# BOSCH-INTER-PROJECT-MCTS-based-autonomous-driving-decision-making

A Monte Carlo Tree Search Approach for Autonomous Driving Decision Making at Intersections

## Abstract

## Introduction

As we teeter on the cusp of a new era in transportation, defined by autonomous driving, we face a multitude of complex problems that challenge our basic assumptions about safety, efficiency, and accessibility. One such issue, which notably escalates in complexity, is the decision-making process for autonomous vehicles at intersections, where multiple parameters such as vehicle speed, trajectory, surrounding traffic behavior, intention predictions, and traffic rule adherence intertwine. This paper seeks to address this intricate issue by harnessing the powerful and heuristic capabilities of the Monte Carlo Tree Search (MCTS) algorithm, a computational tool primarily revered in the realm of game-oriented artificial intelligence. By employing MCTS, we aim to tackle the overlapping and multifaceted decision-making prerequisites vital for autonomous vehicles, particularly in their interaction with intersections.

## Objective

The purpose of our study is twofold: first, to establish and evaluate a decision-making framework for autonomous vehicles based on the Monte Carlo Tree Search (MCTS) algorithm when maneuvering intersections, and second, to optimize three critical dimensions of autonomous driving: safety, efficiency, and comfort. This is achieved by leveraging the unique capabilities of MCTS to simulate diverse action sequences, probe their potential outcomes, and make informed decisions. The balance between these objectives, each of substantial importance and often presenting trade-offs, is achieved through a meticulous process.

## Methods

Initialize:

In the main running function, first we would load the scenario. The scenario basically contains an intersection (with 2/4/6 lanes, ego Vehicle and other actor Vehicles). Then create a waypoint for to be the reference line of the roads, containing the intersection and the destination, after which we get a fernet path (refPath = referencePathFrenet(waypoints)) in terms of the waypoint we selected.

Continually, we give the ego Vehicle its start state, represented by a 1\*6 matrix, startEgoState = [x y theta kappa speed acc]. Then we update the ego Vehicle’s sate, as well as its Fernet State using function global2frenet(refPath, startEgoState).

After finishing the basic settings, we move on to the MCTS algorithm. We use struct to store the properties of each node, and each node has the following properties:

struct(

'state', [x y theta kappa speed acc] , '

time', the current time of node in the MCTS tree(also can be seen as the level of the node),

'children', a matrix containing all index of the children, whose first element is 0.

'visits', the time that the node has been visited

'score', the current score of the node

'index', the index of the node in the MCTS tree

'parent', the index of the parent of the node in the MCTS tree

'UCB', the current UCB amount of the node, whose default value is infinity

'egoFrenetState', the current fernet state of the ego Vehicle

[s ds dss l dl dll]

'avgScore', score / visits

'laneChangingProperties', the current properties of the lane-changing, including times of changing to left, changing to right, and whether the current node has just changed its lane.

);

Select:

In the selection phase, the script employs a variant of the Upper Confidence Bound (UCB) formula:

node’s score / node’s visits + 5 \* sqrt(log(total visits in the tree) / node’s visits)

to balance exploration and exploitation during selection. It selects the child node with the highest UCB score recursively to iterate each level of the MCTS tree and finally return the target node that we need to take further options.

Expand:

During expansion, the expand function generates new nodes by considering possible actions that the car can take and connect them to the node that we just selected. The actions are combinations of possible acceleration changes (acceleration, deceleration, or maintaining current speed) and lane changes (left, right, or staying in the current lane). The function ensures that the speed is within the defined limits and there is no collision whenever accelerate, decelerate, maintain speed or change lane.

Rollout:

The rollout phase (simulation phase), also known as rollout, involves executing a simulation from the newly expanded node until a terminal state is reached. The terminal state is defined either as a collision or the elapse of a maximum time horizon. During simulation, the algorithm choses one of three driving actions each time based on a constant probability: slowing down (10%), speeding up (10%), and maintaining the current speed (80%). When the rollout process reaches the terminal state, it returns the cost of the current node base on the ego Vehicle’s status (more information in the cost function part).

Backpropagate:

Once the rollout phase is completed, the algorithm enters the backpropagation phase. The cost of the simulation is propagated back up the tree, from the terminal node to the root. Each node's score and visit count are updated during this phase.

Cost function:

The cost function mainly consists of 5 parts, cost\_comfort, cost\_safety, cost\_pass, cost\_stimulation & cost\_is\_break\_to\_stop and cost\_lane\_changing.

1. **Cost of stimulation:**

The cost of stimulation promotes the car to move forward when the car has not reached the speed limit whose standard level of this amount is 0.0.

if speed < speedlimit

cost = a \* (speedlimit – speed)^2 - b \* (currentAcc)

In this situation, a \* TimeResolution = b; so we let e.g. a = 1, b = 0.2 when TimeResolution = 0.2. As the formula (speedlimit – speed) \* a = b \* (speedlimit – speed) / TimeResolution.

elseif speed > speedlimit

expectAcc = (speed - speedlimit) / TimeResolution;

cost = abs(acc - expectAcc);

This case determines how much should we decrease the acceleration when we have already reached the speedlimit(represented by the parameter expectAcc).

else

cost = 0.0, the standard level.

Moreover, we also want to make sure the car would search for possible paths when encounter obstacles rather than just stop, ( Stopping is kind of like a “comfort zone ” for the ego Vehicle.) we add another variable on the stimulation: **cost\_is\_break\_to\_stop**

if node.time >= MaxTimeHorizon && node.velocity < 1

cost\_is\_break\_to\_stop = 10.0;

else

cost\_is\_break\_to\_stop = 0.0;

end

This ensure the egoVehicle to get punished at the terminal state when its velocity is lower than 1.

1. **Cost of comfort:**

**The cost of comfort are composed of 3 parts:**

cost\_comfort\_jerk, cost\_comfort\_acc, cost\_comfort\_alter

First we set up the comfort cost of jerk, we made it a sigmoid function, restricting the value between 0 to 2.0.

cost\_comfort\_jerk = 2 / (1 + exp(-jerk));

Then, we consider two factors, the comfort cost of the acceleration, determined by the absolute amount of the acceleration; and the comfort cost of the alternation, we do not expect the ego Vehicle to decelerate more than 2 m/s^2, and also we would like to avoid the ego Vehicle to have different direction of acceleration between two respective time resolution.

cost\_comfort\_acc = 0;

cost\_comfort\_alter = 0;

if comfort < -2

cost\_comfort\_acc = - 2 \* comfort;

end

if acc \* node.parentaCC < 0

cost\_comfort\_alter = 5.0;

end

The total amount of the cost\_comfort is the sum of the above three factors:

cost\_comfort = cost\_comfort\_acc + cost\_comfort\_jerk + cost\_comfort\_alter;

1. **Cost of safety:**

**In this part we consider 6 variables:**

1.SAFE\_DISTANCE = 5; We consider 5m to be the safe distance between two cars.

2.Emergency\_Distance = 1; We consider 1m to be the emergency distance between two cars, at this time we would call for a emergency break.

3.Speed = the ego Vehicle’s current speed.

4.acc = the ego Vehicle’s current acceleration.

5.predicted = predicted poses of the actor Vehicles at current time.

6.distance: the distance between ego Vehicle and the actor Vehicle

The initial cost of safety is 0.0: cost\_safety = 0;

Then for all the detected poses of actor cars, we use a for loop to calculate the cost\_safety of each car, then added them together. We use AABB function to check the collision of two cars.

Here’s the detailed process and the cost function of safety that we create:

First we compute the real distance on the x, y - axis separately, (the real distance is the distance between two cars’ positions minus half of both their length/width).

xdistance = abs(node.state(1) - predicted(i).Position(1)) - 0.5 \* objCarDim(1) - 0.5 \* egoCarDim(1);

ydistance = abs(node.state(2) - predicted(i).Position(2)) - 0.5 \* objCarDim(2) - 0.5 \* egoCarDim(2);

Then, if on the y axis, ydistance <= 0, that means on the current lane, ego Vehicle may collide with the actor Vehicle, so we continue to check the xdistance:

if xdistance <= Emergency\_Distance && xdistance >= 0:

cost\_safety\_temp = -10 \* (xdistance - Emergency\_Distance) + 3.0 + 20 \* speed ^2 + 10 \* acc^3;

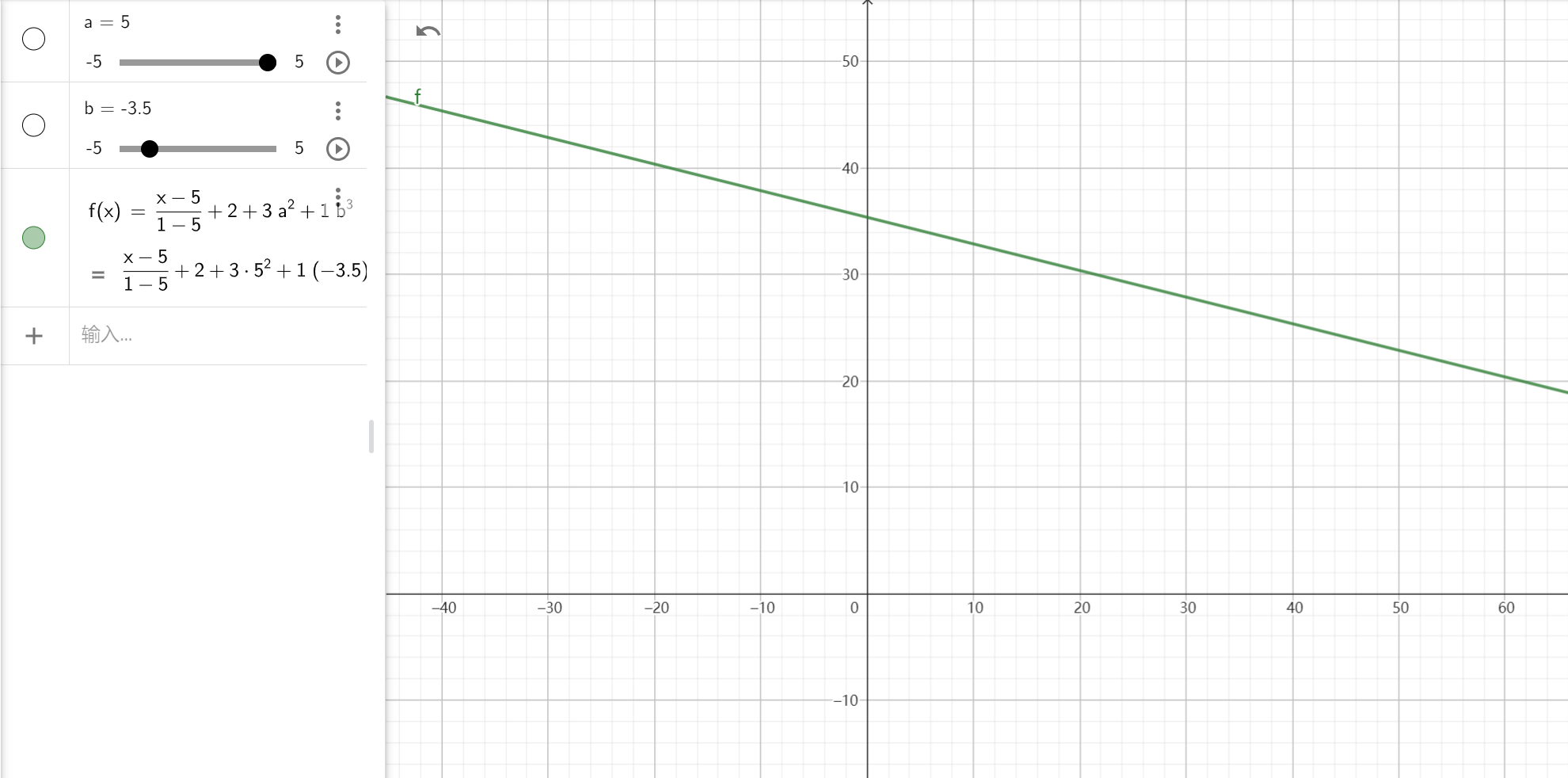
图表, 表格, Excel

描述已自动生成

This graph show the draft of the cost function, where xdistance <= emergency\_distance = 1; The cost function follows a linear trend depending on the distance, whereas also depends on speed^2 and acc^3, to give punishment to cars with higher speed and high acceleration when facing a strong likelihood of collision.

elseif xdistance > Emergency\_Distance && xdistance <= SAFE\_DISTANCE

cost\_safety\_temp = 1.0 \* (xdistance - SAFE\_DISTANCE) / (Emergency\_Distance - SAFE\_DISTANCE) + 2.0 + 3 \* speed ^2 + acc^3;



In this situation, we adjust the slope of the line to -0.25, making it milder. Meanwhile we lower down the coefficient of the speed^2 and the acceleration^3, to make the car would stop near the actor Vehicle around the safety distance.

else meaning that the distance between cars are safe, so

cost\_safety\_temp = max(-1/100 \* (xdistance - SAFE\_DISTANCE) + 2.0, 0.0);

When the xdistance is more than safety distance, we consider the car is driving in a safe environment, so we only give a small cost(no more than 2) to the ego Vehicle, only depending on xdistance to the actor Vehicle.

1. **Cost of pass goal or not:**

**We check the terminal state which the current time of the node is no smaller than the max time horizon we set. If the current position of the ego Vehicle has passed the checkpoint, we consider the car has completed the basic target in the scenario - passing through the intersection, so the cost of pass is 0.0.**

**Otherwise, we consider the distance between the current position and the checkpoint, the more far away from the checkpoint, the more cost it is for the ego Vehicle.**

if node’s current time >= MaxTimeHorizon

then check：

if node’s position on S side > checkPoint

cost\_pass = 0.0;

else

cost\_pass = 2.0 + abs(node’s position on S side - checkPoint);

end

else meaning that the node is not at terminal state, so

cost\_pass = 0.0;

end

1. **Cost of lane-changing:**

**If in the node we expand, ego Vehicle has changed the its lane, then we give a 20 cost on it. However, we make sure that the lane-changing cost could be balanced with the cost\_safety function if there’s an obstacle in front of it.**

if the ego Vehilce has changed its lane

cost\_laneChanging = 20.0;

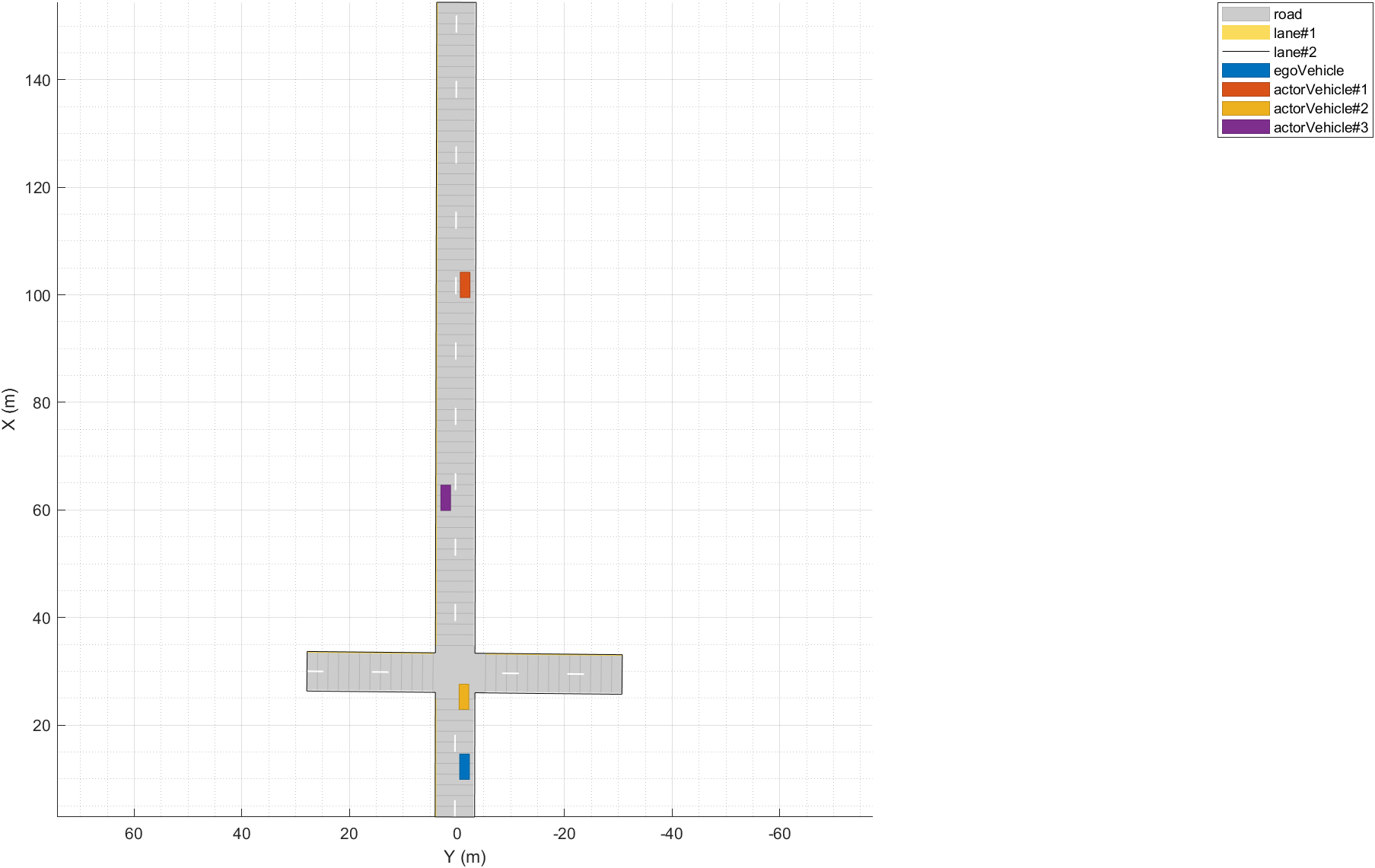
else meaning that the ego Vehicle has not changed its lane, so

cost\_laneChanging = 0.0;

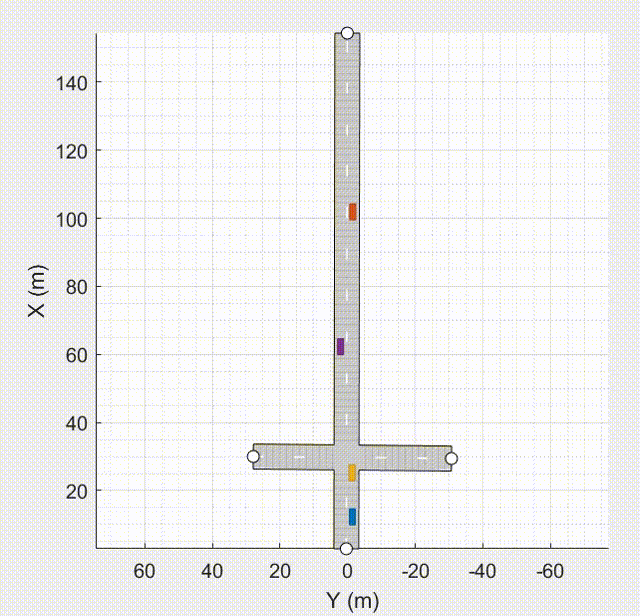
end

## Results

1. Let’s first give a 2-lanes intersection scenario with four cars.

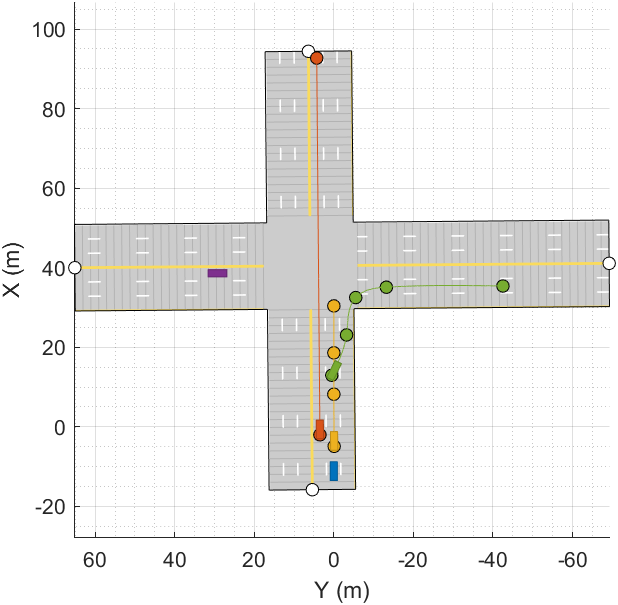


The blue car is the ego Vehicle, and the rest three cars(yellow, purple, orange) are actor vehicles. In this simple scenario, all the actor cars are motionless. The intersection is at x = 34.3, and the destination is x = 153.4. So we expect the car to turn left, and then turn right, and then turn left to bypass the actor cars.

Let’s see how our program matches our expectation: 

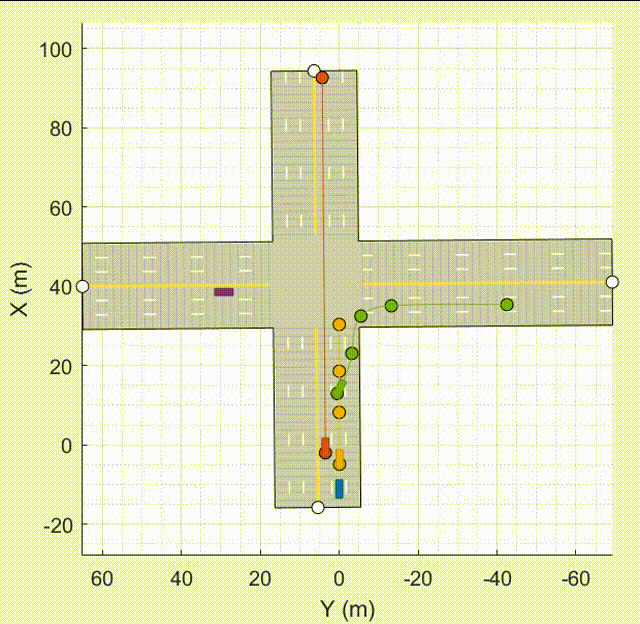
Well done, the ego Vehicle starts at a speed of 5m/s, then moves exactly like what we designed.

1. Then let’s put the ego Vehicle in a more complex scenario, a 6-lanes intersection with moveable actor cars.



The blue car is the ego Vehicle, and the rest are actor Vehicles. The lines are their waypoints. The intersection is at x = 53, and the destination is x = 90.4. How can the ego Vehicle move? It can turn either left of right to avoid colliding with the yellow car, but be careful of the red car simultaneously, after which it would try to avoid the green car…

Let’s see how the car is performing!



Excellent, it succeeds!

## Discussion and Implications

## Future Work

## Conclusion