Truck Platooning

Introduction

A truck platoon consists of group of trucks moving together. One of the major motivations for truck platooning is that a truck moving behind another one, faces much reduced aerodynamic drag. This can account for significant fuel savings (typically in range of 15-20 percent). In this example, a group of trucks on the same lane of the highway having their respective destinations, dynamically try to form a platoon with their respective leading trucks. A truck forms a platoon with the truck ahead only if it anticipates the platooning to be fuel saving.

Factor in favor of platooning – A trucks saves fuel while it moves as part of platoon. So, if a truck is having a large overlapping journey patch with the truck ahead, it would save more fuel.

Factor against platooning - Extra fuel consumption for accelerating/decelerating (for coming closer or matching speed) to form a platoon with the truck ahead.

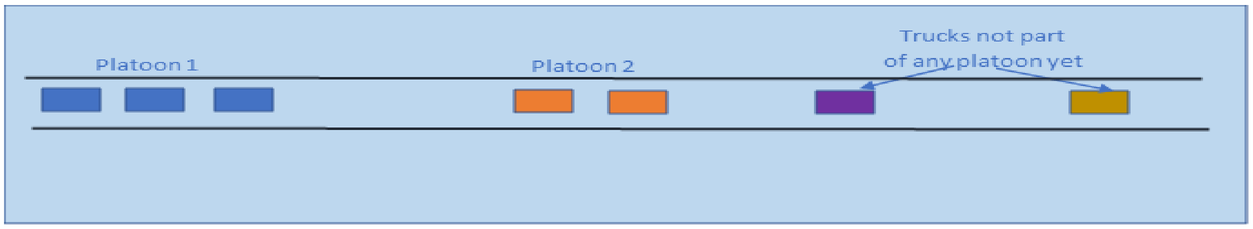
Decision to form a platoon or otherwise is largely dependent on one of the above factors outweighing the other.

Truck platooning strategy allows a smart truck (an independent agent), to form platoon with the truck ahead (if it is fuel efficient) by taking decisions based on the belief/knowledge it has about its environment.

This demo showcases the architecture which allows creating smart-trucks having intelligence of forming a platoon with their respective preceding trucks. Configuration parameters can be set for individual trucks to observe the variations in their behavior. The plan algorithm in the smart actor supports platooning on a straight highway lane 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.

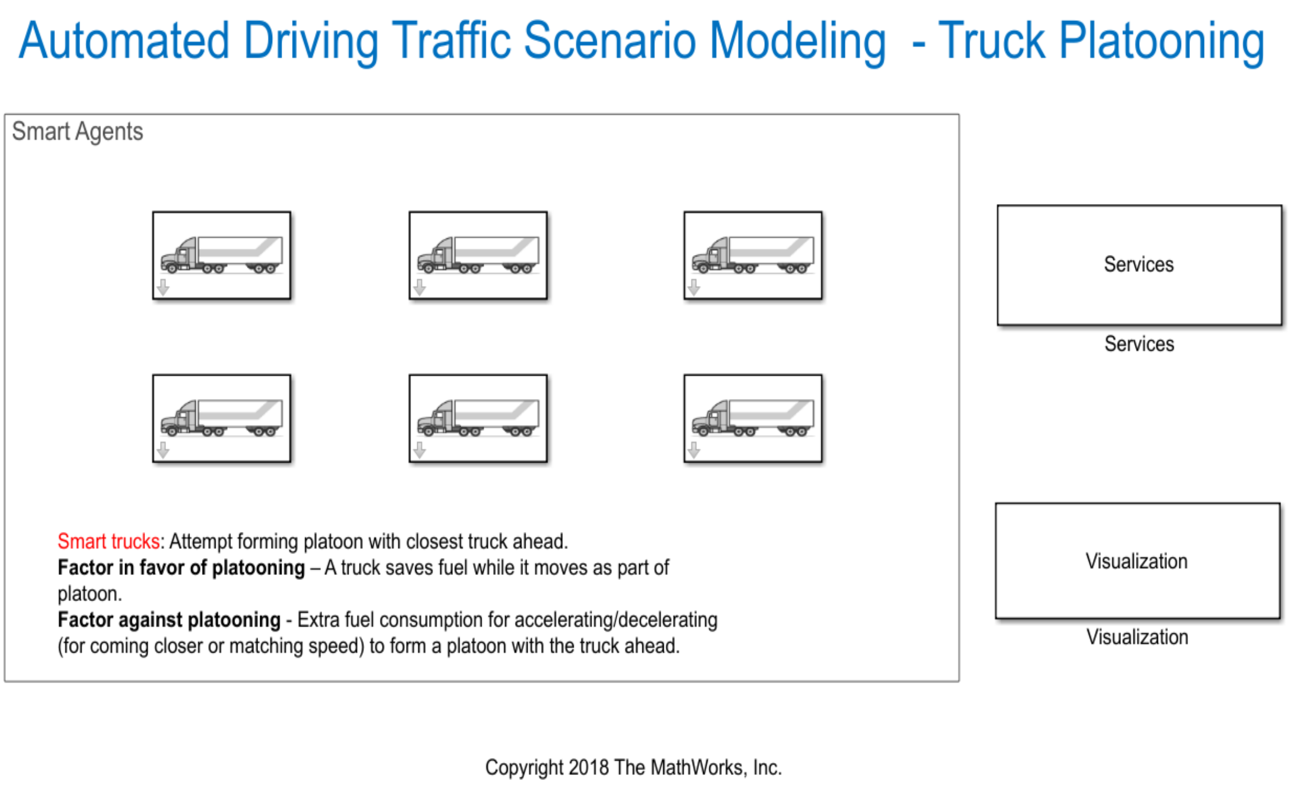
Scenario Description



The scenario has a straight highway lane with only smart-trucks. All the trucks are given respective initial positions, initial velocities and destinations on the highway. Trucks continue to move with their respective initial velocities and they would change it only to avoid collision with truck ahead or for forming a platoon with it. Trucks can dynamically form platoons with their respective lead truck, if they find it fuel efficient. At a time, multiple mini-platoons can co-exist in the simulation. A truck is only concerned with the truck just-ahead and attempts to form platoon with it. It is completely unaware of whether the truck just behind it has joined the platoon.

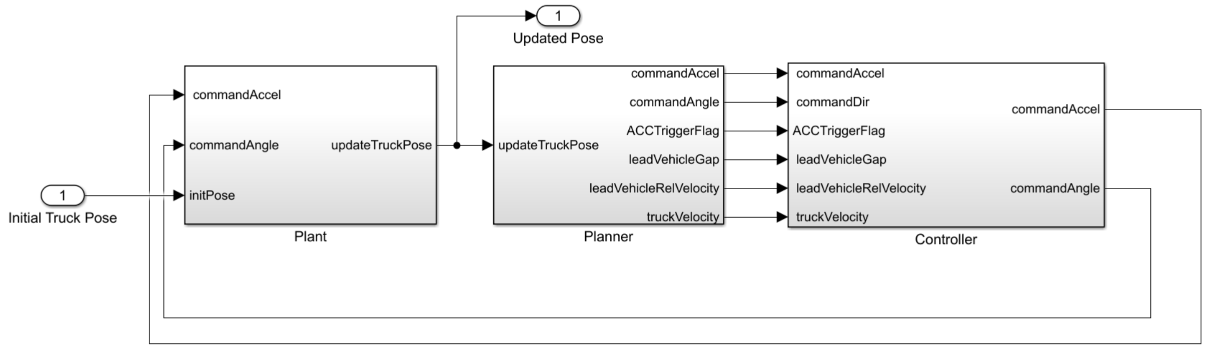
Model Architecture

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


Automated Driving Traffic Scenario Modeling - Truck Platooning
Smart Agents
Services
Services
Visualization
Visualization
Smart trucks: Attempt forming platoon with closest truck ahead.
Factor in favor of platooning – A truck saves fuel while it moves as part of platoon.
Factor against platooning - Extra fuel consumption for accelerating/decelerating (for coming closer or matching speed) to form a platoon with the truck ahead.
Copyright 2018 The MathWorks, Inc.

Smart Truck

This block is a masked sub-system that adds an independent smart truck to the scenario.



Internally, it consists of: Plant, Planner, and Controller.

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-truck movement. Based on the acceleration and command angle sent by controller, the updated pose is calculated.

Planner

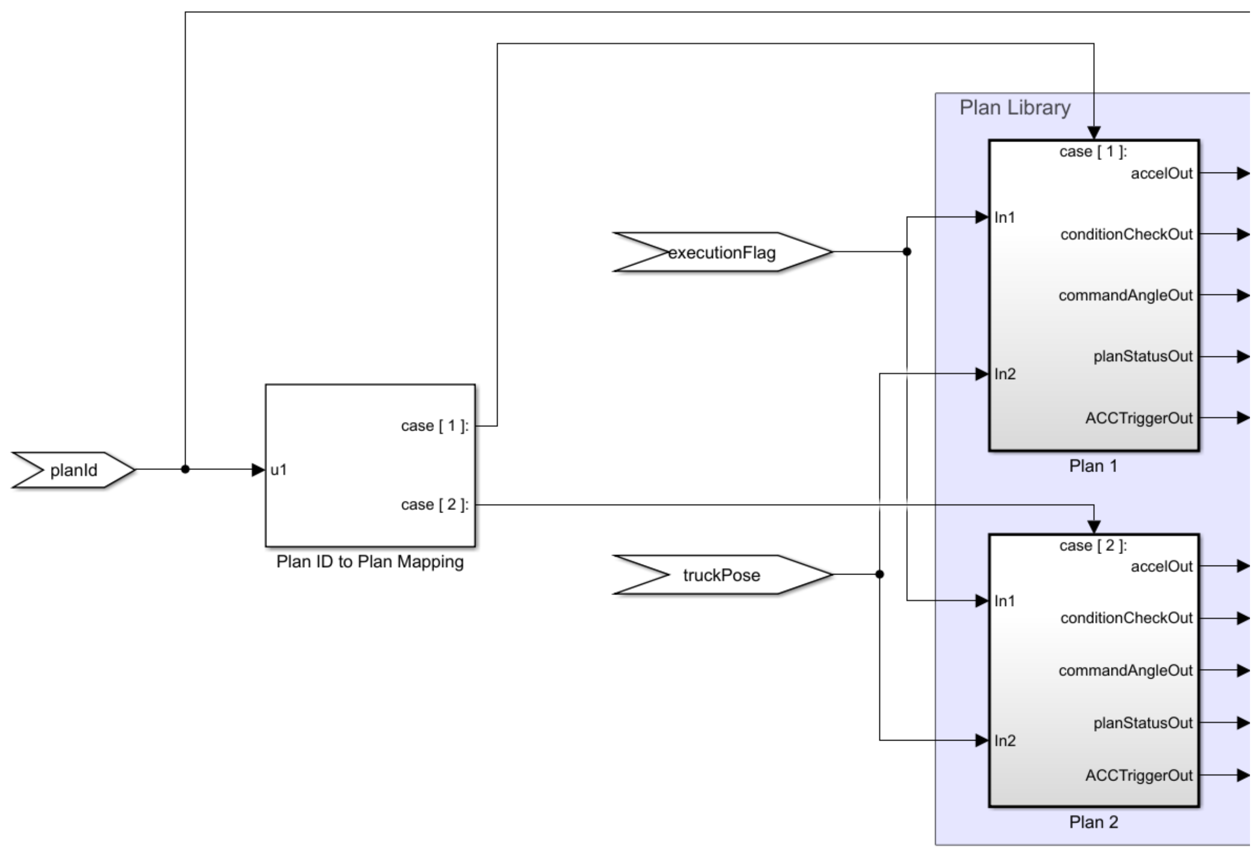
The planner mainly consists of plan-reasoner and plan-library.

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



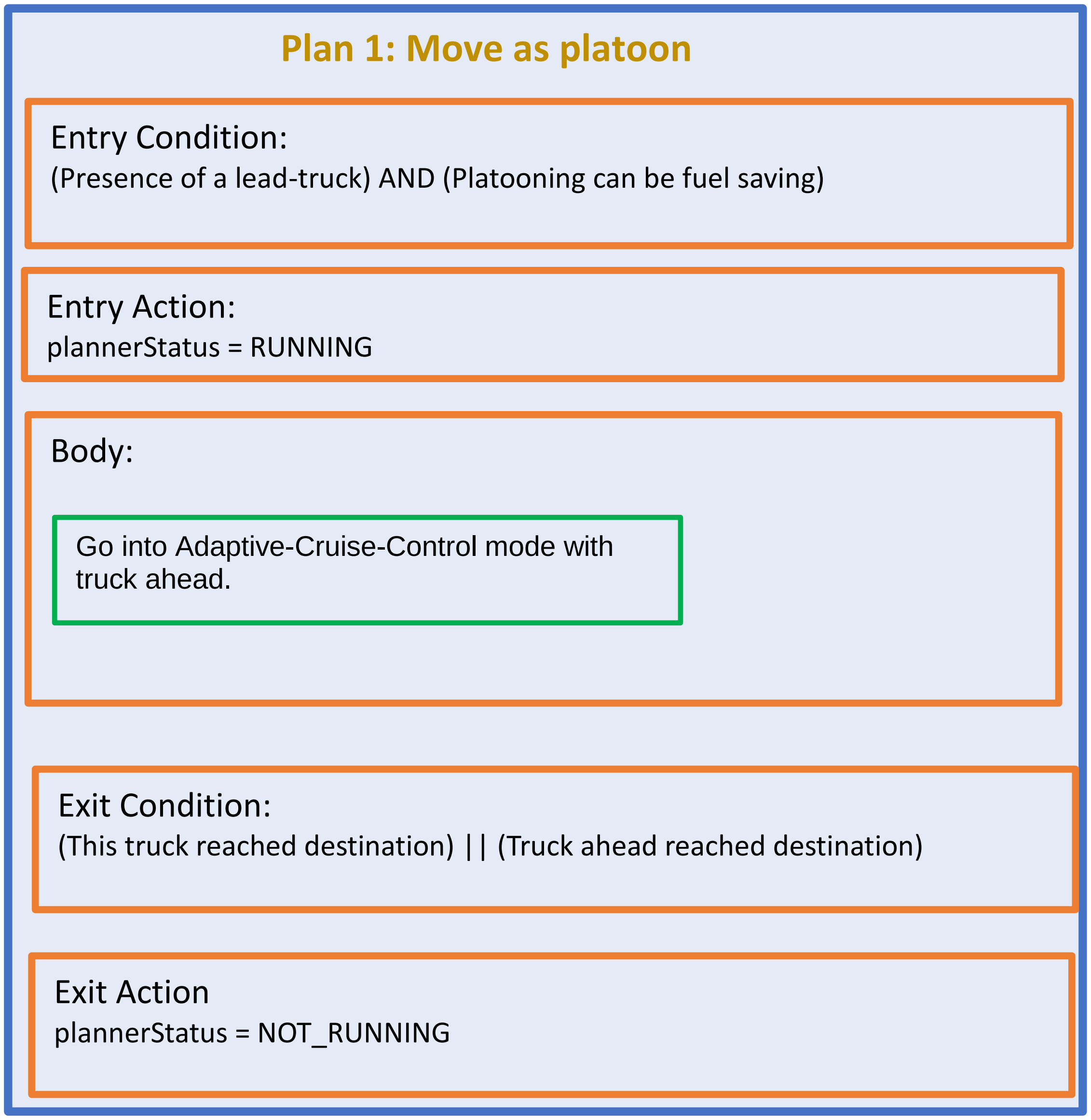
Plan library is a collection of independent plans and their respective entry conditions. Internally a plan can consist of various sub-plans. The diagram above shows two plans in the library with inputs being:

- **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).
- **Truck Pose –** Current pose information of truck.

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 ‘Move as platoon’ plan (first of the two plans) is depicted in a simplified form below:


Plan 1: Move as platoon
Entry Condition:
(Presence of a lead-truck) AND (Platooning can be fuel saving)
Entry Action:
plannerStatus = RUNNING
Body:
Go into Adaptive-Cruise-Control mode with truck ahead.
Exit Condition:
(This truck reached destination) || (Truck ahead reached destination)
Exit Action
plannerStatus = NOT_RUNNING

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. truck ahead, speed of truck ahead, truck's own-speed.

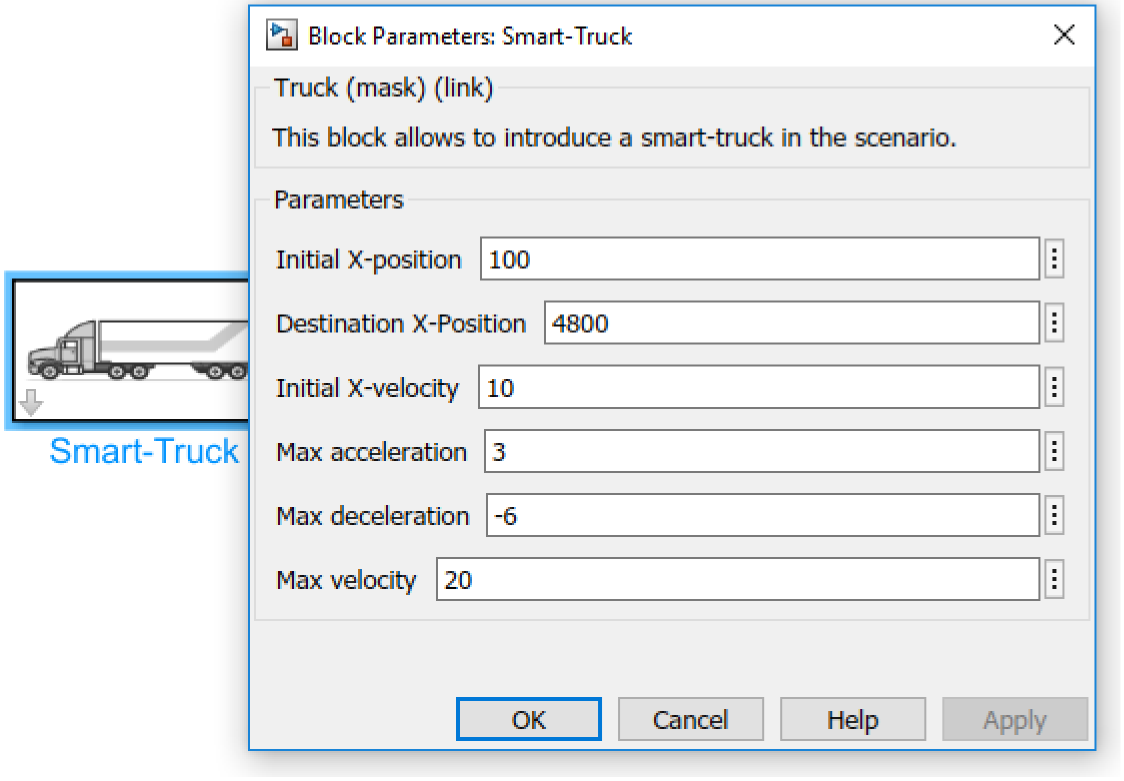
Controller

Based on signals received from planner (explained above), 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 there is only straight line motion, the steering 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 truck and the truck ahead of it.

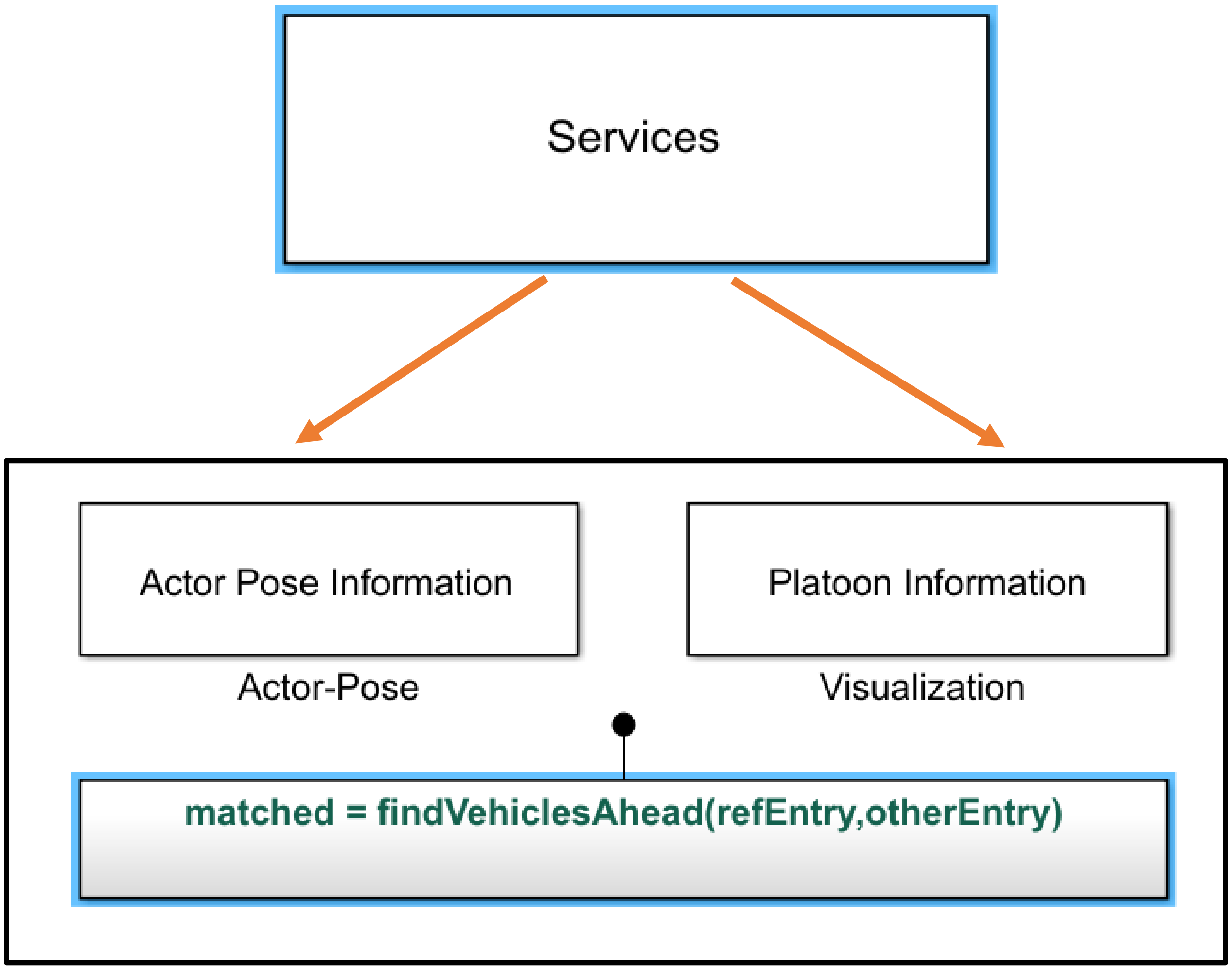
You can create multiple such smart-actors by copying this smart-actor block, and then configuring their mask parameters. The top-level architecture diagram shows three such trucks.

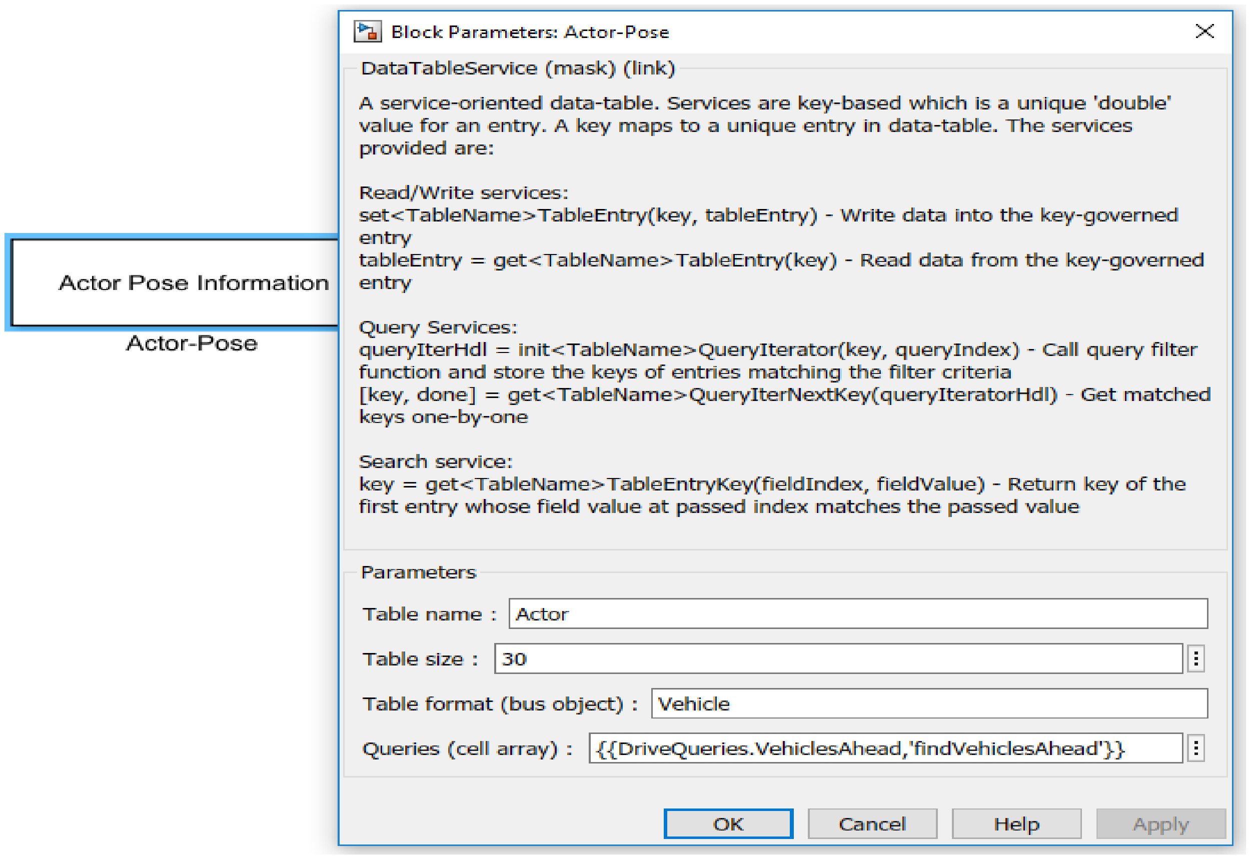

Smart-Truck
Block Parameters: Smart-Truck
Truck (mask) (link)
This block allows to introduce a smart-truck in the scenario.
Parameters
OK
Cancel
Help
Apply

Initial X-position	100
Destination X-Position	4800
Initial X-velocity	10
Max acceleration	3
Max deceleration	-6
Max velocity	20

Services

The information about 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.





DataTableService (mask) (link)

A service-oriented data-table. Services are key-based which is a unique 'double' value for an entry. A key maps to a unique entry in data-table. The services provided are:

Read/Write services:
set<TableName>TableEntry(key, tableEntry) - Write data into the key-governed entry
tableEntry = get<TableName>TableEntry(key) - Read data from the key-governed entry

Query Services:
queryIterHdl = init<TableName>QueryIterator(key, queryIndex) - Call query filter function and store the keys of entries matching the filter criteria
[key, done] = get<TableName>QueryIterNextKey(queryIteratorHdl) - Get matched keys one-by-one

Search service:
key = get<TableName>TableEntryKey(fieldIndex, fieldValue) - Return key of the first entry whose field value at passed index matches the passed value

Parameters

Table name : Actor

Table size : 30

Table format (bus object) : Vehicle

Queries (cell array) : {{DriveQueries.VehiclesAhead,'findVehiclesAhead'}}

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:

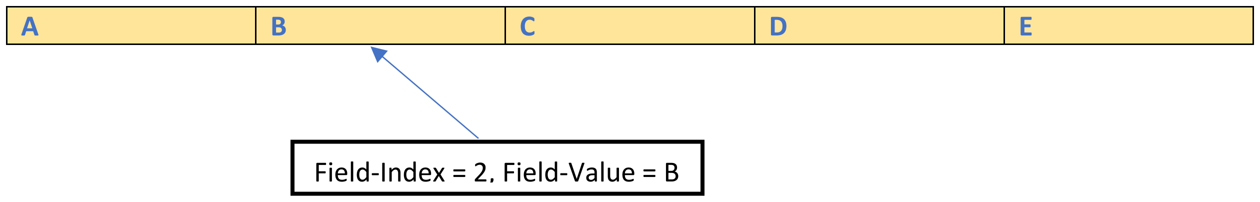
- Set-entry: Writes to the Data Table. Key is used to map the data to a unique entry in the table.
- Get-entry: Reads from the Data Table. Key is used to fetch the required entry from the table.
- Querying Service: It allows to query the Data Table to get the desired entries which match some filter criteria.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

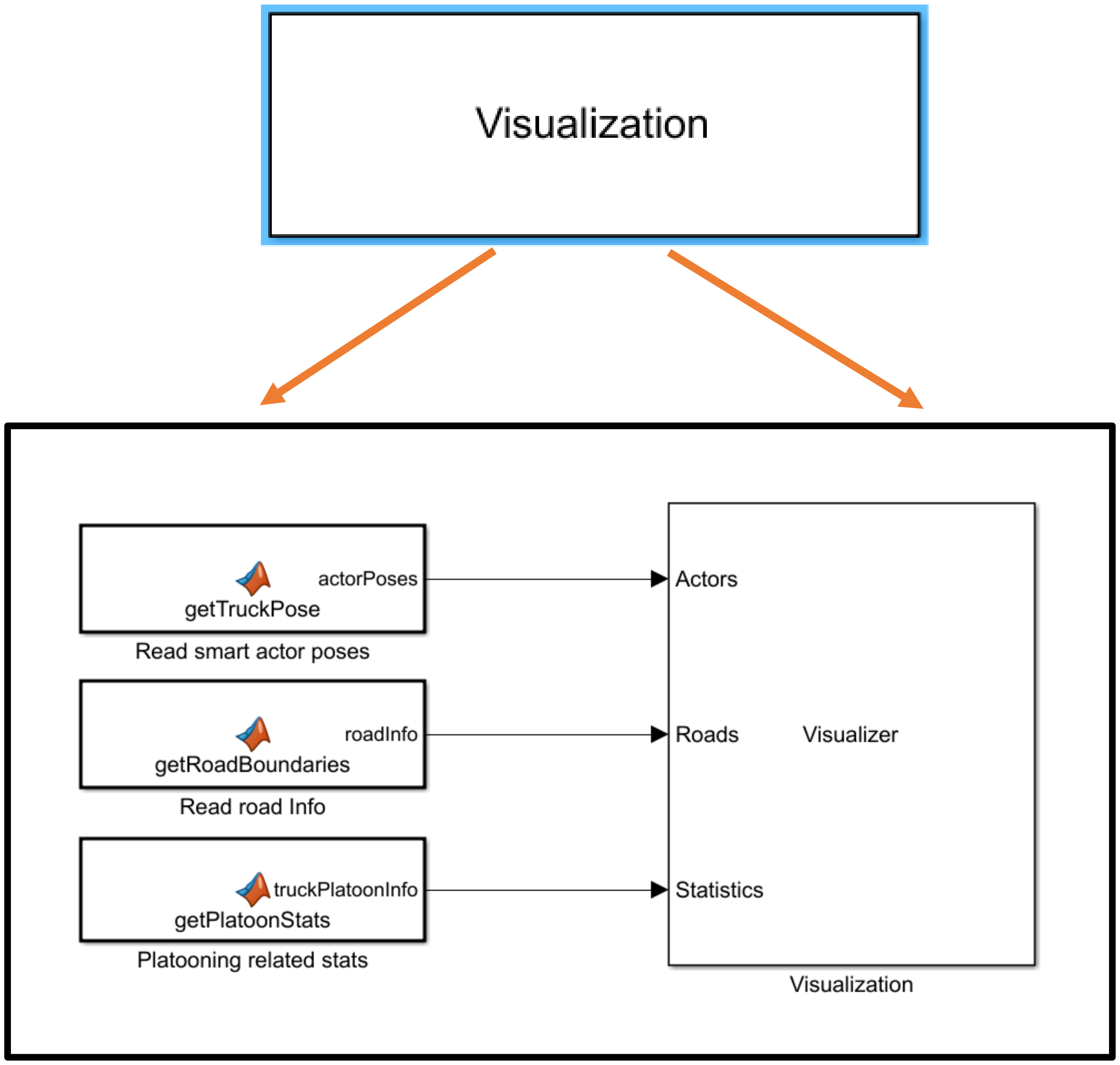


This example uses two Data Tables:

- **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 truck) has about its surroundings.
- **Visualization data-table** – Contains the platoon related statistics to be used for visualization. Each actor has its own dedicated entry in this table.

Visualization

It involves constructing the information required for visualization and sending the same for visualization.



Visualization information consists of three sets of data:

- Smart truck poses – Pose information of all the trucks. Pose contains the position and velocity information of the actor.
- Road information – Road boundary information.
- Platoon stats – For each truck, it has platooning related information**:** truck-ID, platoon-Status (Yes or No), fuel-saving achieved with platoon (if at all platoon formed).
  Platoon-status of a truck tells whether it is moving as platoon with the truck ahead or not.

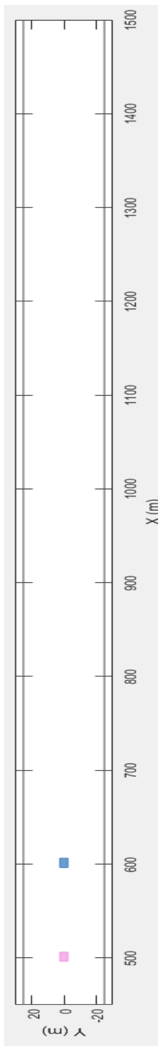
Configuring the model

The model can be configurated 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:

- Sensor range – It governs the smart actor's knowledge about surroundings, which in turn affects its decision making.
- Platoon fuel savings – The percentage amount of fuel a truck saves while moving as part of platoon (due to reduced aerodynamic drag).
- Safe time gap – Minimum time-gap that the smart actor must maintain w.r.t. vehicle ahead.
- Minimum overlap journey for platoon – A truck must have at least this much overlapping journey with the truck ahead to even consider forming a platoon with it.

Scenario Visualization with a sample scenario

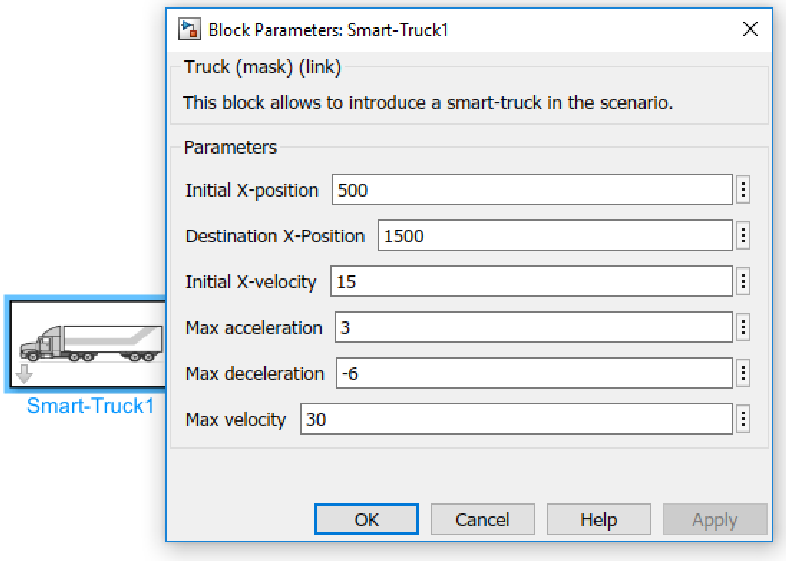
In this section, simulation results are shown for different configuration settings of the trucks. The scenario below shows two trucks with their initial positions.



Screenshot shows the initial positions of the two trucks. Trailing truck on detecting the leading truck ahead, evaluates it as a candidate to form platoon with.

Run-1 (Shorter Overlap Journey)

- Configure trailing truck


Block Parameters: Smart-Truck1

Truck (mask) (link)

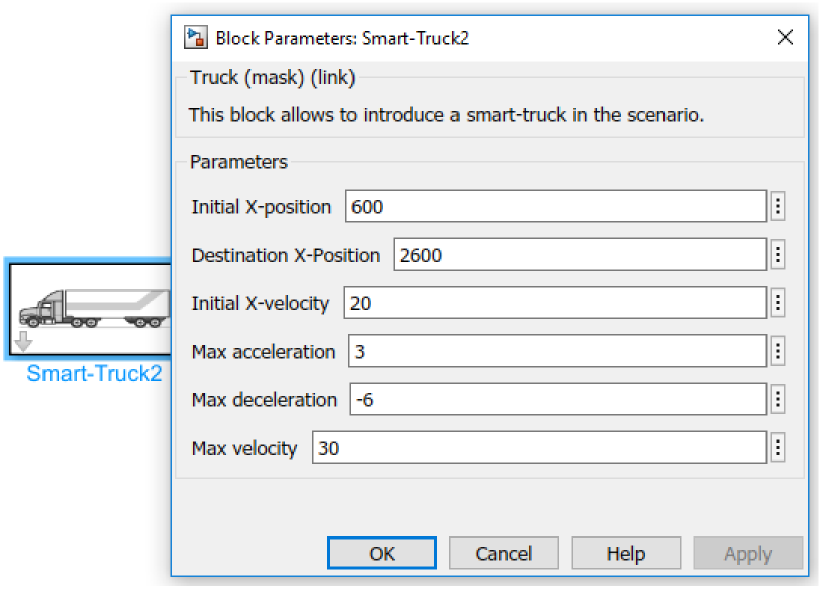
This block allows to introduce a smart-truck in the scenario.

Parameters

Initial X-position	500
Destination X-Position	1500
Initial X-velocity	15
Max acceleration	3
Max deceleration	-6
Max velocity	30

It starts at x=500 at a speed of 15m/s and is destined to go to x=1500.

- Configure lead truck


Block Parameters: Smart-Truck2

Truck (mask) (link)

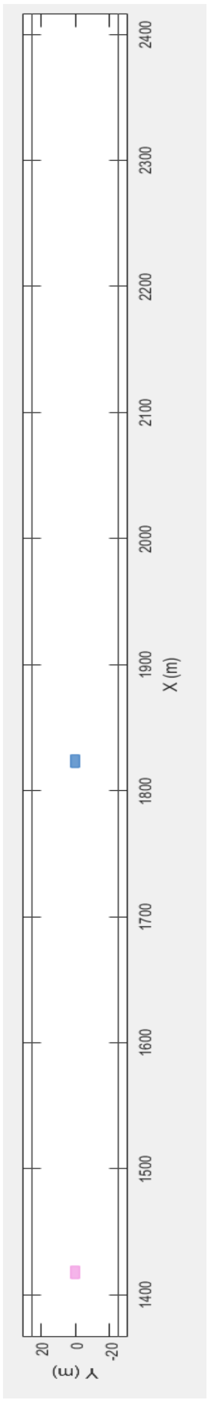
This block allows to introduce a smart-truck in the scenario.

Parameters

Initial X-position	600
Destination X-Position	2600
Initial X-velocity	20
Max acceleration	3
Max deceleration	-6
Max velocity	30

It starts at x=600 at a speed of 20m/s and is destined to go to x=2600.

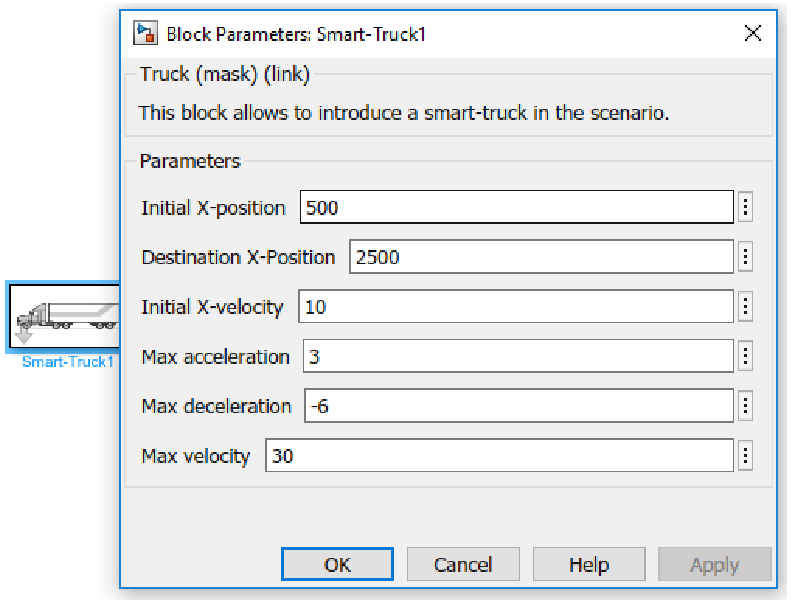
- **Result** – Trailing truck decides against platooning with the leading one.



- On running the simulation with these two trucks, the trailing truck does not form platoon with the lead one (as can be seen in the mid-simulation screenshot, they are moving far apart as governed by their respective initial positions and velocities).
- The major reason being, that they had a shorter overlapping journey (x=600 to x=1500 i.e. 900 meters) hence platoon fuel savings would be applicable for a smaller distance. So, while taking the decision to form a platoon, the trailing truck anticipated that the extra fuel it is going to burn to accelerate for matching up the speed and coming closer to leading one would outweigh the fuel savings it would achieve by forming the platoon. Hence decided against it.

Run-2 (Longer Overlap Journey)

- Configure trailing truck



Block Parameters: Smart-Truck1

Truck (mask) (link)

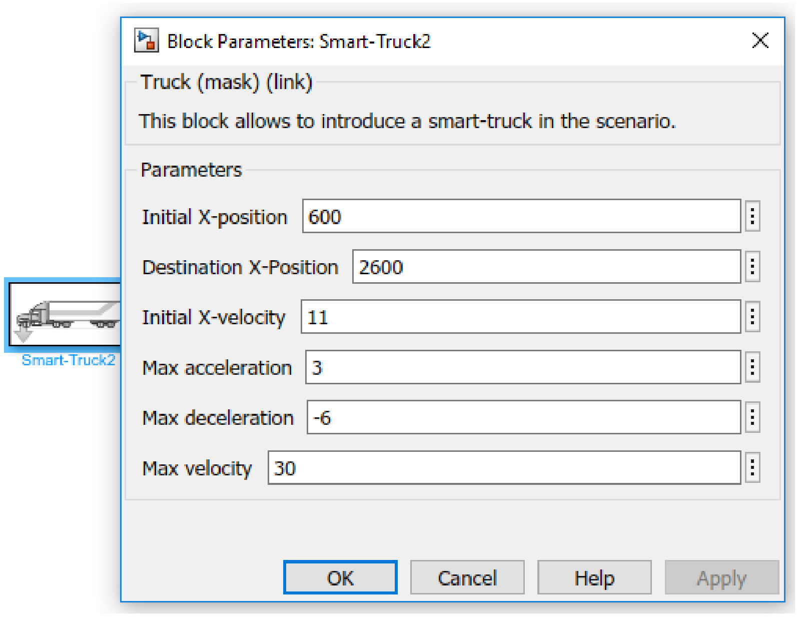
This block allows to introduce a smart-truck in the scenario.

Parameters

Initial X-position	500
Destination X-Position	2500
Initial X-velocity	10
Max acceleration	3
Max deceleration	-6
Max velocity	30

It starts at x=500 with 10m/s speed and is destined till x=2500.

- Configure lead truck



Block Parameters: Smart-Truck2

Truck (mask) (link)

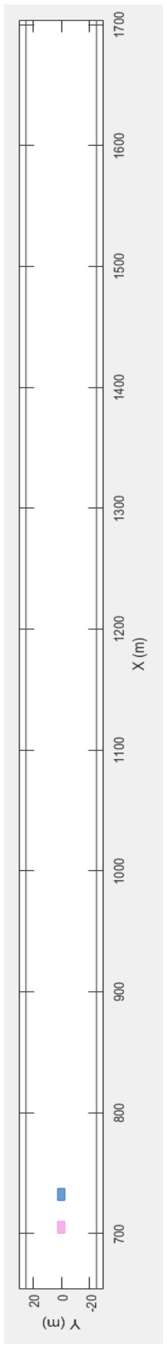
This block allows to introduce a smart-truck in the scenario.

Parameters

Initial X-position	600
Destination X-Position	2600
Initial X-velocity	11
Max acceleration	3
Max deceleration	-6
Max velocity	30

It starts at x=600 with 11m/s speed and is destined to go to x=2600.

- **Result** – Trailing truck forms platoon with the leading one.



In this case, because of much bigger overlapping journey (x=600 to x=2500 i.e. 1900 meters) and lesser speed difference (only 1m/s), trailing truck decides in favor of platooning as it anticipated that to be fuel saving. As shown in the simulation screenshot, the two trucks are moving close to each other as platoon.