

Overtake Maneuver

Introduction

Overtake maneuver strategy allows a smart vehicle (an independent agent), to safely overtake the vehicles ahead of it by taking decisions based on the belief/knowledge it has about its environment.

This demo showcases the Simulink model architecture for creating and simulating **synthetic scenarios** by reading as input the scenario file saved using the Driving Scenario Designer (DSD) application. This architecture allows creation of synthetic scenarios, by:

- Marking an actor in the scenario as an autonomous smart actor.
- Installing car-following (driver) model on some of the actors.
- Introducing rogue actors (actors devoid of any intelligence) in the scenario.

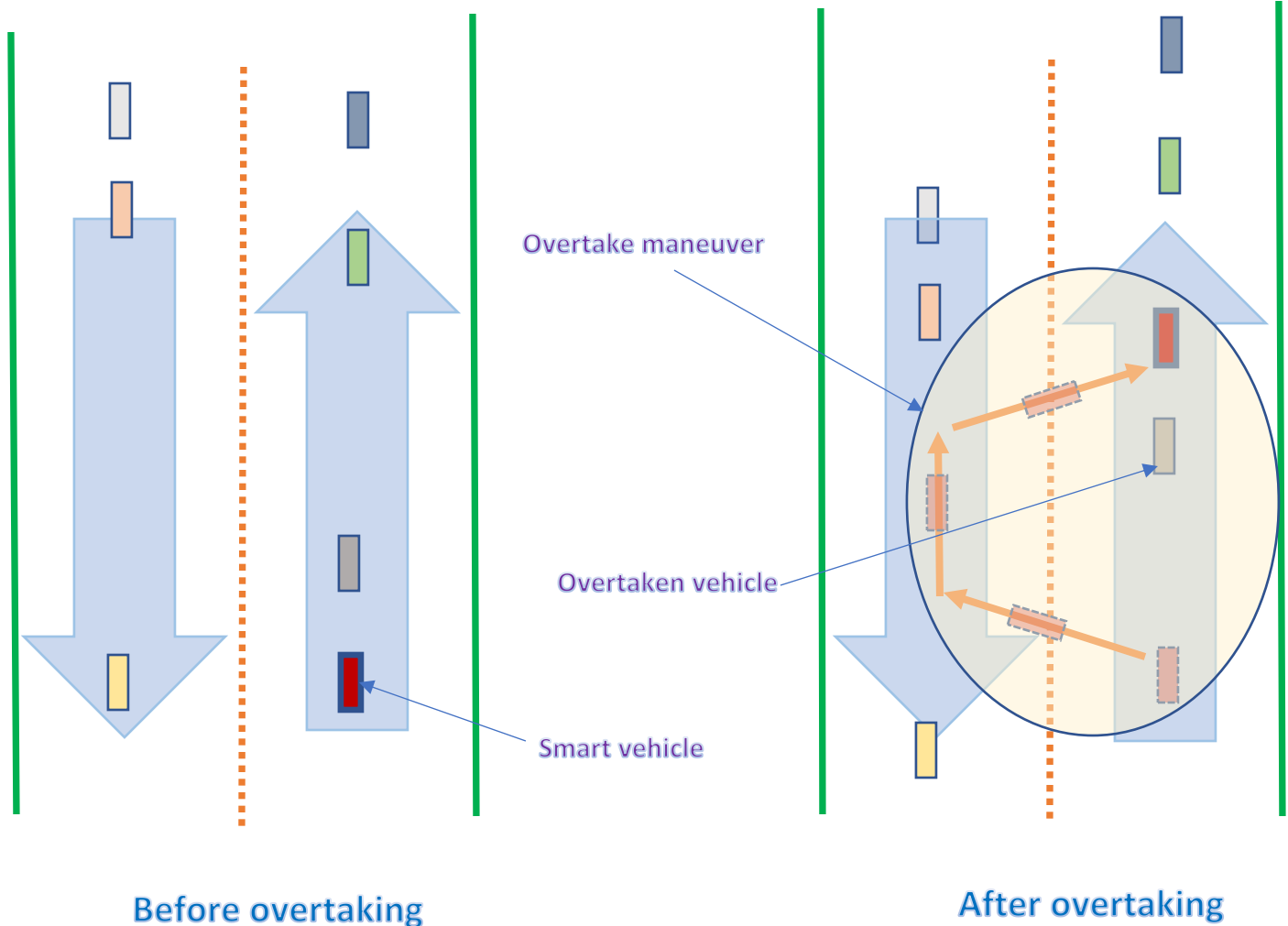
Driving scenario designer (DSD) application is part of Automated Driving System Toolbox (ADST). Refer to the documentation [here](#) for more information.

Configuration parameters can be set for individual actors to observe the variations in the behavior. The plan algorithm in the smart actor supports overtake maneuver on straight road, as a proof of concept.

Toolbox dependencies for this model are: Automated Driving System Toolbox (ADST), MATLAB, Model Predictive Control Toolbox, Simulink, Simulink Coder, Stateflow.

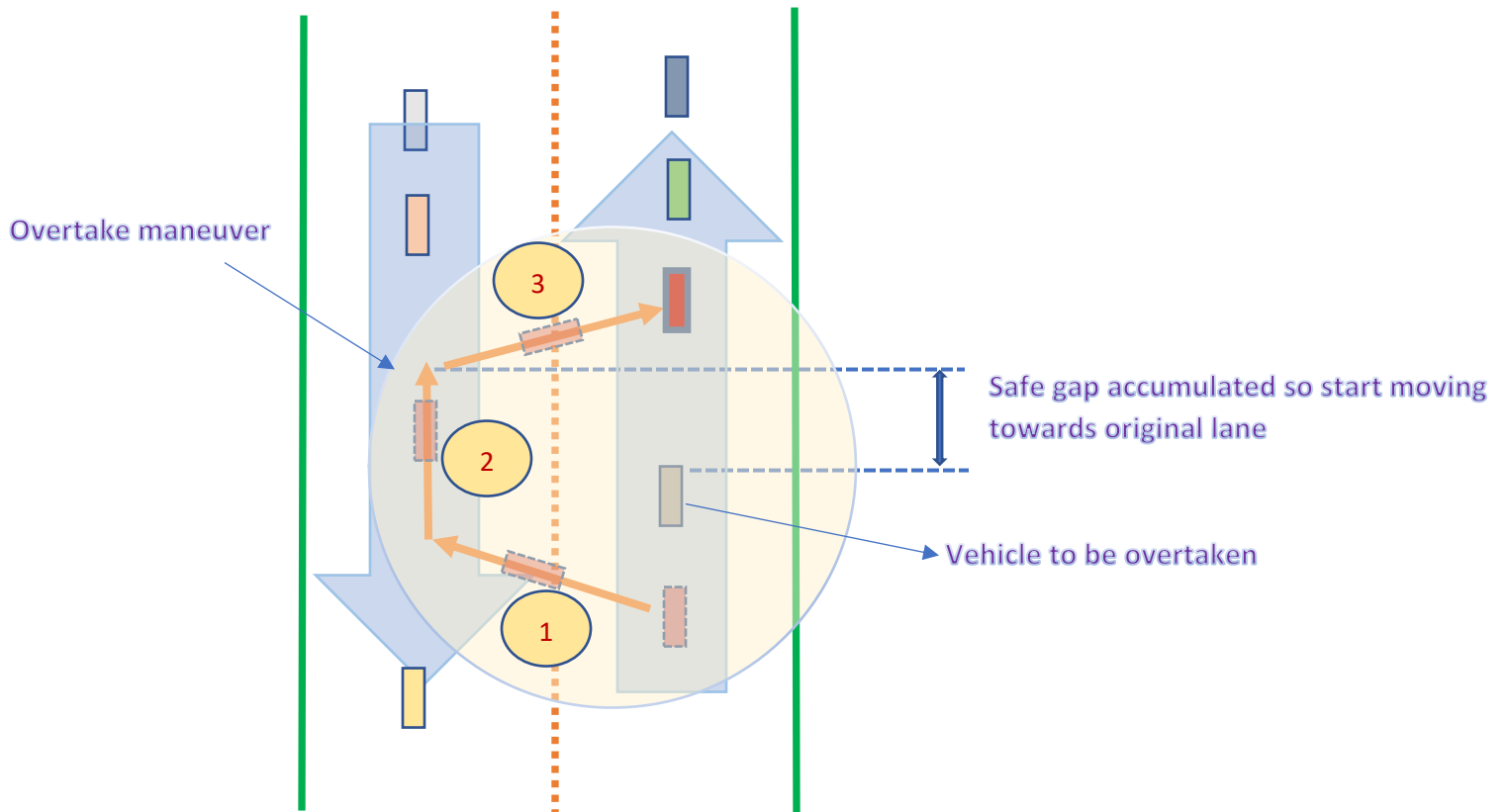
Note: As the model requires a scenario file saved using the DSD application, a sample scenario file named *scenarioInput.mat* is already included in the package. So, you can straightway run the model.

Scenario Description



The scenario consists of a straight road of two lanes, with traffic moving in opposite directions. The smart vehicle moves in its lane (right lane) and continuously looks for opportunity to overtake the vehicle ahead of it. On finding an overtaking opportunity, it moves to the adjacent lane, overtakes the vehicle ahead by moving parallel to it and then moves back to its original lane as shown in the above diagram. If the traffic conditions allow, the smart vehicle can even overtake more than one rogue vehicle in single overtake maneuver based on the configured aggressiveness level.

Overtake Maneuver: Steps



- 1. Movement to the left lane**
Smart vehicle first moves to the adjacent left lane with a 45 degrees inclination.
- 2. Movement in the left lane**
For some duration, smart vehicle moves with maximum acceleration in the adjacent lane till it accumulates sufficient gap with respect to rogue vehicle to be overtaken.
- 3. Movement back to the original lane.**
The smart vehicle moves back to the original lane with a 45 degrees inclination.

Scenario Generation

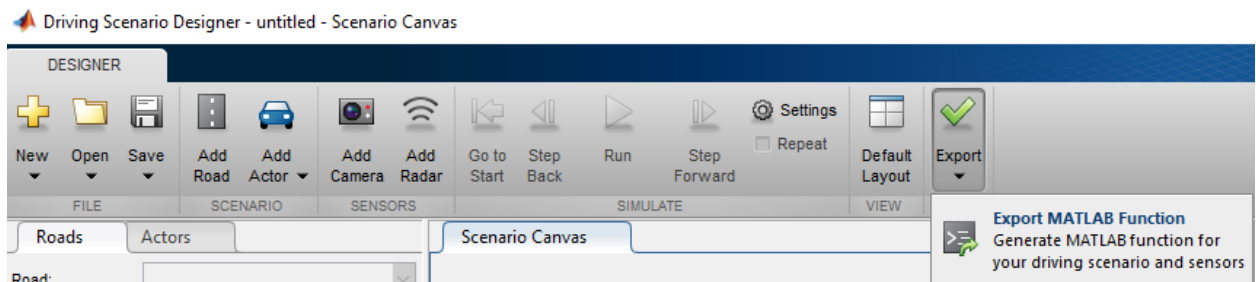
The model requires a scenario file as input which can be generated by creating a scenario in driving scenario designer (DSD) application. This file contains the complete scenario information including roads/lanes information and pose information of all the actors at each time-step of the simulation time.

The implemented overtake maneuver plan algorithm (can be changed) in the smart actor expects the scenario to have:

- A road with two opposite-direction straight lanes with right lane having +x directed vehicular traffic as shown in [Scenario-description](#).
- Rogue vehicles moving with constant velocity. The smart actor takes its decision assuming that actors surrounding it move at constant pace.

After creation of scenario in the DSD application, generating the required scenario file is a 2-step process:

- Export the scenario as a MATLAB function in a .m file.

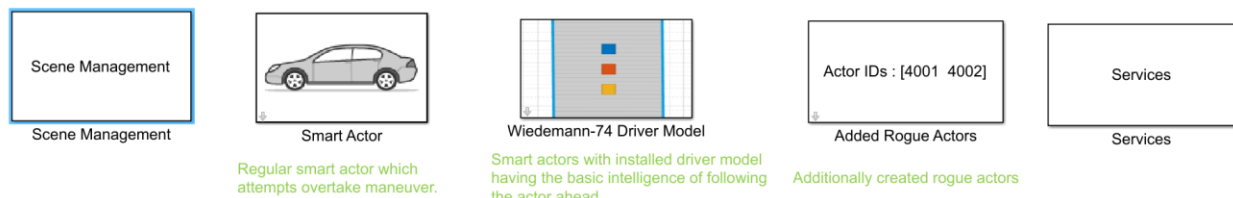


- Call the function **scenarioRecord (generatedFile, 'scenarioInput')** from MATLAB terminal. The scenarioRecord.m file is included in the demo.
generatedFile – The name of .m file generated in the first step.
scenarioInput – The name of output scenario file.

Model Architecture

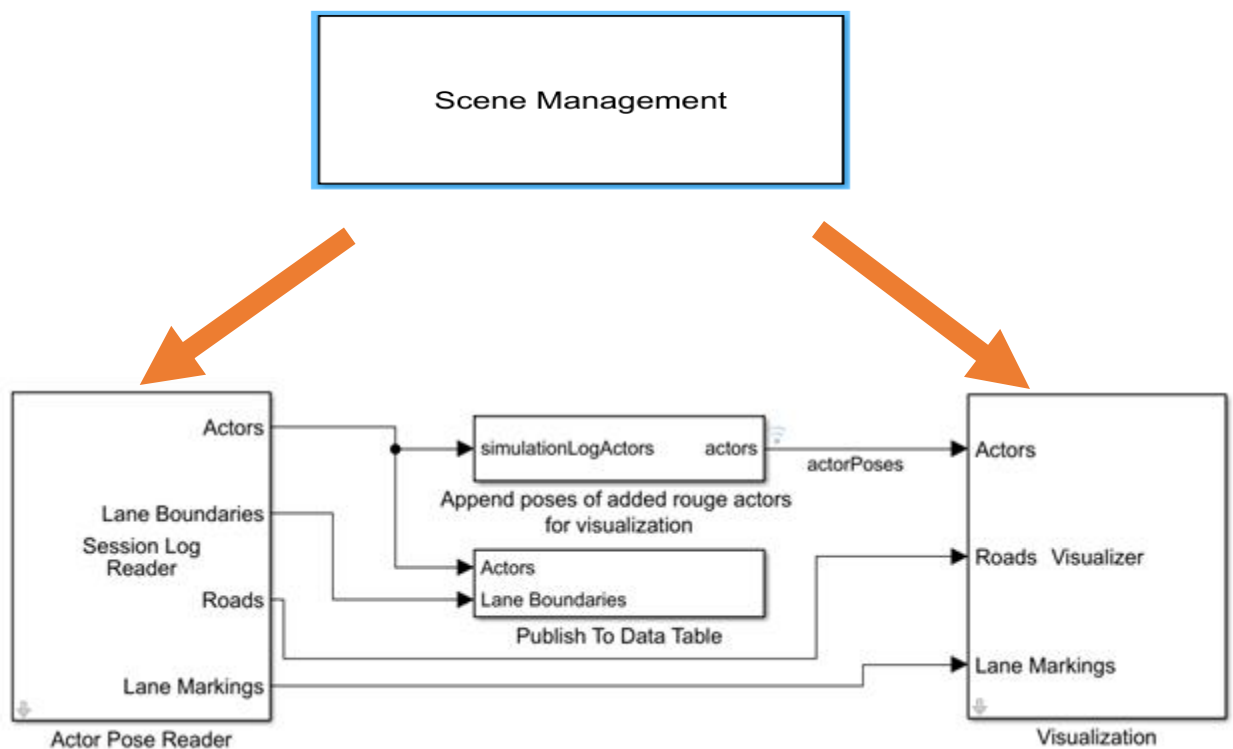
The diagram below shows the various architectural blocks of the model.

Automated Driving Traffic Scenario Modeling - Overtake Maneuver



Copyright 2018 The MathWorks, Inc.

Scene management



As shown in the figure, scene-management mainly consists of Actor-Pose-Reader and Visualization.

Actor Pose Reader

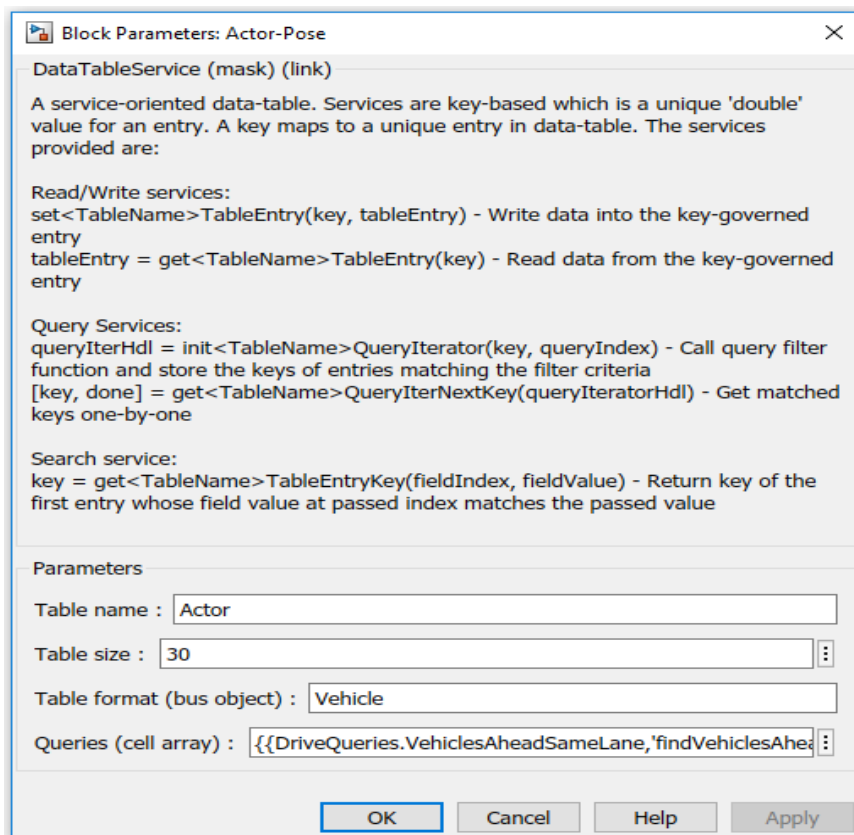
It consumes the above discussed scenario file as input. As the model simulation time progresses, it reads the corresponding pose information from the file. Additionally, it also takes as input the updated pose information for smart agents (i.e. regular smart agents and agents with a driver model installed). As output, this block gives the pose information of all the actors, lane boundary information of the smart actors, and the road-infrastructure information.

Visualization

The 'Actor Pose reader' block outputs are fed to the 'Visualization' block. The user can choose to have the visualization either in the world coordinates or with a selected smart-actor as origin.

Services

The information about vehicles (rogue as well as smart actors) is stored in Data Table Service block. It provides the services equivalent to a database where information can be written to/read from. It also provides custom query interface for querying the database.



It is implemented using C++ MEX S-Function and has functions registered as services provided by the Data Table. The S-Function of the data table has DWork to maintain the actual data. The services provided are key-based, with each key mapping to a unique table-entry. User must use a value which is unique to each actor as the key. In this example, actors are vehicles and their unique IDs are used as the keys.

Services provided are:

- Set-entry: Writes to the Data Table. Key is used to map the data to a unique entry in the table.
- Get-entry: Reads from the Data Table. Key is used to fetch the required entry from the table.
- Querying Service: It allows to query the Data Table to get the desired entries which match some filter criteria.

Querying is done by calling two sub-services:

- ❖ Init Query Iterator – Find the entries in the Data Table which match the query criteria and store the keys of those entries in a DWork. Input parameters are:
 - (i) key – Key of the querying actor.
 - (ii) queryIndex – Query enumeration which maps to a Simulink function where the filter logic to select the table entries which match the criteria is implemented. For instance, in driving scenarios, it could have the logic to select the vehicles which are ahead in its lane.

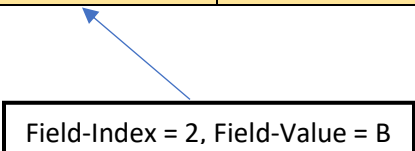
It returns a query handle for this query call.

- ❖ Iterate over the query results – Use the query handle returned by the above call, to fetch the selected keys one-by-one (1 key is returned per call). Along with the key, a 'done' flag is returned to indicate whether all the keys have been read from DWork populated using 'init query' call.

Users can define custom queries as enumeration and map them to their respective implemented Simulink filter functions.

- Search service: It returns the key of the first entry which matches the supplied 'field index – field value' pair. Field index is the index of a field in a table entry (row) as shown below.

A	B	C	D	E
---	---	---	---	---



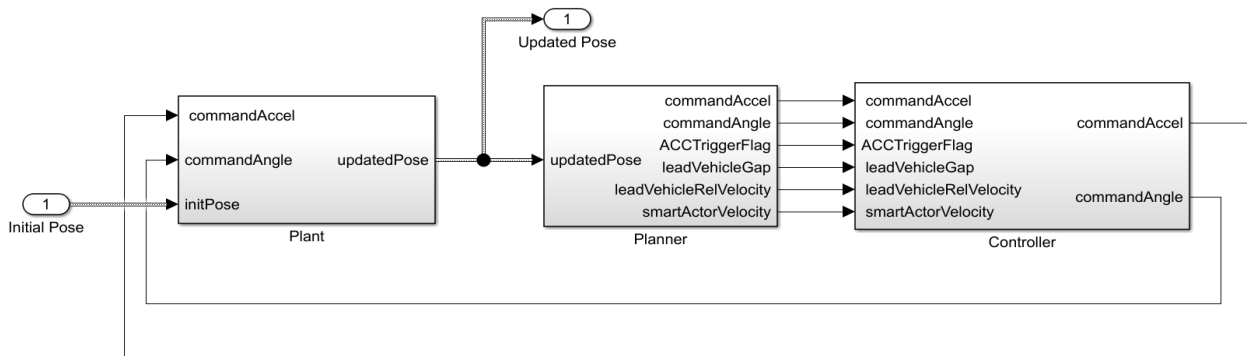
Field-Index = 2, Field-Value = B

This example uses two Data Tables:

- **Actor Data Table** – Contains the current pose information of all the actors. Each actor has a dedicated entry in the Data Table with Actor-ID being the primary key. This information is updated every step-time to reflect the latest pose information of all the actors. It represents the belief/knowledge the agent (smart car) has about its surroundings.
- **Scene Data Table** – Contains the lane-boundary information of smart actors. Each smart actor has its own dedicated entry in this table.

Smart Actor

This block allows to nominate any one of the scenario log actors as a smart actor.



Smart actor consists of the plant, planner and controller blocks.

The 'plant' block calculates the smart actor's movement and feeds the updated pose information to the planner. Based on the pose information, the planner executes the current plan or selects a new one if no plan is running and sends the output to controller. The controller then closes the loop by feeding the command-acceleration and command-angle back to plant, using which the plant again calculates the updated position of smart actor.

Plant

This block simulates the smart-car movement. Based on the acceleration and command angle sent by controller, the updated pose is calculated.

Planner

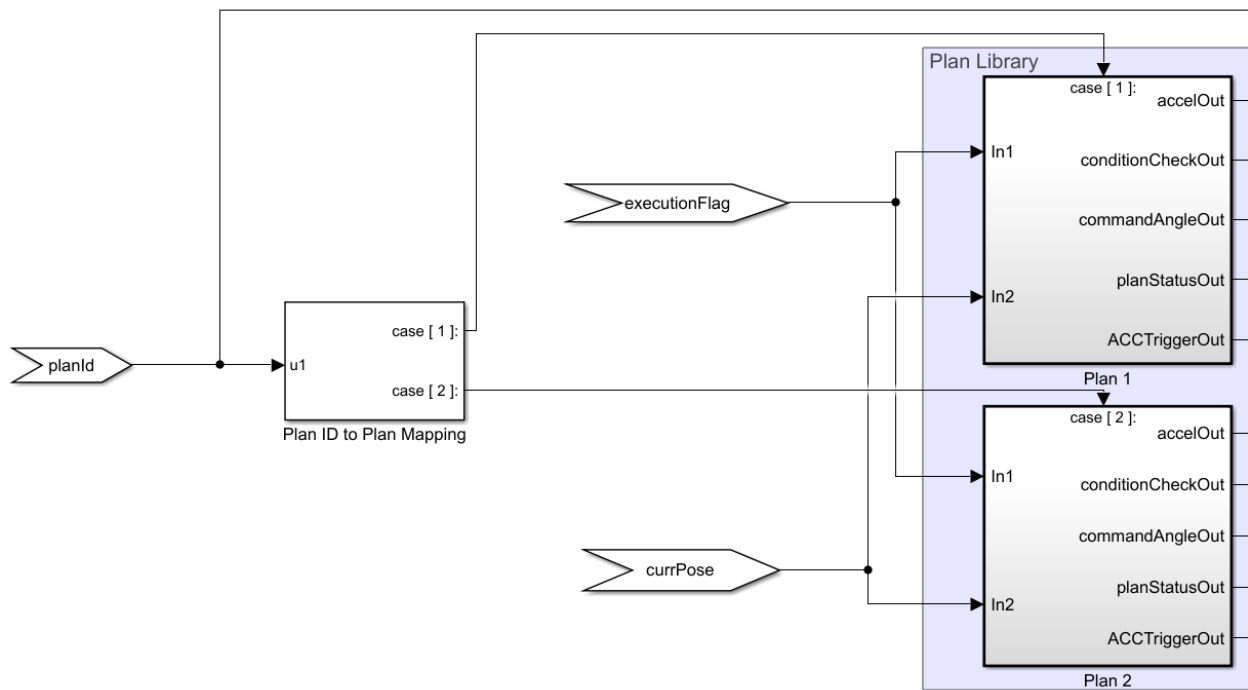
The planner mainly consists of plan-reasoner and plan-library.

Plan-Reasoner

The reasoner periodically checks if a plan is running. If not running, it selects a new plan to execute. Otherwise, continues with execution of the ongoing plan.

Selecting a new plan involves going through the plans stored in the plan library in priority-sorted way and selecting the first plan which has all its entry-conditions met.

Plan Library



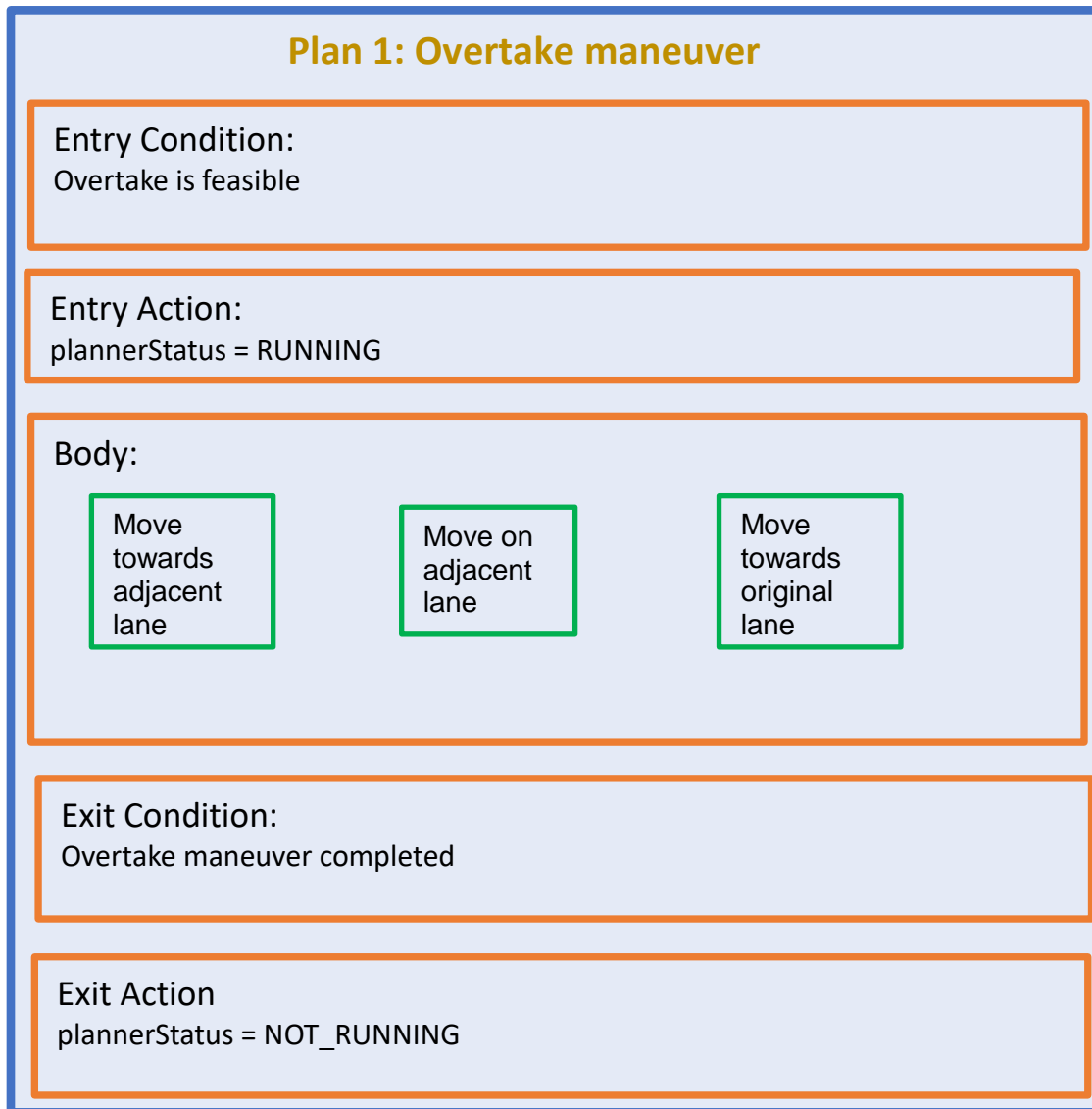
Plan library is a collection of independent plans and their respective entry conditions. Internally a plan can consist of various sub-plans. The diagram above shows two plans in the library with inputs being:

- **Plan ID** – Triggers the respective plan.
- **Execution Flag** – Whether plan is called just to check its feasibility (Value = 0) or actual plan-body execution (Value = 1).
- **Current Pose** – Current pose information of smart car.

A plan has:

- **Entry condition** – The checks required to verify that the plan is feasible.
- **Entry Action** – The actions taken before executing the actual plan. Although the plan can have a set of entry actions, one compulsory action is to set the status of planner as running so that reasoner does not try to select a new plan on its next execution.
- **Body** – The actual implementation of the plan and its subplans (if any)
- **Exit Condition** – The condition which marks completion of the plan.
- **Exit Action** – The set of actions taken at the end of the plan. Again, compulsory exit action is to set the planner status as 'not running' so that reasoner can go for selecting a new plan.

The 'overtake maneuver' plan (first of the two plans) is depicted in a simplified form below:



A plan executes till its completion. During its execution, it continuously sends its output to controller.

Planner sends following signals to Controller:

- Command-Acceleration
- Command-Angle
- Adaptive Cruise Control (ACC) flag. (Value 0 or 1)
- Information for ACC control: Gap w.r.t. vehicle ahead, speed of vehicle ahead, smart car's speed.

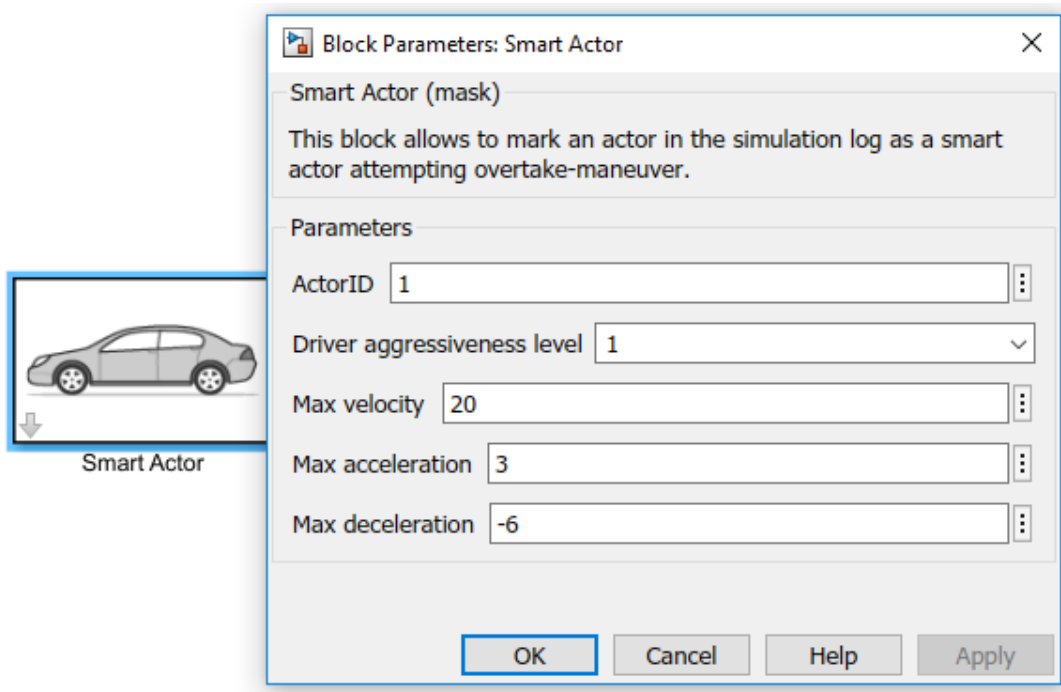
Controller

Based on signals received from the planner, the controller's course of action is:

- **If ACC flag is 0**
Controller just passes through the command-acceleration and command-angle sent by planner as output.
- **If ACC flag is 1**
Controller ignores the command-acceleration and command-angle sent by planner and calculates them on its own. It does so by supplying the signals sent for ACC control (by planner) as input to **Adaptive-Cruise-Control** system Simulink block internal to it. As ACC is for straight line motion, the command angle is sent as 0.

Controller then feeds back the command-angle and command-acceleration to the plant. Controller internally has adaptive-cruise-control(ACC) system to control the gap between the smart-car and the car ahead of it.

You can mark multiple actors as smart-actors by copying this smart-actor block, and then configuring their mask parameters.

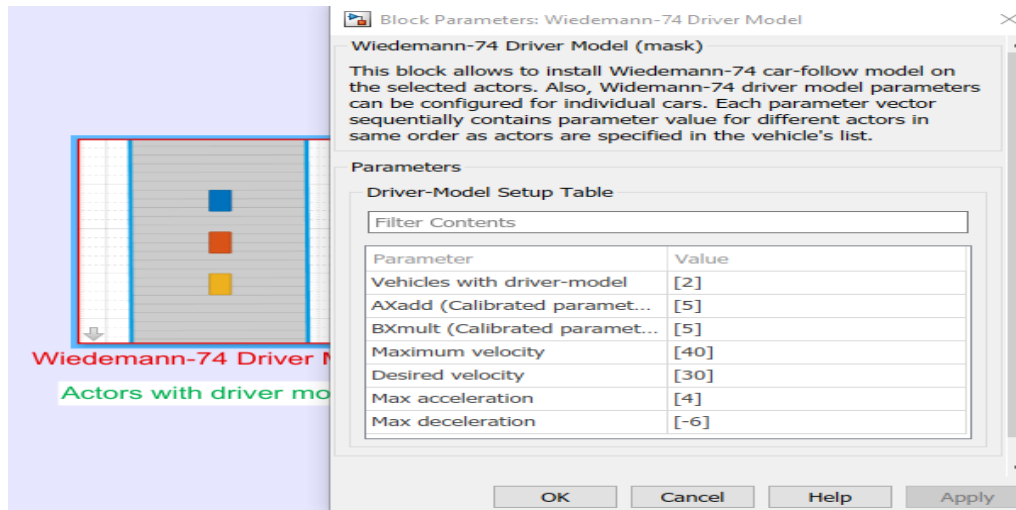


The parameter 'driver aggressiveness level' governs how aggressive an overtake-maneuver the smart car can attempt for. Value '1' means that smart car is not going to attempt for an overtake which involves overtaking more than 1 vehicle in its lane.

Smart Actors with Driver Model

This block allows installing a driver-model (Wiedemann-74 car-follow driver model) on the selected actors. The movement of such chosen actors is now governed by the installed driver model instead of the simulation scenario log. The block mask allows the user to specify the actors for installing

Wiedemann-74 car-follow model, by specifying the actor IDs. Additionally, various configuration parameters of the driver-model can be set individually for each one of them.



Wiedemann-74 driver model: An Overview

It is a car-follow model which enables a car to follow its lead-car while maintaining a safe-gap.

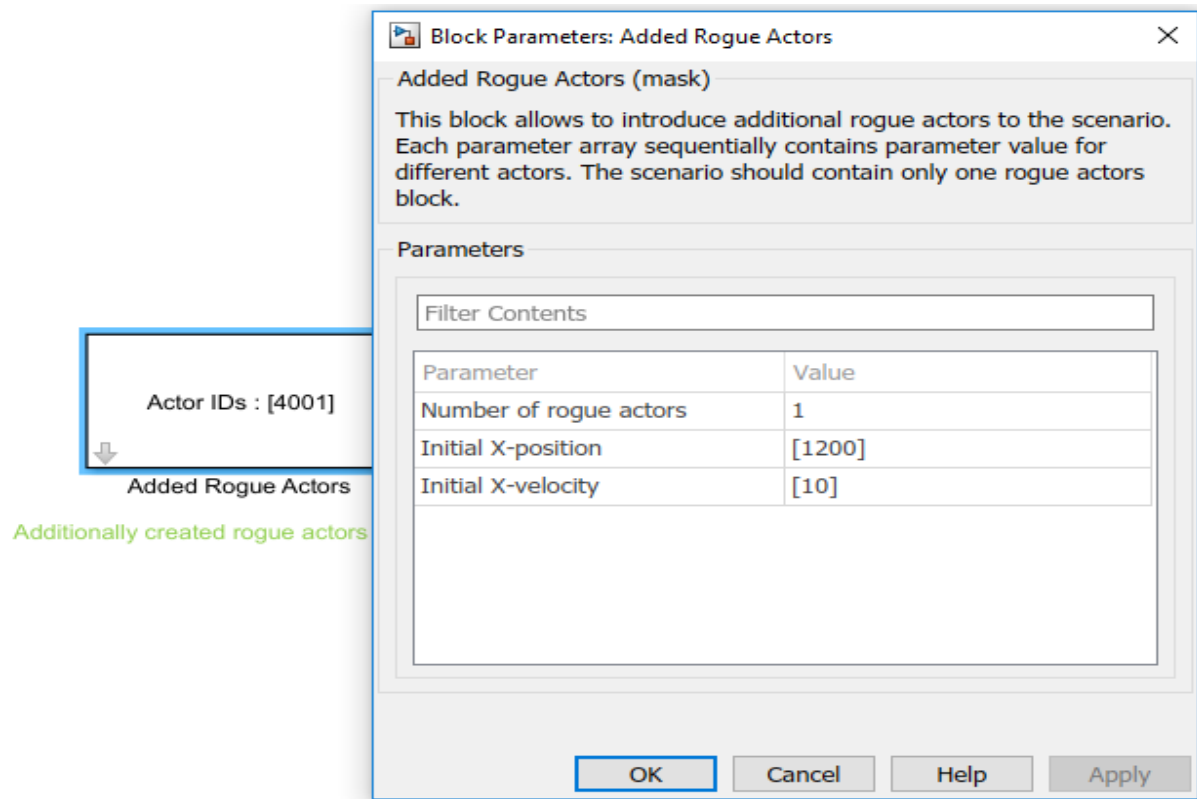
Wiedemann-model is built upon the fact that a driver can be in following driving regimes:

- **Free-driving**
The vehicle ahead is too far away to have any influence on this vehicle. In this mode the driver just tries to reach a certain desired speed.
- **Approaching**
The car is approaching the car ahead in an effort to achieve the desired safe-gap. In this mode, the car (which is currently moving at a higher speed in comparison to car-ahead) decelerates so that it matches the speed of car-ahead when desired safe-gap is achieved.
- **Following**
The driver is closely following the car-ahead while adhering to the safe-gap threshold and with almost zero speed difference.
- **Emergency**
If the distance between cars falls below the required safe-gap, the driver reacts to it by decelerating. Depending on the current gap and relative speeds of cars, the reaction could be aggressive like application of emergency brakes. It can happen if the car-ahead decelerates abruptly.

The driver switches between these regimes based on certain thresholds calculated using current gap and speed difference between the two cars. The regime the car is in governs the calculation of acceleration/deceleration to be applied on it.

Added Rogue Actors

This block allows to create additional rogue actors in the scenario with the initial pose information as mask parameter. This block supports vectorization to introducing multiple rogue actors with different properties.



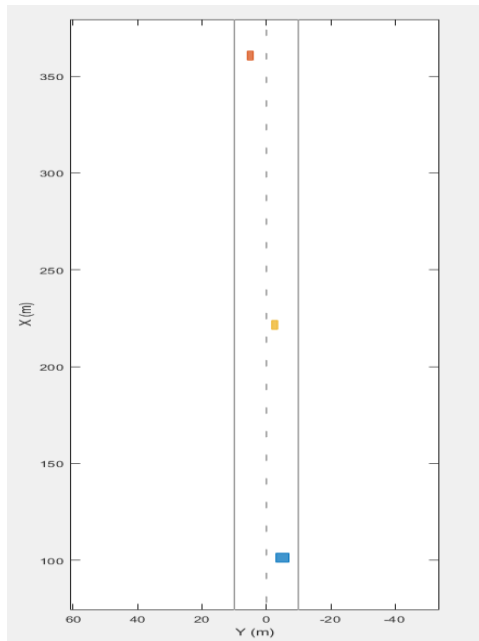
Configuring the model parameters

The model can be configured by setting various block-specific mask-parameters described above. In addition to that, some model parameters can be configured in 'modelParams.m'. Important model parameters that can be configured in the file are:

- Scenario file – User can set the input scenario file name to be used for importing the scenario.
- Sensor range – It governs the smart actor's knowledge about surroundings, which in turn affects its decision making.
- Safe time gap – Minimum time-gap that the smart actor must maintain w.r.t. the vehicle ahead.
- Minimum Overtake Proximity – Smart-actor must be at least this much close to the vehicle in front of it, before considering attempting overtake maneuver.

Scenario Visualization with a sample scenario

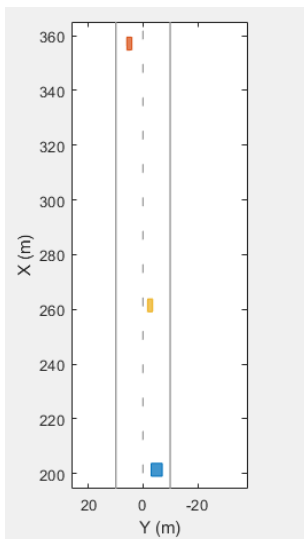
Here is the depiction of a simulation run with a sample scenario. Also, the effect on the behavior of smart actor on configuring its parameters is shown.



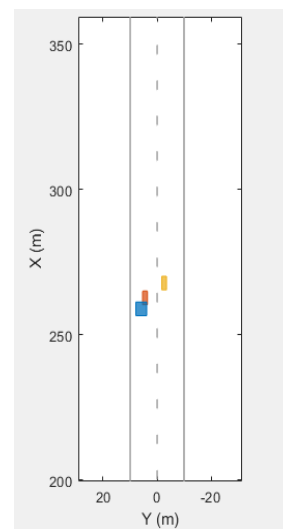
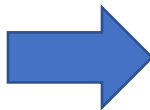
Scenario Description

1 intelligent smart-car (blue), 1 slow moving/stationary rogue car (yellow) ahead of it in the same lane and another oncoming rogue car (red) in the adjacent lane. The intention of the smart-car is to overtake the slow-moving vehicle in front of it by moving to adjacent lane while keeping itself safe from the oncoming vehicle in that lane.

Low Sensing range (50 meters)



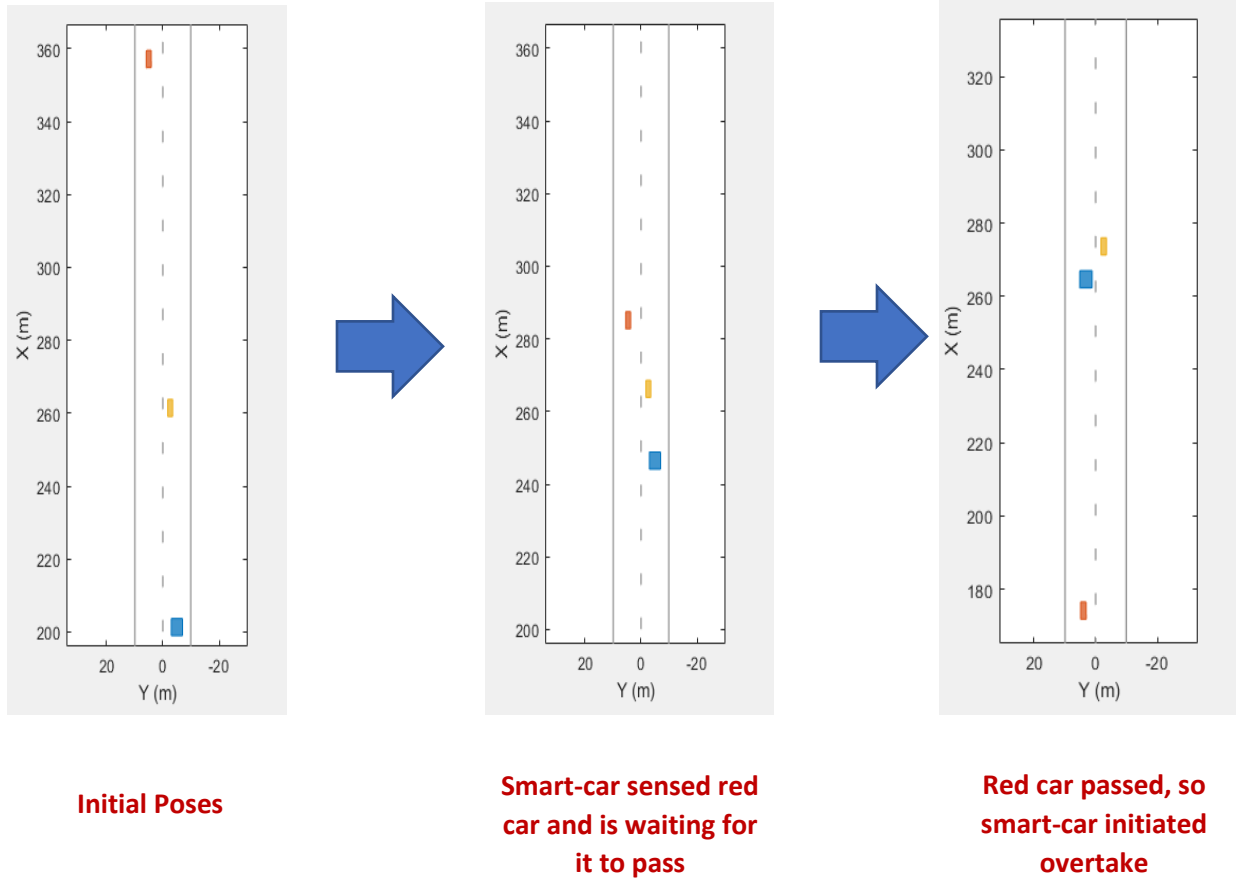
Overtake Initiated



Collision

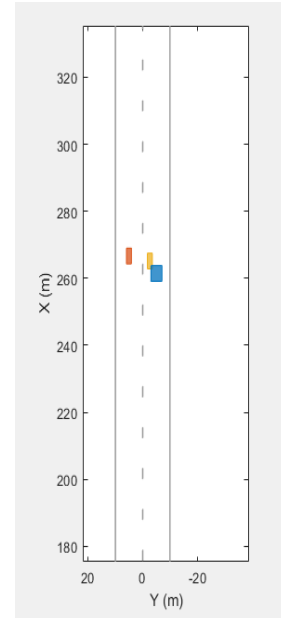
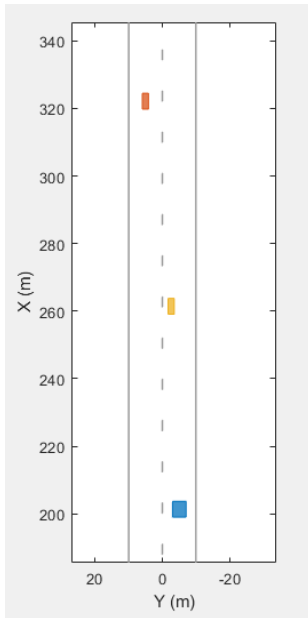
As shown above, to avoid slow moving vehicle ahead, the smart-car moves to the adjacent lane to overtake it. But, due to small sensing range could not detect oncoming vehicle in that lane (at the time of initiating the overtake maneuver) and ran into a head-on collision with it.

High Sensing range (150 meters)



Due to better sensing range, smart-car could now sense the oncoming red-car. Hence it waited for red-car to pass through by moving slowly behind yellow rogue car while maintaining a safe-gap with it. And once red car passed, changes the lane to overtake the slow moving yellow car ahead.

High Sensing range (150 meters) but low deceleration capability (-2m/s^2)



**Smart-car can't overtake
yellow car due to the
approaching red car so only
option is to decelerate**

**Even after applying max-
deceleration, smart-car could
not avoid collision with
yellow car**

In this case, smart car collides with yellow car ahead of it because of lower deceleration capability and at the same time not getting any opportunity to avoid it by overtaking.