of it (Figure 4-4 and Figure 4-5).

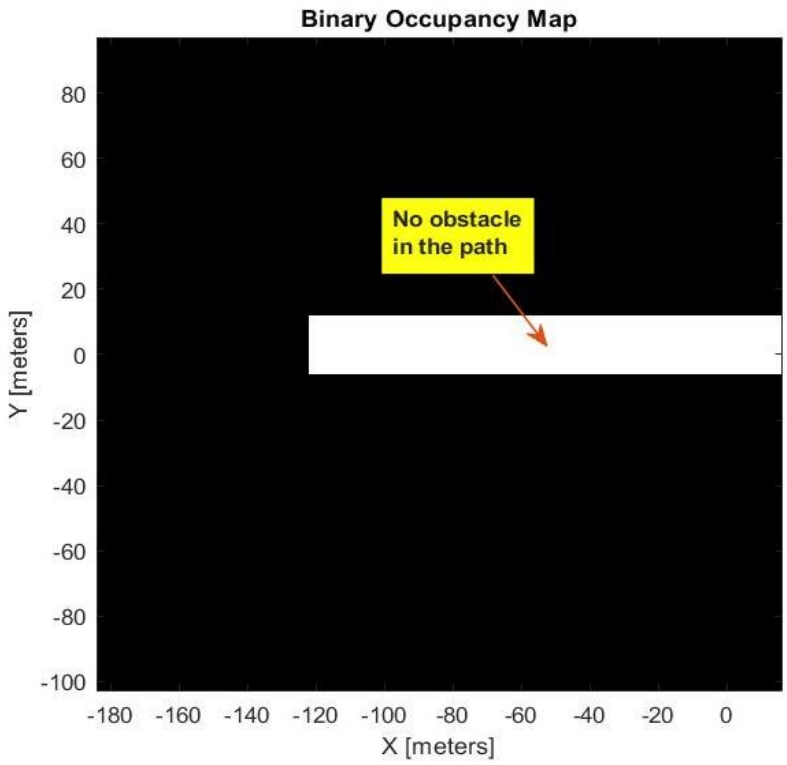

Figure 4-4: Binary occupancy map when no obstacle is detected by camera

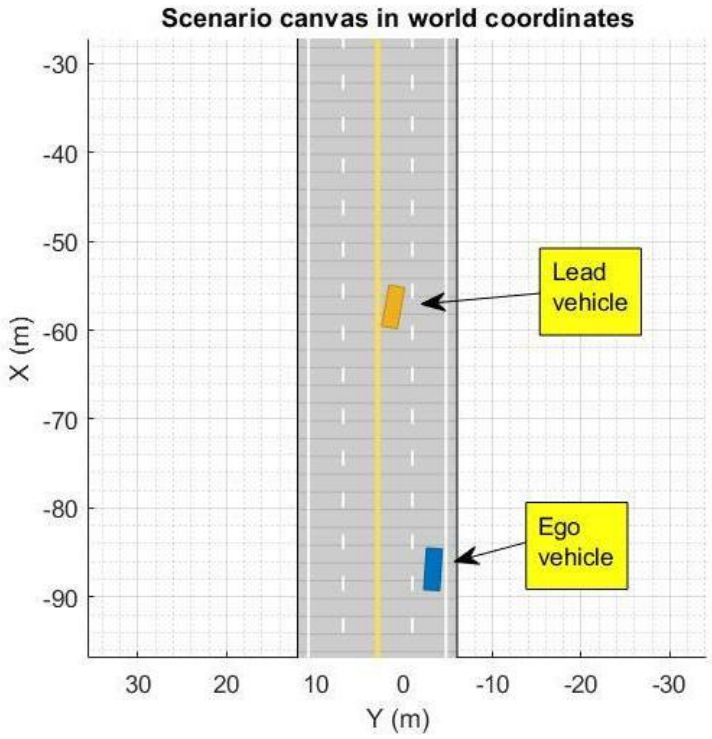

Figure 4-5: Scenario canvas with both the ego vehicle and lead vehicle moving

While moving in the scenario, if any static or dynamic obstacle is detected by the ego vehicle, the planner plans optimal path accordingly to avoid collision and reach the goal.

Now, as the ego vehicle starts moving forward in the scenario, the controller takes action to control the ego vehicle velocity. In Figure 4-6, the lead vehicle starts moving into the ego vehicle lane, but it's not yet detected by the ego vehicle.

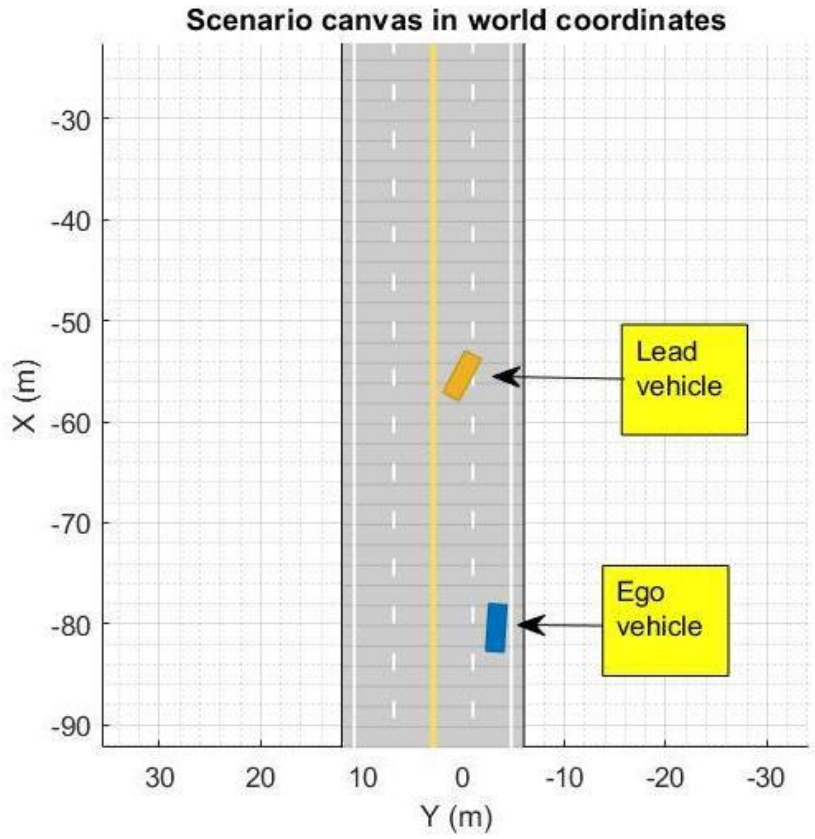

Figure 4-6: Lead vehicle starts moving into the ego vehicle lane

Next, in Figure 4-7, the ego vehicle detects the lead vehicle in front of it and the planner plans a trajectory to pass the lead vehicle after updating and modifying the cost in the binary occupancy map by setting the cost of free space to 0 (zero) and occupied space to 1 (one).

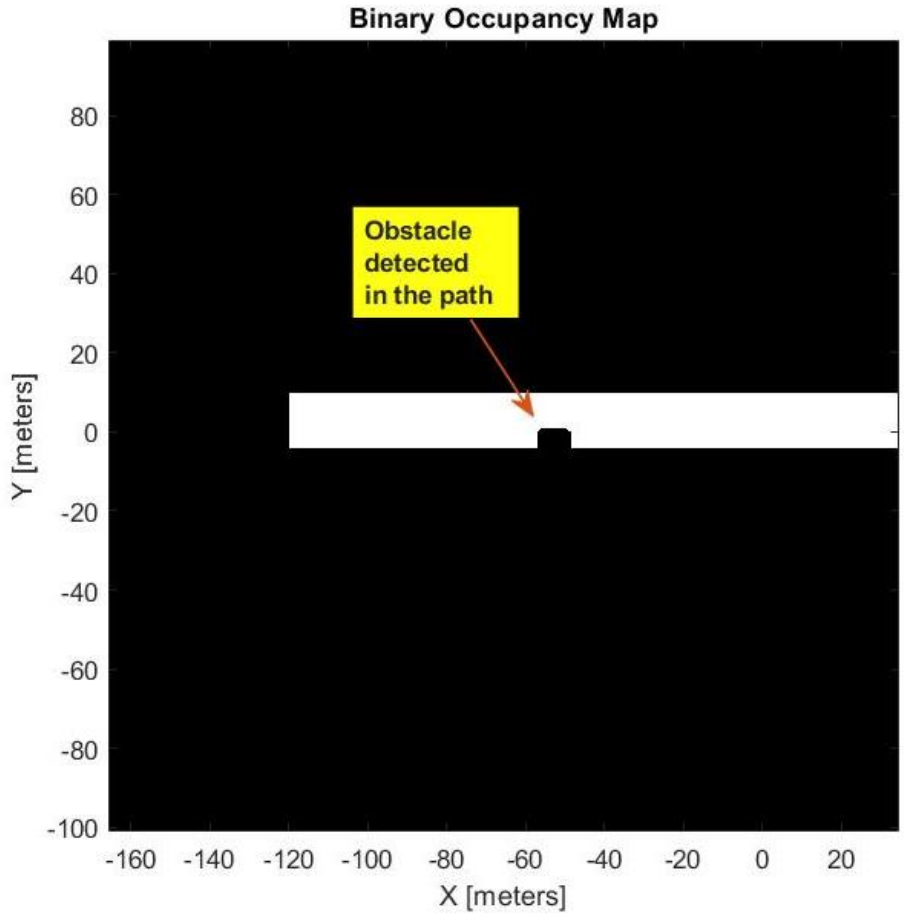

Figure 4-7: Updated binary occupancy map after an obstacle is detected in the path by the camera sensor

Now, as the lead vehicle (obstacle) is dynamically changing its place, the ego vehicle plans and follows a new trajectory to avoid the obstacle every time it gets the updated binary occupancy map with most recent obstacle position (Figure 4-8, Figure 4-9 and Figure 4-10).

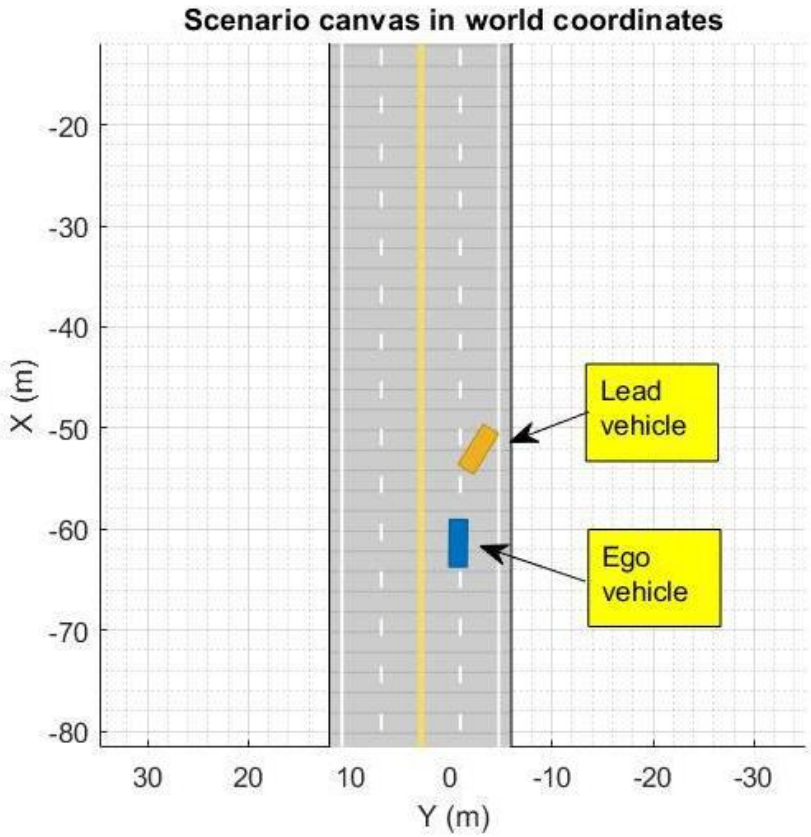

Figure 4-8: Ego vehicle starts moving to the adjacent left lane for obstacle avoidance

The orientation of the lead vehicle in the binary occupancy map is different from how it appears in the scenario canvas. This feature of MATLAB could not be avoided in this project but increasing the obstacle width in the occupancy grid map helps address this situation and plan optimal path without colliding with the lead vehicle.

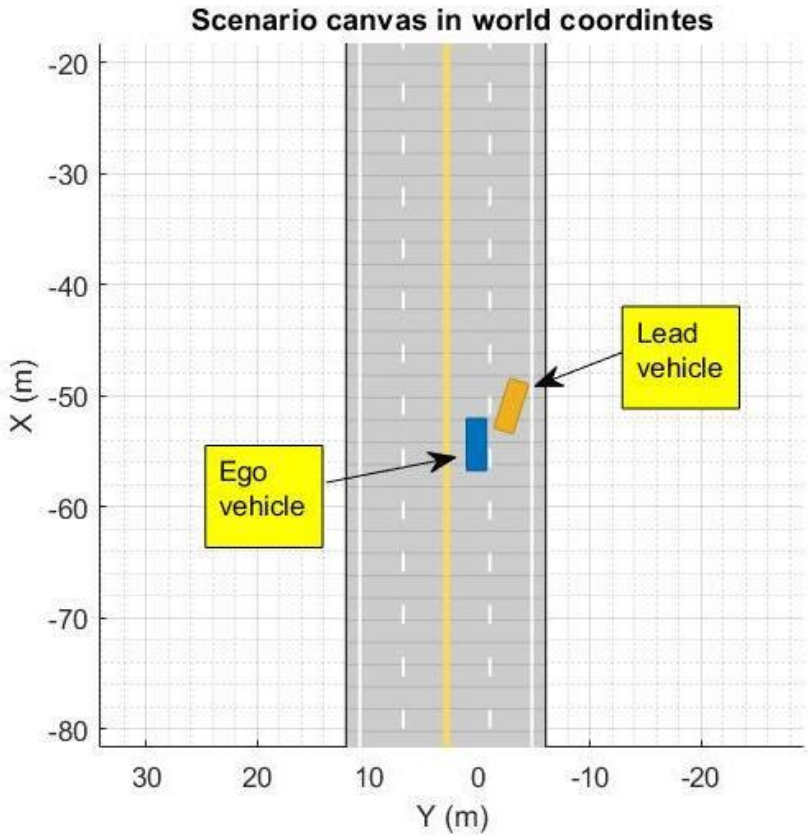

Figure 4-9: Ego vehicle is performing dynamic obstacle avoidance maneuver

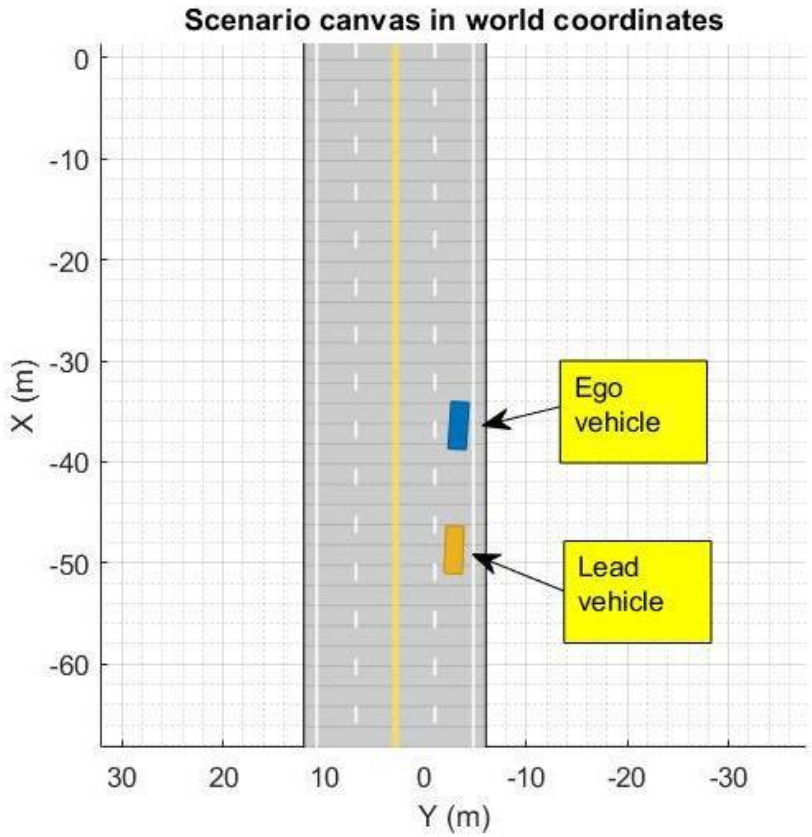

Figure 4-10: Ego vehicle successfully avoids collision and overtakes the lead vehicle

In Figure 4-10, the ego vehicle successfully avoids collision with the dynamic lead vehicle and overtakes with the help of the planning algorithm. The path that the ego vehicle ultimately followed to avoid moving obstacle and reach its goal is shown in Figure 4-11.

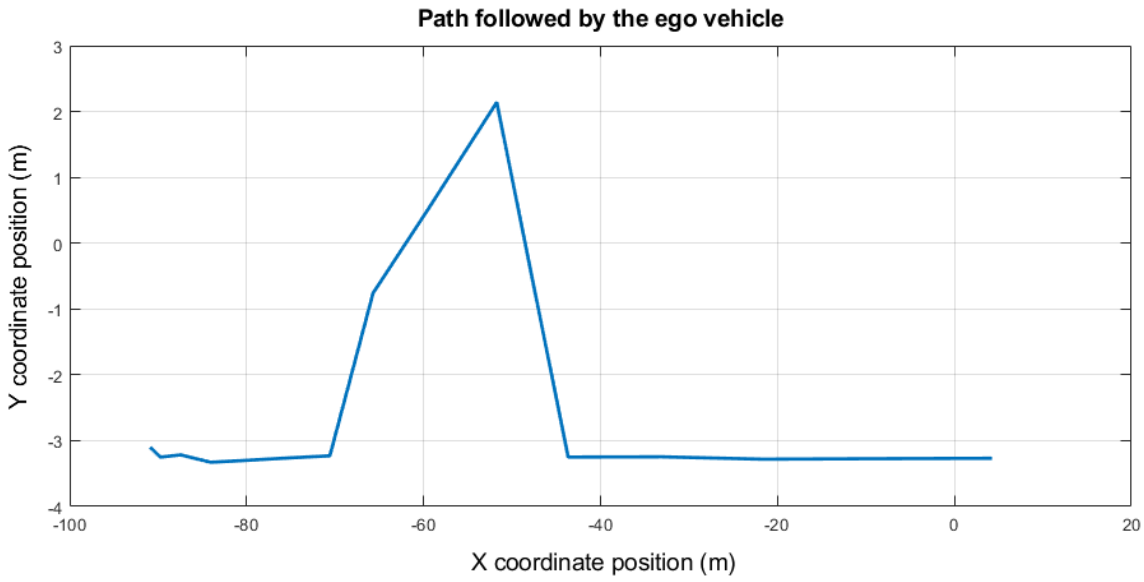

Figure 4-11: Ultimate path followed by the ego vehicle

In Figure 4-11, the units of the axes are in meters. The coordinates (X and Y) here match the scenario coordinates created in Driving Scenario Designer and each point in the plot corresponds to the ego vehicle position in the scenario canvas (world coordinates).

MPC tracks the commanded ego vehicle velocity when the relative distance between the ego vehicle and the lead vehicle is greater than the safe distance. When the relative distance is too close to the safe distance, the MPC controls the headway. When the obstacle is detected by the ego vehicle and the ego vehicle wants to pass the lead vehicle, the ego vehicle performs obstacle avoidance maneuver and MPC tracks the desired lateral position.

The safe distance between the ego vehicle and the lead vehicle changes as both the vehicles start moving in the scenario. The time gap while calculating the safe distance is chosen according to the 3 second rule.

In Figure 4-12, change of ego vehicle velocity is shown over the time. The MPC tracks the ego vehicle velocity to the commanded velocity until 2 seconds as no obstacle was detected. At the time (at 2 seconds) when the ego vehicle detects the lead vehicle in the same lane, the relative distance between the lead vehicle and the ego vehicle is less than the calculated safe distance. So, at this point, the MPC tries to reduce the ego vehicle velocity by reducing the tractive force applied to the vehicle and tries to maintain the safe distance from lead vehicle. Once the ego vehicle moves to the adjacent lane to avoid the obstacle at 7th second, MPC again starts increasing the ego vehicle velocity to track the commanded velocity.

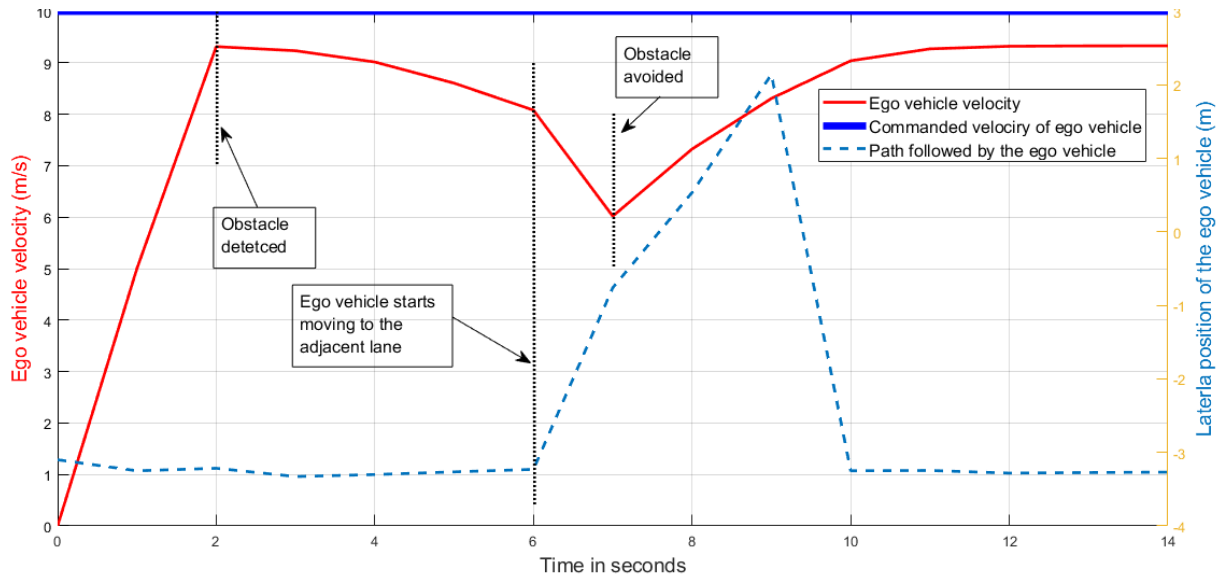

Figure 4-12: MPC tracks the ego vehicle velocity according to safe distance and obstacle position

Figure 4-13 shows the percentage of the traction force applied by the MPC during the entire maneuver of the ego vehicle.

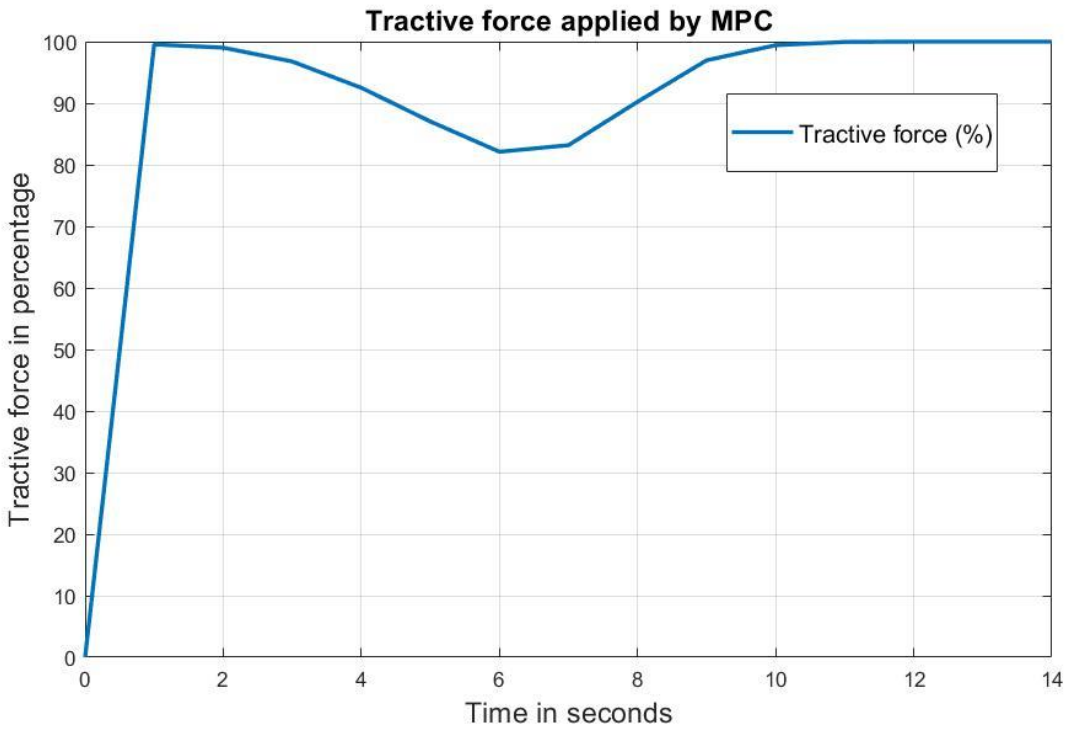

Figure 4-13: Change of traction force over the time for vehicle longitudinal control

In Figure 4-13, as the MPC is trying to track the ego vehicle velocity to the commanded velocity until 2 seconds, the tractive force applied to the vehicle increases to reach the commanded velocity. At 2 seconds, when the obstacle is detected and the relative distance is less than the safe distance, the MPC starts reducing the velocity by reducing the tractive force to maintain the safe distance from the lead vehicle. As the ego vehicle avoids the obstacle at 7 seconds, the MPC again starts increasing the ego vehicle velocity by increasing the tractive force from 7^{th} second to track the commanded velocity.

As the planner plans the path for the ego vehicle, the desired position needs to be followed by the vehicle to avoid the obstacle. To track the desired path, the controller applies a steering angle to the steering wheel. Figure 4-14 shows the steering angle applied by the controller over the time.

After the ego vehicle detects the obstacle, at 6th second the ego vehicle starts moving to the adjacent left lane to follow the desired path and avoid the obstacle, that is why the MPC starts increasing the steering angle at 6th second for the ego vehicle to steer left. When the ego vehicle starts moving back to its previous lane according to the planned path, the MPC starts reducing the steering angle at 9th second for the vehicle to steer to the right. After returning to its previous lane, the ego vehicle moves straight, and steering angle becomes zero.

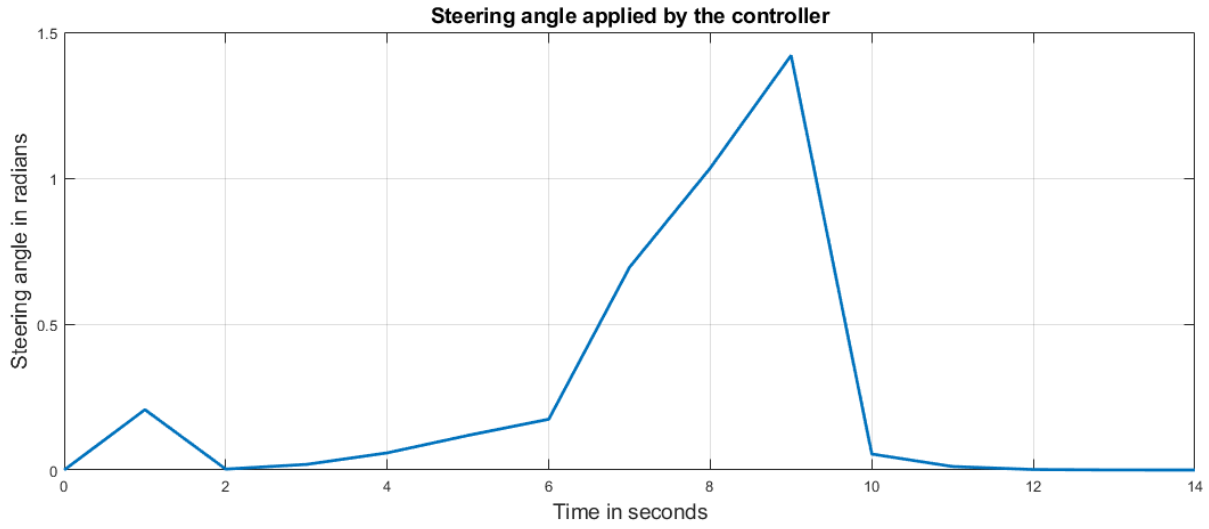

Figure 4-14: Change of steering angle over the time for vehicle lateral control

The controller helps track the desired lateral position determined from the path planner with the steering angle command. The tracked lateral path has been plotted in Figure 4-15 over the time. The Y axis in Figure 4-15 represents the lateral position of the ego vehicle in the world coordinate.

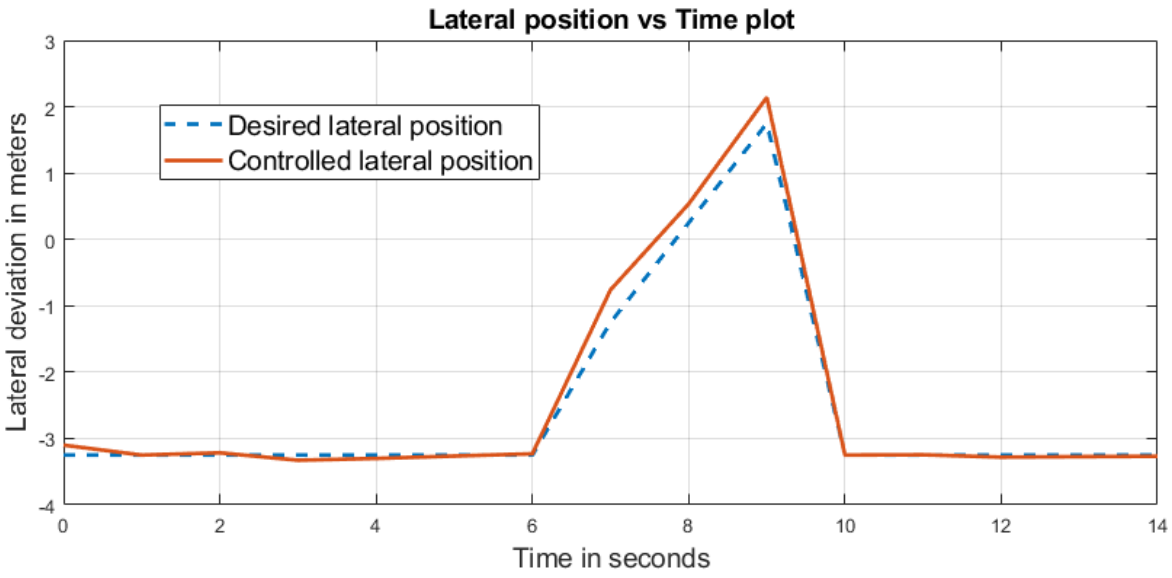

Figure 4-15: Tracked lateral position of ego vehicle by MPC

4.2.2 Second Scenario – Obstacle moving in the same lane

In the second scenario, there is an ego vehicle, and in front of it there is one lead vehicle which is moving with a constant speed in the same lane as the ego vehicle. Figure 4-16, Figure 4-17 and Figure 4-18 show that the ego vehicle autonomously follows the dynamically planned optimal path to avoid the lead vehicle and ultimately overtakes it without colliding. The ego vehicle in the scenario first tries to track the commanded velocity until it detects a moving obstacle. Once the obstacle is detected, the ego vehicle moves to the adjacent lane to execute obstacle avoidance maneuver. After the ego vehicle avoids the obstacle the velocity of the ego vehicle again increases.

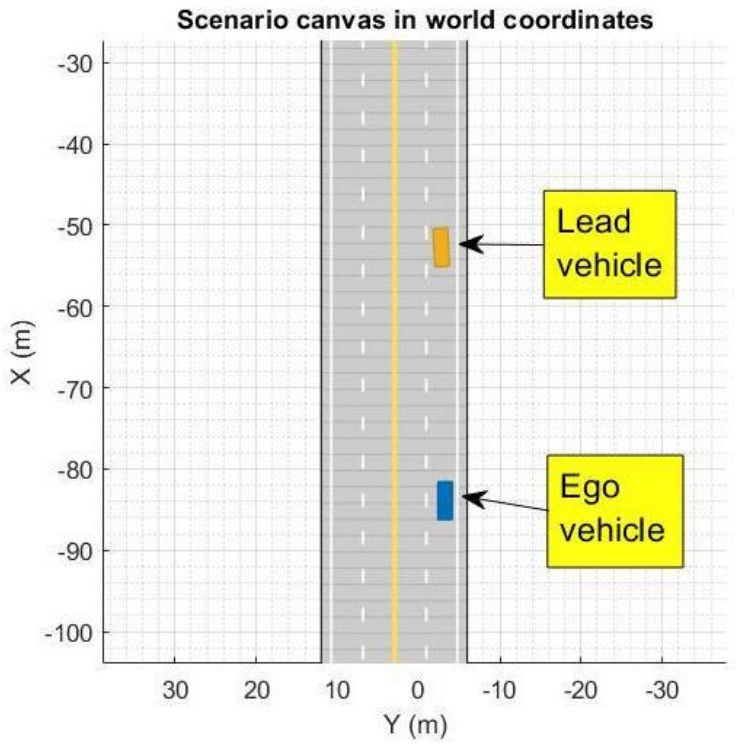

Figure 4-16: Ego vehicle is approaching the lead vehicle

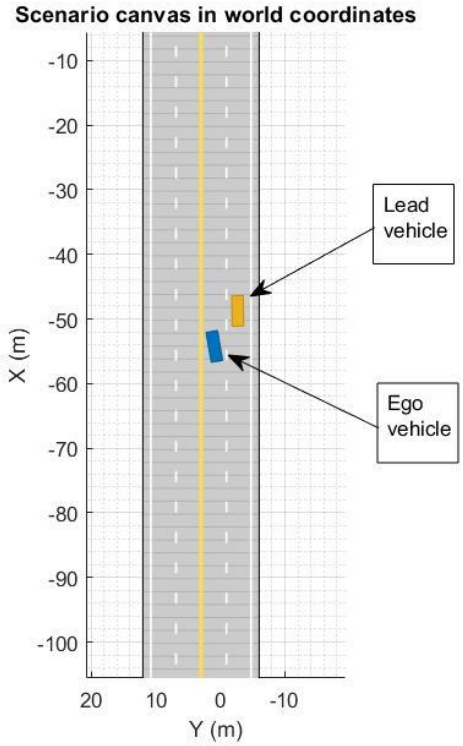

Figure 4-17: Ego vehicle is demonstrating obstacle avoidance

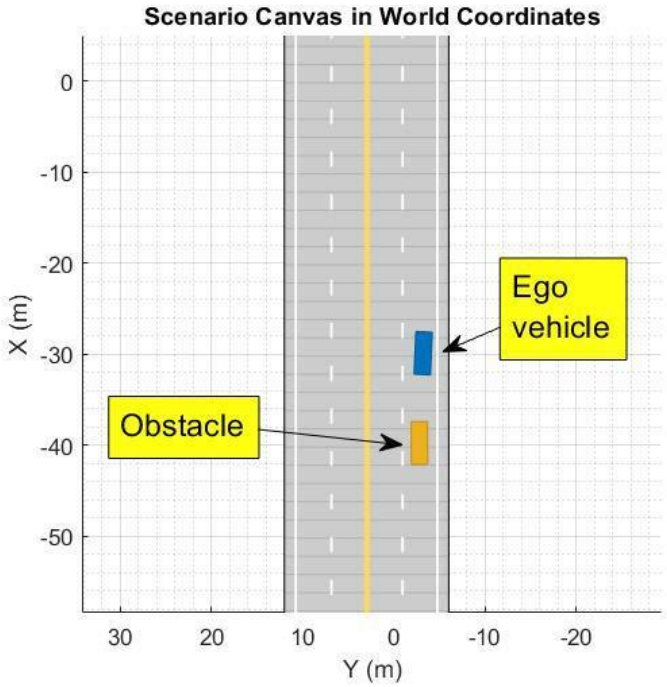

Figure 4-18: Ego vehicle overtakes the lead vehicle

In Figure 4-19, change of ego vehicle velocity is shown over the time. MPC tracks the ego vehicle velocity to the commanded velocity until the ego vehicle detects an obstacle. As the ego vehicle detects an object at 7th second, the ego vehicle starts obstacle avoidance maneuver. After the ego vehicle avoids the obstacle at 9th second, the ego vehicle tracks the commanded velocity again.

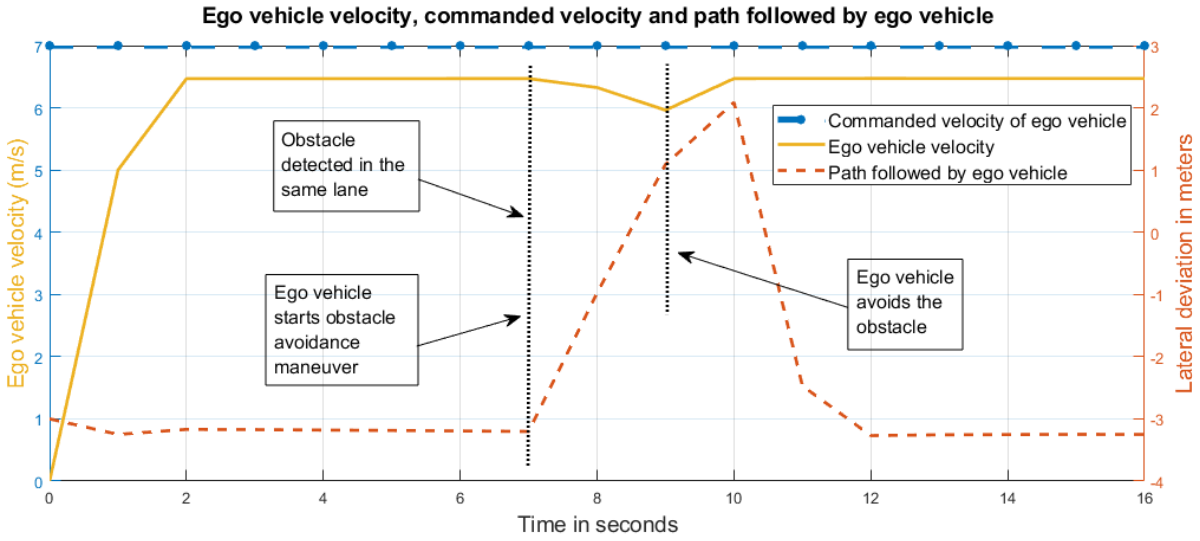

Figure 4-19: Ego vehicle velocity over the simulation time

Figure 4-20 shows the percentage of the traction force applied by the MPC during the entire maneuver of the ego vehicle.

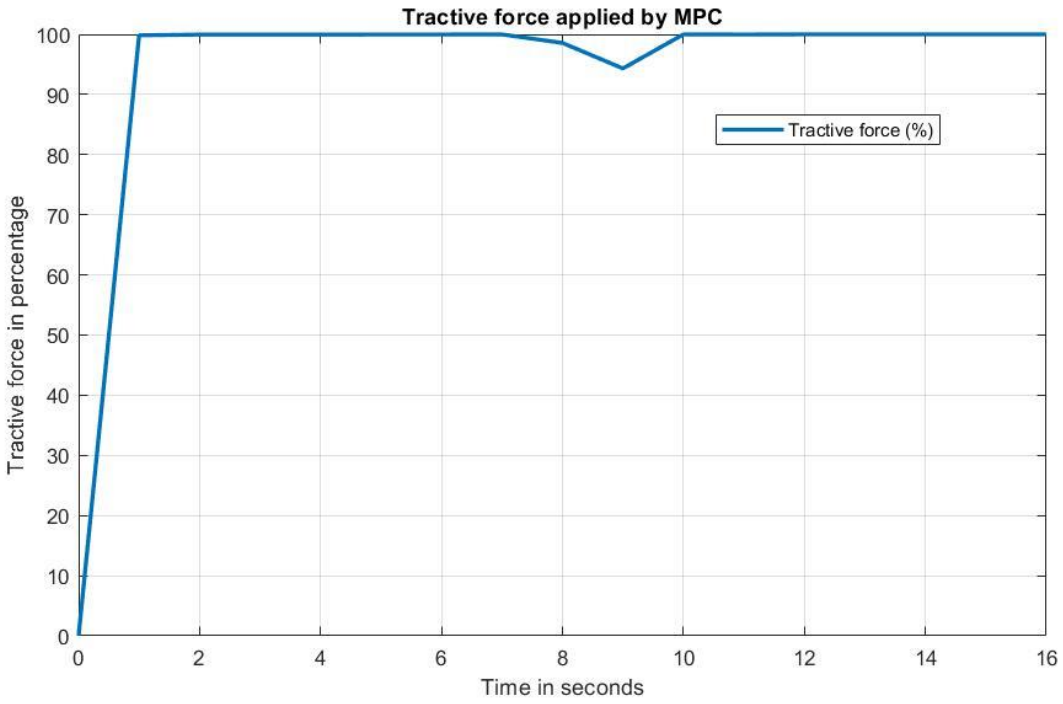

Figure 4-20: Change of traction force over the time for vehicle longitudinal control

In Figure 4-20, as the MPC is trying to track the ego vehicle velocity to the commanded velocity, the tractive force applied to the vehicle increases until 7 seconds. At 7 seconds, the ego vehicle detects an obstacle and starts obstacle avoiding maneuver. As the ego vehicle avoids the obstacle at 9 seconds after moving to the adjacent lane, the MPC again starts increasing the ego vehicle velocity by increasing the tractive force from 9th second to track the commanded velocity.

As the planner plans the path for the ego vehicle, the desired position needs to be followed by the vehicle to avoid the obstacle. To track the desired path, the controller applies a steering angle to the steering wheel. Figure 4-21 shows the steering angle applied by the controller over the time. After the ego vehicle detects the obstacle, the ego vehicle starts moving to the adjacent left lane at 7th second to follow the desired path and avoid the obstacle, that is why the MPC starts increasing

the steering angle at 7th second for the ego vehicle to steer left. After completing obstacle avoidance, the ego vehicle moves straight, and the steering angle becomes zero.

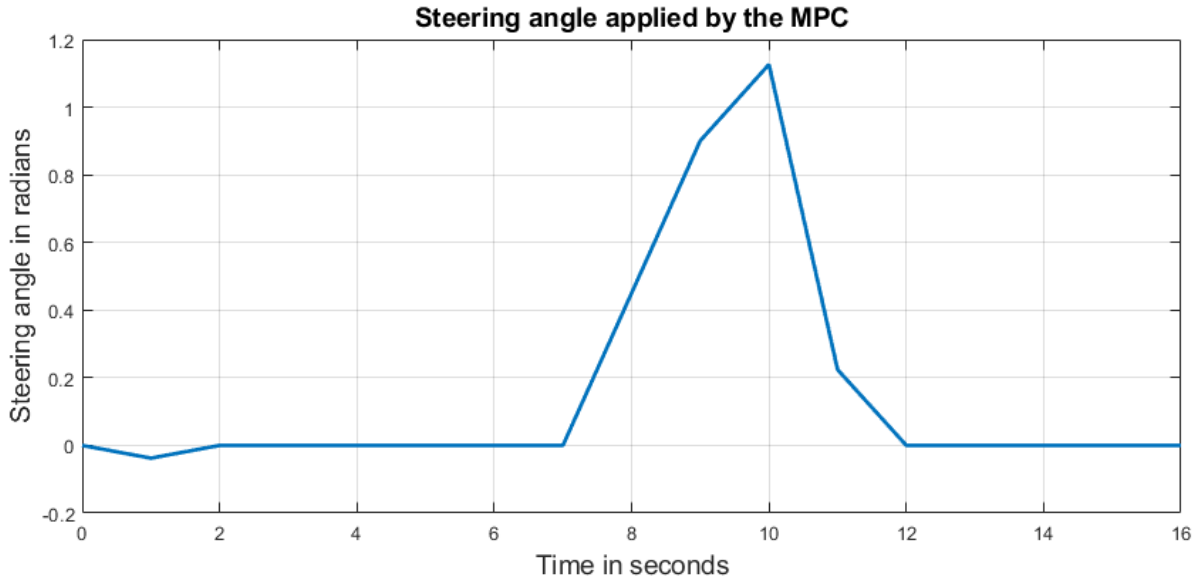

Figure 4-21: Change of steering angle over the time for vehicle lateral control

The controller helps track the desired lateral position determined from the path planner with the steering angle command. The tracked lateral path has been plotted in Figure 4-22 over the time. The Y axis in Figure 4-22 represents the lateral position of the ego vehicle in the world coordinate.

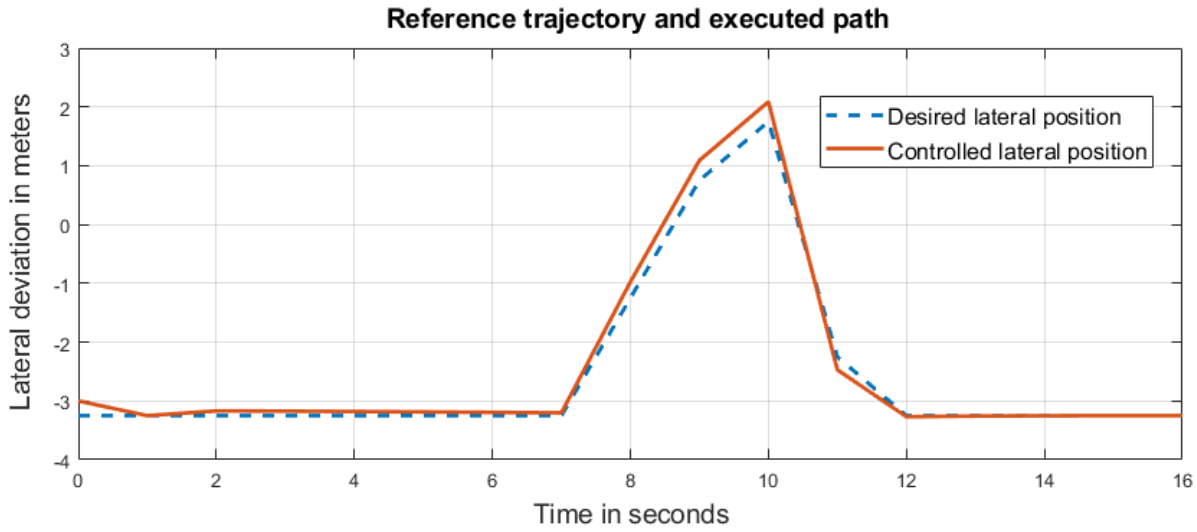

Figure 4-22: Comparison of reference trajectory and executed ego vehicle trajectory for obstacle avoidance maneuver

5. Conclusion and Future Scope

The focus of the research is mainly concentrated on the implementation of a novel, computationally fast and memory efficient optimal path planning and control algorithm for self-driving cars in a dynamic environment.

5.1 Conclusion

The problem of motion planning and control in a dynamic environment is addressed with D* path planner integrated with Cubic B-Spline trajectory generator, and a model predictive controller for combined longitudinal and lateral vehicle control. A constant time gap approach is used to define the safe distance between the ego and lead vehicles. The MPC controller tracks the relative distance between the ego and lead vehicle to the safe distance, and also executes the desired path for obstacle avoidance. Furthermore, a model-in-the-loop method is used to validate the motion planning and control algorithms using MATLAB Driving Scenario Designer where the ego vehicle successfully plans and executes a path from its current position to the goal position while avoiding a dynamic obstacle in a simulated traffic scenario. Longitudinal and lateral control of the vehicle during the motion execution was also demonstrated with the working of MPC.

5.2 Future Scope

Further development on the implemented motion planning and control algorithm can be achieved by testing the algorithm in actual vehicles and hardware-in-the-loop method. The prediction model can be improved by analyzing the vehicle data therefore improving controller performance. Controller performance can also be improved by using nonlinear MPC where the prediction model

includes the nonlinearities associated with tire and steering system dynamics. The monocular camera sensor used from the MATLAB Driving Scenario Designer has limitations and assumes less sensor noise, whereas, in reality, the sensor noise is a big problem to consider. Using more robust image processing, localization and sensor fusion algorithms can improve the sensor data quality by making it more accurate. The control algorithm can also be tested real-time in actual vehicles in various other road scenarios with different weather, light and traffic conditions.

6. Reference List

1. Christos Katrakazas, Mohammed Quddus, Wen-Hua Chen, Lipika Deka, Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions, Transportation Research Part C: Emerging Technologies, Volume 60, 2015, Pages 416-442, ISSN 0968-090X, https://doi.org/10.1016/j.trc.2015.09.011.

2. Paden, Brian & Čáp, Michal & Yong, Sze Zheng & Yershov, Dmitry & Frazzoli, Emilio. (2016). A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles. IEEE Transactions on Intelligent Vehicles. 1. 10.1109/TIV.2016.2578706.

3. "Improper Driving and Road Rage," Injury Facts, 29-Mar-2021. [Online]. Available: https://injuryfacts.nsc.org/motor-vehicle/motor-vehicle-safety-issues/improper-driving-and-road-rage/. [Accessed: 23-Apr-2021].

4. M. I. T. T. R. Insights, "Self-driving cars take the wheel," MIT Technology Review, 02-Apr-2020. [Online]. Available: https://www.technologyreview.com/2019/02/15/137381/self-driving-cars-take-the-wheel/. [Accessed: 23-Apr-2021].

5. Gonzalez Bautista, David & Pérez, Joshué & Milanes, Vicente & Nashashibi, Fawzi. (2015). A Review of Motion Planning Techniques for Automated Vehicles. IEEE Transactions on Intelligent Transportation Systems. 1-11. 10.1109/TITS.2015.2498841.

6. "SAE J3016 automated-driving graphic," SAE International, 15-May-2020. [Online]. Available: https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving graphic#:~:text=The%20J3016%20standard%20defines%20six,%2Dvehicle%20(AV)%20capabilities. [Accessed: 23-Apr-2021].

7. Urmson, Chris & Baker, Christopher & Dolan, John & Rybski, Paul & Salesky, Bryan & Whittaker, William & Ferguson, Dave & Darms, Michael. (2009). Autonomous Driving in Traffic: Boss and the Urban Challenge. AI Magazine. 30. 17-28. 10.1609/aimag.v30i2.2238.

8. E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische mathematik, vol. 1, pp. 269–271, 1959.

9. N. J. Nilsson, "A mobile automaton: An application of artificial intelligence techniques," tech. rep., DTIC Document, 1969.

10. A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 156–165, Society for Industrial and Applied Mathematics, 2005.

11. R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," Transportation Science, vol. 46, pp. 388–404, 2012.

12. F. Havlak and M. Campbell, “Discrete and continuous, probabilistic anticipation for autonomous robots in urban environments,” Transactions on Robotics, vol. 30, pp. 461–474, 2014.

13. Q. Tran and J. Firl, “Modelling of traffic situations at urban intersections with probabilistic non-parametric regression,” in Intelligent Vehicles Symposium (IV), 2013 IEEE, pp. 334–339, IEEE, 2013.

14. A. C. Madrigal, “The trick that makes Google’s self-driving cars work,” The Atlantic, 2015. http://www:theatlantic:com/technology/archive/2014/05/all-the-world-a-track-the-trick-that-makes-googlesself-driving-cars-work/370871/.

15. R. Verma and D. D. Vecchio, “Semiautonomous multivehicle safety,” Robotics & Automation Magazine, IEEE, vol. 18, pp. 44–54, 2011.

16. S. Z. Yong, M. Zhu, and E. Frazzoli, “Generalized innovation and inference algorithms for hidden mode switched linear stochastic systems with unknown inputs,” in Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on, pp. 3388–3394, IEEE, 2014.

17. V. Kunchev, L. Jain, V. Ivancevic, and A. Finn, “Path planning and obstacle avoidance for autonomous mobile robots: A review,” in Knowledge-Based Intelligent Information and Engineering Systems. Springer, 2006, pp. 537–544.

18. Y. K. Hwang and N. Ahuja, “Gross motion planninga survey,” ACM Computing Surveys (CSUR), vol. 24, no. 3, pp. 219–291, 1992.

19. Gonzalez Bautista, David & Pérez, Joshué & Milanes, Vicente & Nashashibi, Fawzi. (2015). A Review of Motion Planning Techniques for Automated Vehicles. IEEE Transactions on Intelligent Transportation Systems. 1-11. 10.1109/TITS.2015.2498841.

20. Bohren, J., Foote, T., Keller, J., Kushleyev, A., Lee, D., Stewart, A., Vernaza, P., Derenick, J., Spletzer, J. and Satterfield, B. (2008), Little Ben: The Ben Franklin Racing Team's entry in the 2007 DARPA Urban Challenge. J. Field Robotics, 25: 598-614.

21. A. Bacha, C. Bauman, R. Faruque, M. Fleming, C. Terwelp, C. Reinholtz, D. Hong, A. Wicks, T. Alberi, D. Anderson et al., “Odin: Team victortango’s entry in the darpa urban challenge,” Journal of Field Robotics, vol. 25, no. 8, pp. 467–492, 2008.

22. M. Likhachev and D. Ferguson, “Planning long dynamically feasible maneuvers for autonomous vehicles,” The International Journal of Robotics Research, vol. 28, no. 8, pp. 933–945, 2009.

23. A. Stentz, “Optimal and efficient path planning for partially-known environments,” in Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on. IEEE, 1994, pp. 3310–3317.

24. M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, “Anytime search in dynamic graphs,” Artificial Intelligence, vol. 172, no. 14, pp. 1613–1643, 2008.

25. D. Ferguson and A. Stentz, “Using interpolation to improve path planning: The field d* algorithm,” Journal of Field Robotics, vol. 23, no. 2, pp. 79–101, 2006.

26. M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke et al., “Junior: The stanford entry in the urban challenge,” Journal of field Robotics, vol. 25, no. 9, pp. 569–597, 2008.

27. S. Kammel, J. Ziegler, B. Pitzer, M. Werling, T. Gindele, D. Jagzent, J. Schr¨oder, M. Thuy, M. Goebl, F. v. Hundelshausen et al., “Team annieway’s autonomous system for the 2007 darpa urban challenge,” Journal of Field Robotics, vol. 25, no. 9, pp. 615–639, 2008.

28. D. Ferguson, T. M. Howard, and M. Likhachev, “Motion planning in urban environments,” Journal of Field Robotics, vol. 25, no. 11-12, pp. 939–960, 2008.

29. M. Elbanhawi and M. Simic, “Sampling-based robot motion planning: A review,” Access, IEEE, vol. 2, pp. 56–77, 2014.

30. J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman, et al., “A perceptiondriven autonomous urban vehicle,” Journal of Field Robotics, vol. 25, pp. 727–774, 2008.

31. M. Massera, Carlos & Wolf, Denis & Grassi Jr, Valdir & Osorio, Fernando. (2014). Longitudinal and lateral control for autonomous ground vehicles. IEEE Intelligent Vehicles Symposium, Proceedings. 10.1109/IVS.2014.6856431.

32. A. Khodayari, A. Ghaffari, S. Ameli and J. Flahatgar, "A historical review on lateral and longitudinal control of autonomous vehicle motions," 2010 International Conference on Mechanical and Electrical Technology, Singapore, 2010, pp. 421-429, doi: 10.1109/ICMET.2010.5598396.

33. W. Lang, C. Wang, and Y. Chiang, "On the car-following model with fuzzy control," in Proc. 2nd Nat. Conf Fuzzy Theory & Application (Fuzzy'94), pp. 380-385.

34. Rajamani, R. (2006). Vehicle Dynamics and Control. 10.1007/0-387-28823-6.

35. O. Amidi, “Integrated mobile robot control,” Robotics Institute, Pitts-burgh, PA, Tech. Rep. CMU-RI-TR-90-17, May 1990.

36. G. Hoffmann, C. Tomlin, D. Montemerlo, and S. Thrun, “Autonomousautomobile trajectory tracking for off-road driving: Controller design,experimental validation and racing,” inAmerican Control Conference,2007. ACC ’07, 2007, pp. 2296–2301.

37. Sai Rajeev Devaragudi and Bo Chen, “MPC-based control of autonomous vehicles with localized path planning for obstacle avoidance under uncertainties,” 2019 ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications (MESA2019), Anaheim, CA, August 18 – 21, 2019.

38. C. Urmson, J. Anhalt, H. Bae, J. A. D. Bagnell, C. R. Baker , R. E.Bittner, T. Brown, M. N. Clark, M. Darms, D. Demitrish, J. M.
Dolan,D. Duggins, D. Ferguson , T. Galatali, C. M. Geyer, M. Gittleman,S. Harbaugh, M. Hebert, T. Howard, S. Kolski, M. Likhachev, B. Litk-ouhi, A. Kelly, M. McNaughton, N. Miller, J. Nickolaou, K. Peterson,B. Pilnick, R. Rajkumar, P. Rybski, V. Sadekar, B. Salesky, Y.-W. Seo,S. Singh, J. M. Snider, J. C. Struble, A. T. Stentz, M. Taylor , W. R.
L.Whittaker, Z. Wolkowicki, W. Zhang, and J. Ziglar, "Autonomousdriving in urban environments: Boss and the urban challenge,"Journalof Field Robotics Special Issue on the 2007 DARPA Urban Challenge,Part I, vol. 25, no. 8, pp. 425–466, June 2008

39. Create occupancy grid with binary values - MATLAB. [Online]. Available: https://www.mathworks.com/help/nav/ref/binaryoccupancymap.html. [Accessed: 23-Apr-2021].

40. Stentz, Anthony. (2011). The D* Algorithm for Real-Time Planning of Optimal Traverses.

41. "controlPoints," Generate polynomial trajectories using B-splines - MATLAB. [Online]. Available: https://www.mathworks.com/help/robotics/ref/bsplinepolytraj.html?searchHighlight=bsplinepolytraj&s_tid=srchtitle#mw_a67f0807-9b13-4ec8-962f-16d462ae5057. [Accessed: 23-Apr-2021].

42. (2009) Discrete-time MPC for Beginners. In: Model Predictive Control System Design and Implementation Using MATLAB®. Advances in Industrial Control. Springer, London. https://doi.org/10.1007/978-1-84882-331-0_1

43. Camacho E.F., Bordons C. (2007) Model Predictive Controllers. In: Model Predictive control. Advanced Textbooks in Control and Signal Processing. Springer, London. https://doi.org/10.1007/978-0-85729-398-5_2

44. S.Joe Qin, Thomas A. Badgwell, A survey of industrial model predictive control technology, Control Engineering Practice, Volume 11, Issue 7, 2003, Pages 733-764, ISSN 0967-0661, https://doi.org/10.1016/S0967-0661(02)00186-7.

45. Camacho E.F., Bordons C. (2007) Constrained Model Predictive Control. In: Model Predictive control. Advanced Textbooks in Control and Signal Processing. Springer, London. https://doi.org/10.1007/978-0-85729-398-5_7

46. Choset, H. (n.d.). motionplanning/lecture/AppH-astar-dstar_howie. http://www.cs.cmu.edu/~choset.

47. Wikimedia Foundation. (2021, September 23). *B-spline*. Wikipedia. Retrieved October 9, 2021, from https://en.wikipedia.org/wiki/B-spline.

48. *Programmatic scenario authoring*. Programmatic Scenario Authoring - MATLAB & Simulink. (n.d.). Retrieved October 10, 2021, from

https://www.mathworks.com/help/driving/programmatic-scenario-authoring.html?s_tid=CRUX_topnav.

49. Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A. Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. 2005. Principles of Robot Motion : Theory, Algorithms, and Implementations. Intelligent Robotics and Autonomous Agents. Cambridge, Mass: A Bradford Book. https://search-ebscohost-com.services.lib.mtu.edu/login.aspx?direct=true&db=e000xna&AN=126017&site=ehost-live.