Computational Foundations of Cyber Physical Systems

REINFORCEMENT LEARNING-BASED PATH FOLLOWING CONTROL FOR A VEHICLE WITH VARIABLE DELAY IN THE DRIVETRAIN

Group 1

OBJECTIVE

Implement an RL -based path following controller using DDPG algorithm and apply it on the given vehicle model in the paper “Reinforcement Learning-based Path Following Control for a Vehicle with Variable Delay in the Drivetrain”.

WORK FLOW

01 Understanding the Research Paper

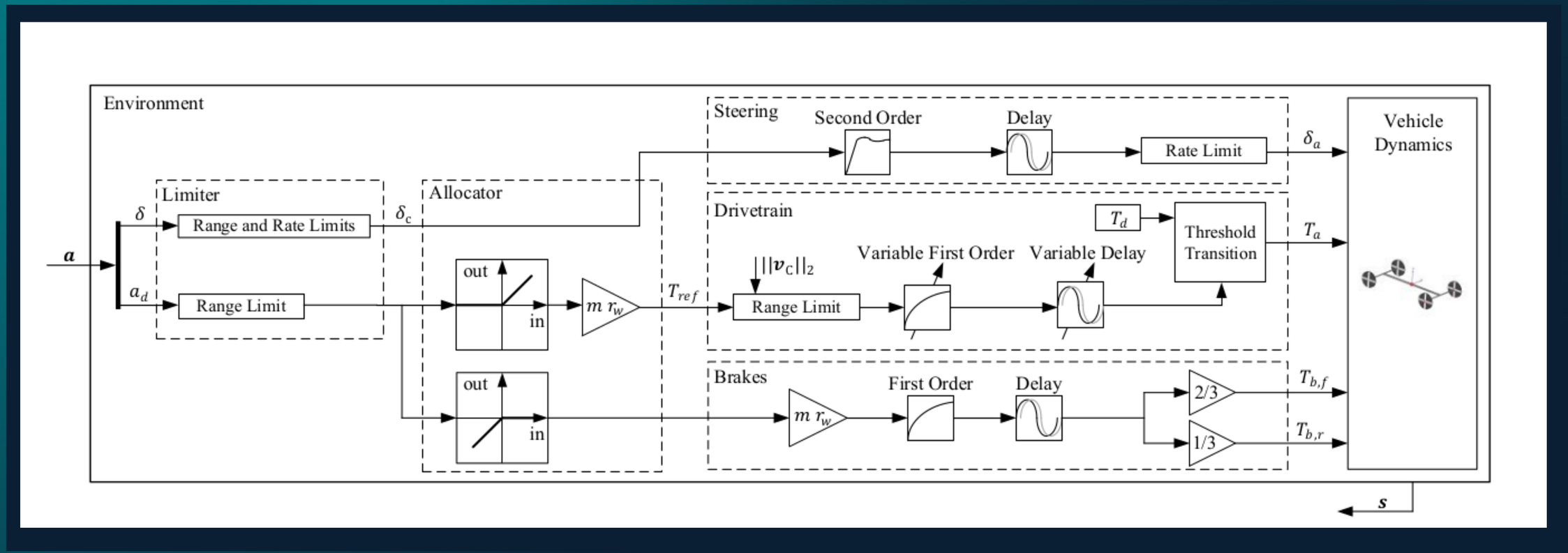
02 Creating the Vehicle Dynamics

03 Computing The Feedback State and Reward

04 Creating The Driving Scenario

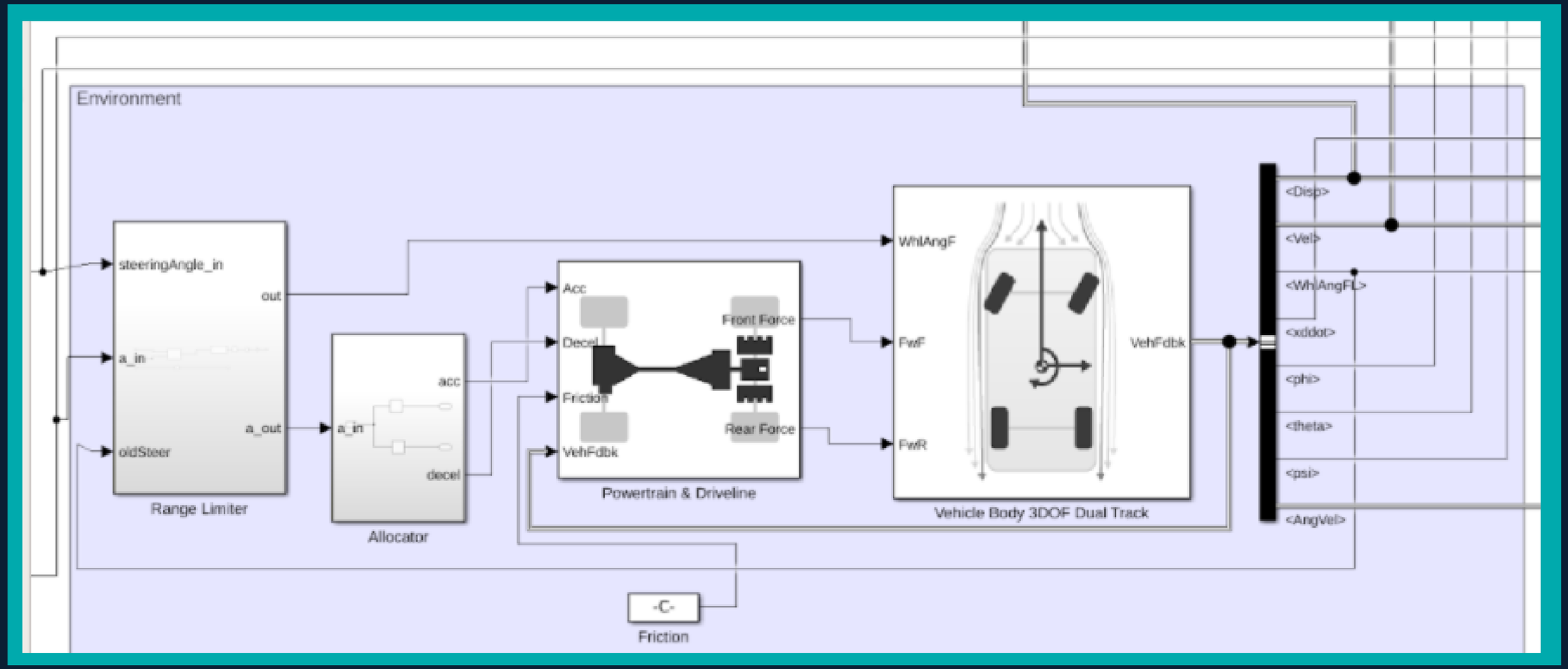
05 Creating the RL DDPG Agent

06 Simulating the Model and RL agent

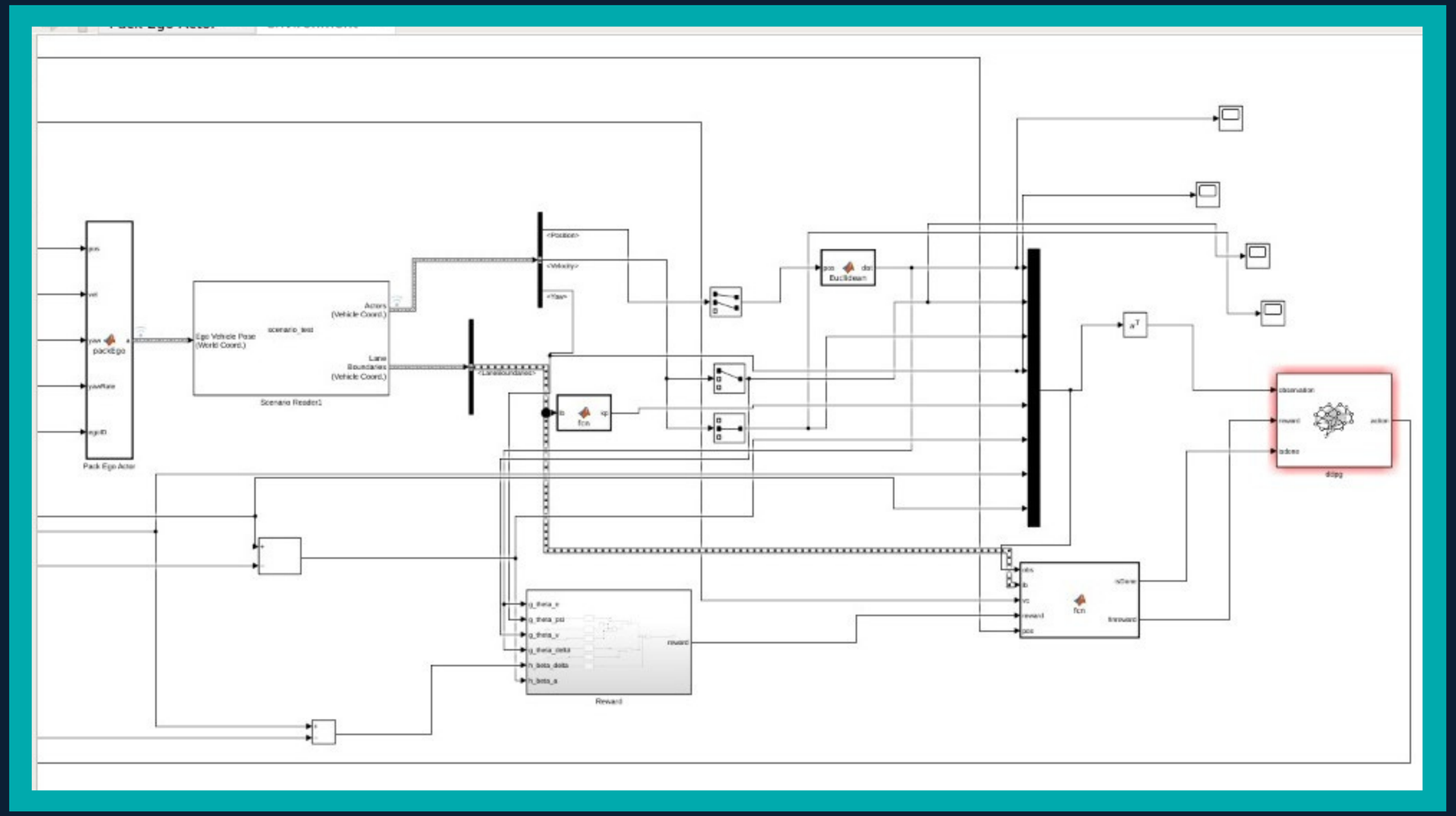


SYSTEM BLOCK DIAGRAM

OUR ENVIRONMENT



OUR ENVIRONMENT CONTD.

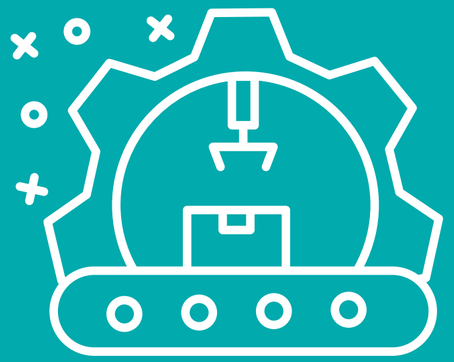


ENGINE

A simple engine block from the Vehicle Dynamics Blockset was used to model the engine. The engine speed at an instant was decided based on the gear ratio and wheel speed, and the maximum torque at that rpm was decided using the torque curve.
After calculating the produced torque to be sent to the wheels, it was distributed equally among the 4 wheels (all-wheel drive system). A variable first-order system and time delay were implemented as described in the paper to model rising and falling responses.

GEARBOX

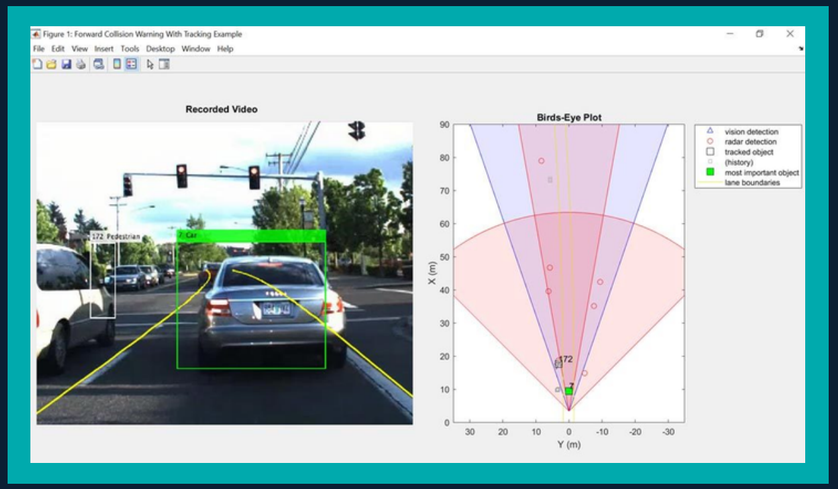
An 8 speed ZF automatic gearbox logic was implemented. The gear ratios for each gear, along with a final drive factor of 3.1 were used to calculate wheel torque for a given engine torque. Gear changes were decided based on the current vehicle velocity, and the cutoff points for each gear was set at engine rpm 2500 (beyond the peak torque value)



VEHICLE DYNAMICS BLOCKSET

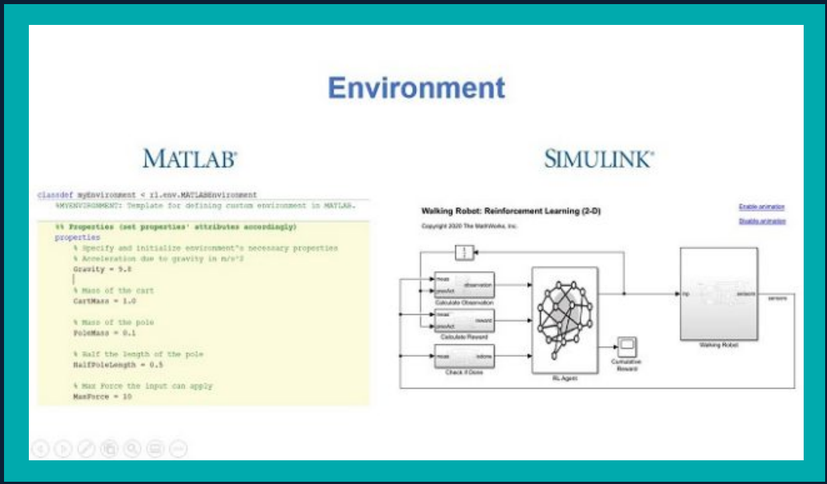
Vehicle Dynamics Blockset™ provides fully assembled reference application models that simulate driving maneuvers in a 3D environment. One can use the prebuilt scenes to visualize roads, traffic signs, trees, buildings, and other objects around the vehicle. One can customize the reference models by using their own data or by replacing a subsystem with your own model. The blockset includes a library of components for modeling propulsion, steering, suspension, vehicle bodies, brakes, and tires.

AUTONOMOUS DRIVING TOOLBOX



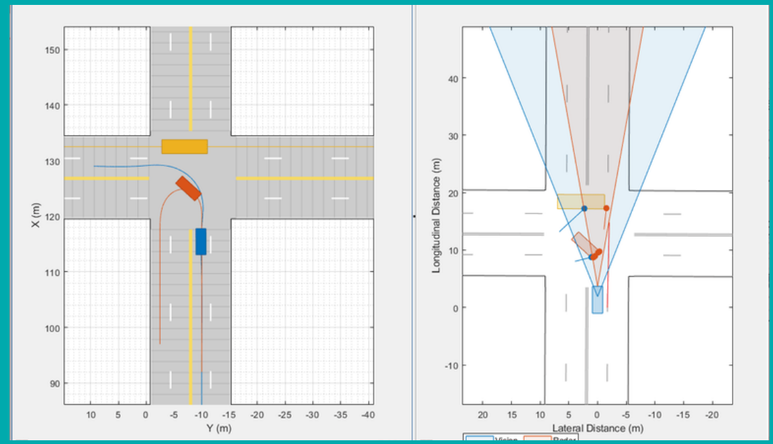
Automated Driving Toolbox™ provides algorithms and tools for designing, simulating, and testing ADAS and autonomous driving systems. One can design and test vision and lidar perception systems, as well as sensor fusion, path planning, and vehicle controllers. Visualization tools include a bird’s-eye-view plot and scope for sensor coverage, detections and tracks, and displays for video, lidar, and maps.

REINFORCEMENT LEARNING TOOLBOX



Reinforcement Learning Toolbox™ provides an app, functions, and a Simulink® block for training policies using reinforcement learning algorithms, including DQN, PPO, SAC, and DDPG. You can use these policies to implement controllers and decision-making algorithms for complex applications such as resource allocation, robotics, and autonomous systems.

DRIVING SCENARIO DESIGNER APP



The **Driving Scenario Designer** app enables you to design synthetic driving scenarios for testing your autonomous driving systems.
Using the app, you can create road and actor models using a drag-and-drop interface., configure vision, radar, lidar, and INS sensors mounted on the ego vehicle etc.

DRIVING SCENARIO DESIGNING

BASIC SCENARIO

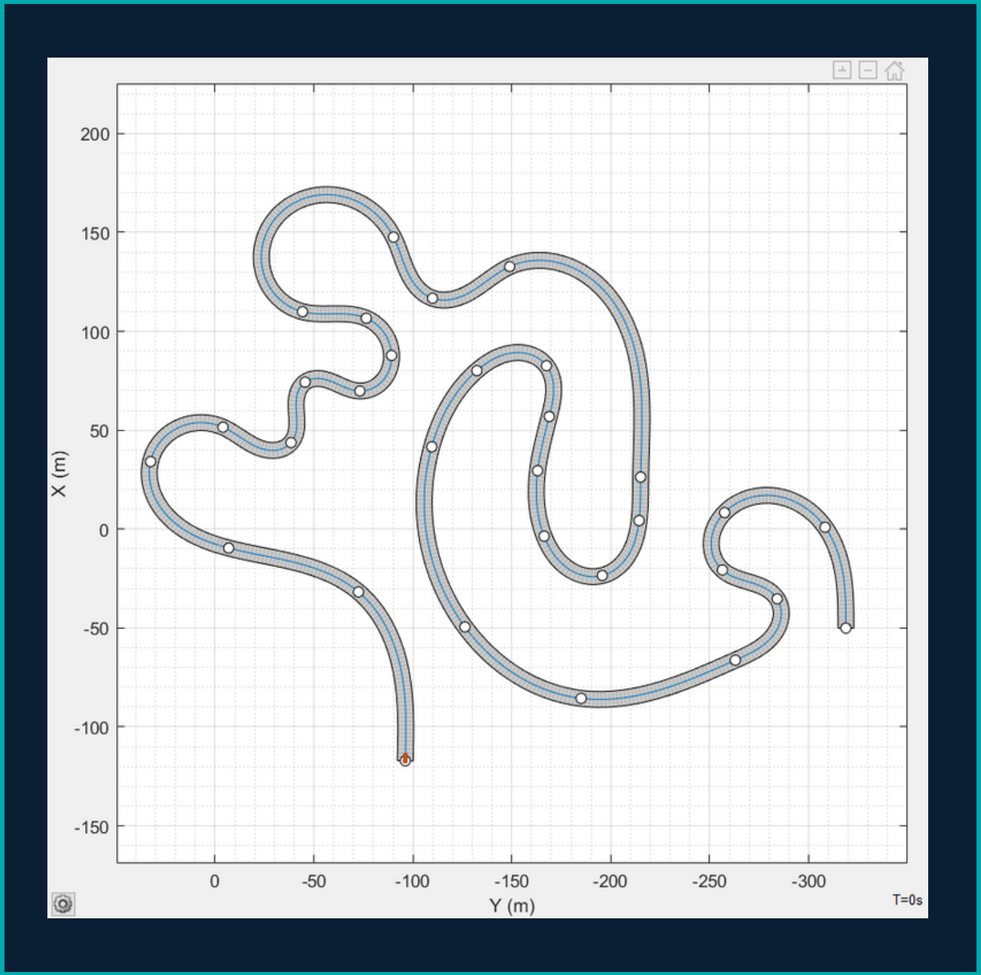
We used the Driving Scenario Designer App of MATLAB to design synthetic driving scenarios for testing our autonomous driving system. We first built our basic scenario by adding the roads and the ego vehicle and actors through the interface of the app. The path was built in order to include both left and right turns with different curvatures so as to train our model for different conditions.

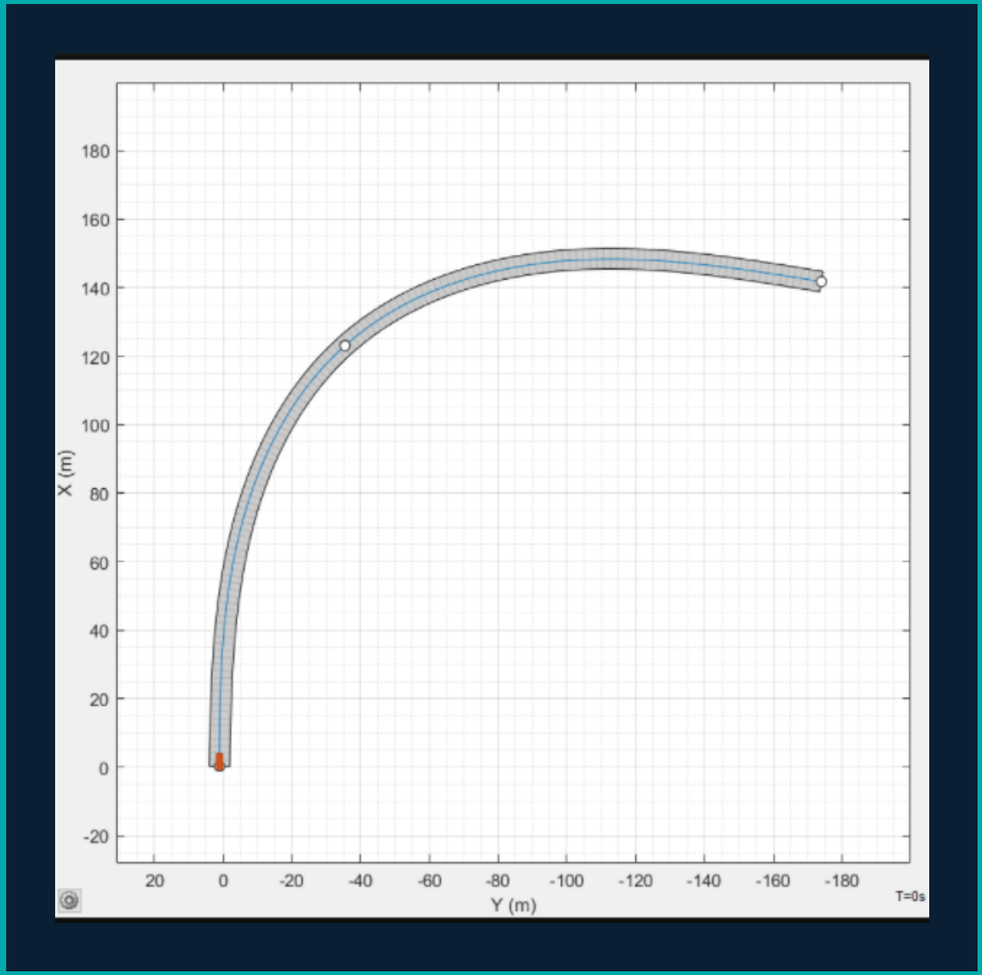
ALIGNMENT

After laying out the foundation of our path and vehicles, we exported the scenario file to a MATLAB function to make changes in our scenario programmatically. We had the ego vehicle and the actor aligned completely and the path of the actor was set to exactly coincide with the middle of the road.

EXPORTED AS A SIMULINK MODEL

The ego vehicle is then supposed to be commanded by our algorithm whereas the actor following the set path represents the ego vehicle in an ideal scenario. The new scenario was then exported as a Simulink model as the scenario reader block which took ego vehicle pose as the input and gave the output in the form of actor vehicle pose and the scenario lane boundaries.





DRIVING SCENARIO

REWARD FUNCTION

The reward function should be constructed such that maximizing the reward yields good control performance. In our implementation, we have 3 kinds of errors namely, the cross-track error, the velocity error and the orientation error.

01 The cross-track error is defined as the difference between desired lateral position and the lateral position of the car, denoted in the path frame.

02 The velocity error is defined as the difference between the desired velocity vd in path x-direction and the velocity of the car in x-direction, denoted in the path frame.

03 The orientation error is defined as the angle between the path frame and the car frame.

The reward function is chosen as follows:

$$r_{tot} = r_e + r_{\Delta\delta} + r_{\Delta a}$$

To construct the reward function we also make use of two auxiliary functions. The first is a Gaussian-like function g (x), which is defined as follows:

$$g_\theta(x) = \theta_1 \, e^{\frac{-x^2}{2\theta_2}}$$

The second is a dead zone-based function h**β**(x)

$$h_\beta(x) = \begin{cases} 0, & \text{if } |x| < \beta_1 \\ -\beta_2 \cdot |x|, & \text{else} \end{cases}.$$

Calculating each of the rewards:

$$r_e\left(e_y^{\mathrm{P}}, e_\psi^{\mathrm{P}}, e_{v_x}^{\mathrm{P}}\right) := g_{\theta_e}\left(e_y^{\mathrm{P}}\right)\left(1 + g_{\theta_\psi}\left(e_\psi^{\mathrm{P}}\right) + g_{\theta_v}\left(e_{v_x}^{\mathrm{P}}\right)\right).$$

$$r_{\Delta\delta}\left(e_y^{\mathrm{P}}, \Delta\delta\right) := g_{\theta_{\Delta\delta}}\left(e_y^{\mathrm{P}}\right) h_{\beta_{\Delta\delta}}(\Delta\delta).$$

$$r_{\Delta a_d}(\Delta a_d) := h_{\beta_{\Delta a}}(\Delta a_d)$$

Finally, we sum up all the rewards calculated and feed this reward as an input to the RL agent

DDPG

Q-Learning & Policy Gradients

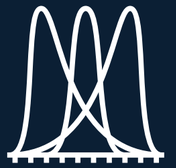
Deep Deterministic Policy Gradient (DDPG) is a reinforcement learning technique that combines both Q-learning and Policy gradients.

Actor-Critic Technique

DDPG being an actor-critic technique consists of two models: Actor and Critic. The actor is a policy network that takes the state as input and outputs the exact action (continuous).

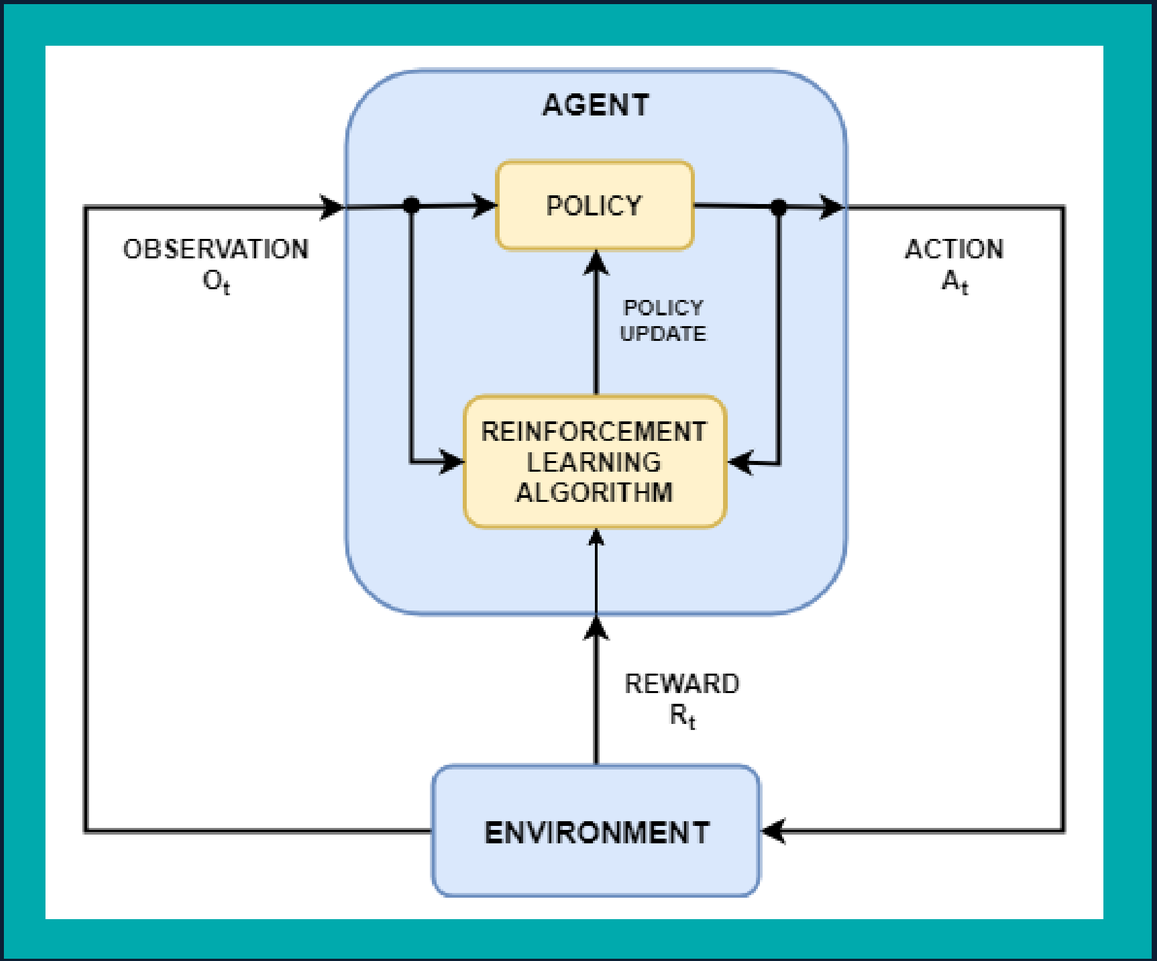
Q-value network

The critic is a Q-value network that takes in state and action as input and outputs the Q-value.



Continuous Action Setting

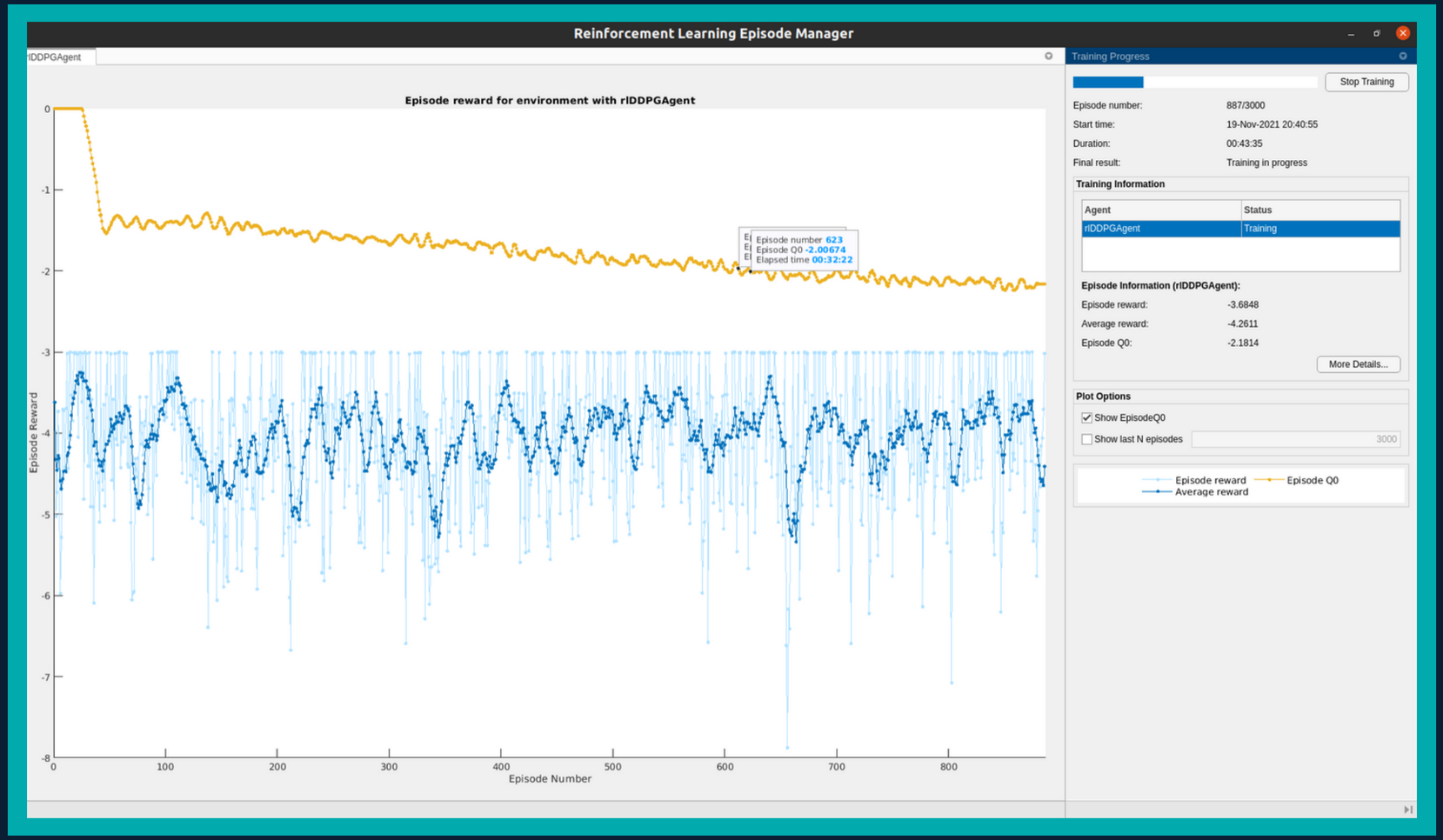
DDPG is used in the continuous action setting and the “deterministic” in DDPG refers to the fact that the actor computes the action directly instead of a probability distribution over a discrete action space.



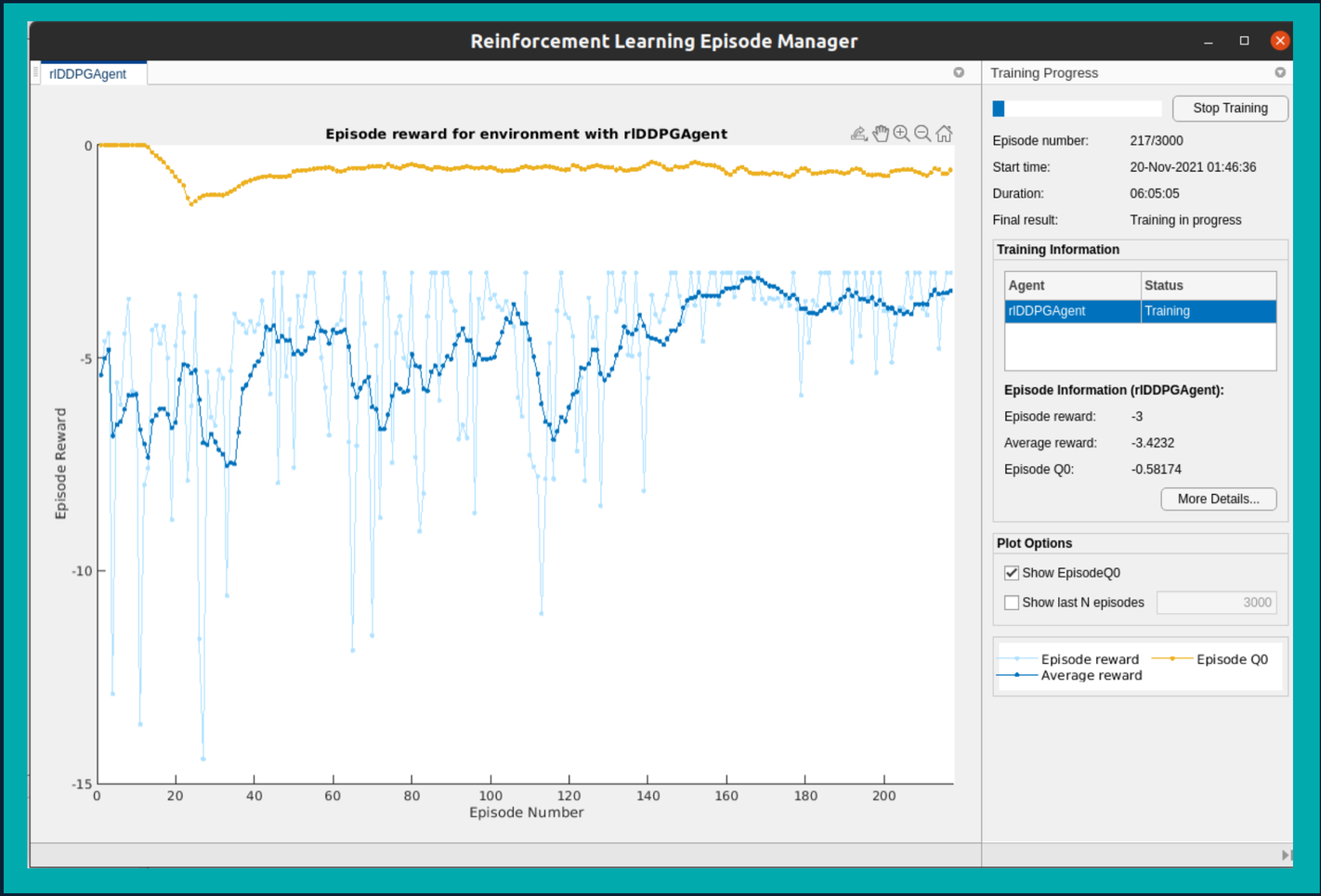
We use the following specifics while building our DDPG agents:

- Steps per episode=80
- maxepisodes = 3000
- Sampling time = 0.5
- TargetSmoothFactor = 1e-3
- DiscountFactor = 0.98
- MiniBatchSize for training = 64
- Experience buffer size = 1e6

OBSERVATIONS - CURVED TRACK



OBSERVATIONS - SIMPLE PATH



Creating the Vehicle Dynamics:
Srihas, Archisman, Hrushitha, Manishkha, Swarnava

Creating the Observations and Errors:
Srihas, Archisman, Swarnava, Ananya, Bhargav

Creating the RL Agent:
Rounak, Aneeta, Swarnava

Creating the Driving Scenarios:
Swarnava, Ananya, Devraj, Chandana

Presentation:
Everyone

THANK YOU