Simulation Manual.

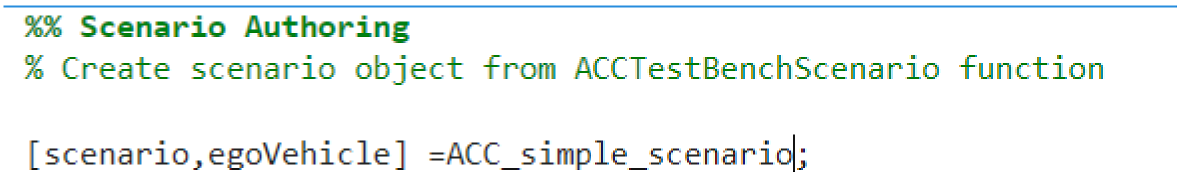
1. Introduction

In this manual I will be explaining how to train a Reinforcement Learning agent in Matlab Simulink , how to perform hyperparameter tuning using the Matalb Experiment Manager and how to perform environmental robustness testing on the trained RL agent. The MATLAB version used in this manual is R2022b.

2. Training a RL Agent

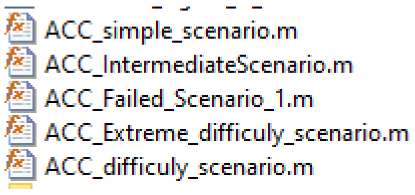
2.1. Load in ACC model parameter

First you need to open the *"ACCTestBenchExample.slx"* file which contains the complete reinforcement learning setup. Then run the *"helperACCSetUp.m,"* file, which loads all the variables required for the ACC model. These variables include constants for the car bicycle model, the buses used for radar and vision detection, the scenario, the starting positions of the cars and other parameters needed to execute the ACC model. If you wish to modify the training scenario for the agent, you can change the specific line of code to a different scenario (see Figure 1). There are various pre-defined scenarios available (See Figure 2), including the ones discussed in the thesis. Furthermore, if you prefer to design your own scenario, you can accomplish this using the Driving Scenario Designer, which is accessible in the Apps section of Matlab.



```
%% Scenario Authoring
% Create scenario object from ACCTestBenchScenario function

[scenario,egoVehicle] =ACC_simple_scenario;
```

Figure 1 Load in Training Scenario



ACC_simple_scenario.m
ACC_IntermediateScenario.m
ACC_Failed_Scenario_1.m
ACC_Extreme_difficuly_scenario.m
ACC_difficuly_scenario.m

Figure 2 Available Scenarios

2.2. Implement RL block in ACC model

Once this file is executed, we can proceed to incorporate a RL agent into the model. To do this use the Reinforcement Learning Block from the Simulink library. Additionally, you will need to create a custom reward function that encourages the agent to exhibit specific behaviors. Furthermore, you must design a custom "IsDone" block that indicates to the RL agent when it can conclude the learning episode. In the RL block, the observation signals are expected to be in the form of a vector signal. This implies that if you have multiple observation signals, you need to combine them into a vector format using a MUX block, which is available in the Simulink Library. If you are utilizing my portfolio's Matlab/Simulink files, the observation signals are already correctly linked to a MUX block within the Agent A and Agent S blocks. Besides the merging of observation signals, these two blocks include the reward functions from Agent A and Agent S, as discussed in the thesis. If you intend to train using the Agent A reward function, establish the connection by linking the observation signals to the Agent A block. Subsequently, connect the observation and reward signals from the Agent A block to the corresponding observation and reward signals of the RL block. Similarly, if you wish to utilize the Agent S reward function, follow the same procedure by connecting the Agent S block in a similar manner. If you want to change the episode termination conditions, you can change these by altering the logic within the "isDone" block. The complete setup can be seen in Figure 3.

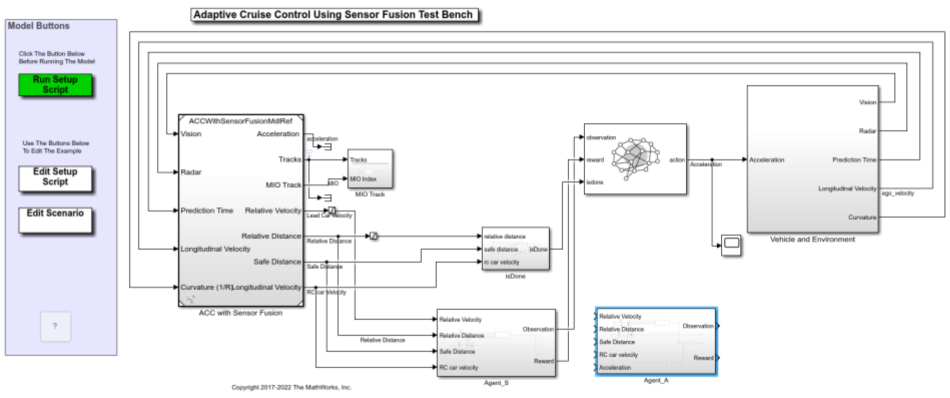
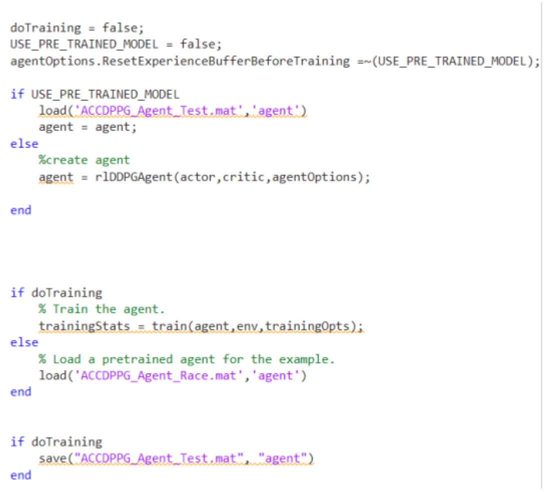


Figure 3 Reinforcement Learning Setup in Simulink

2.3. Perform RL training

When the agent is setup within the ACC model you can open the *"agentsetupDDPG.m "* file which sets up the DDPG agent. It configures the actor-critic network, training options, and hyperparameters. If you wish to load a trained agent, ensure that the boolean variable *"doTraining"* is set to false and provide the correct file name of the trained agent's .mat file. Conversely, if you want to train a new agent, set *"doTraining"* to true. If you intend to load the experience of a previously trained agent, set the *"USE_PRE_RAINED_AGENT"* boolean to true and provide the appropriate .mat file name of the pre-trained agent (See Figure 4).

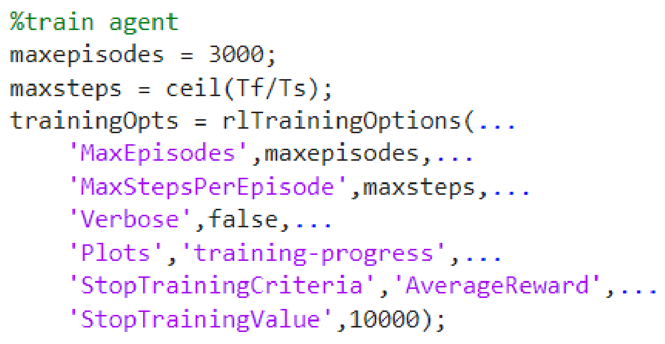


```
doTraining = false;
USE_PRE_TRAINED_MODEL = false;
agentOptions.ResetExperienceBufferBeforeTraining =~(USE_PRE_TRAINED_MODEL);

if USE_PRE_TRAINED_MODEL
    load('ACCDPPG_Agent_Test.mat','agent')
    agent = agent;
else
    %create agent
    agent = rlDDPGAgent(actor,critic,agentOptions);

end



if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load a pretrained agent for the example.
    load('ACCDPPG_Agent_Race.mat','agent')
end


if doTraining
    save("ACCDPPG_Agent_Test.mat", "agent")
end
```

Figure 4 RL training execution code

To regulate the duration of the training process, we can utilize the "StopTrainingCriteria" parameter. This option allows us to define specific conditions under which the training should be halted. For instance, we can set the training to stop once a certain number of episodes have been completed, or we can specify a target average reward value that, once reached, triggers the termination of training. You can find detailed information about all the available "StopTrainingCriteria" options in the following link: https://nl.mathworks.com/help/reinforcement-learning/ref/rl.option.rltrainingoptions.html.

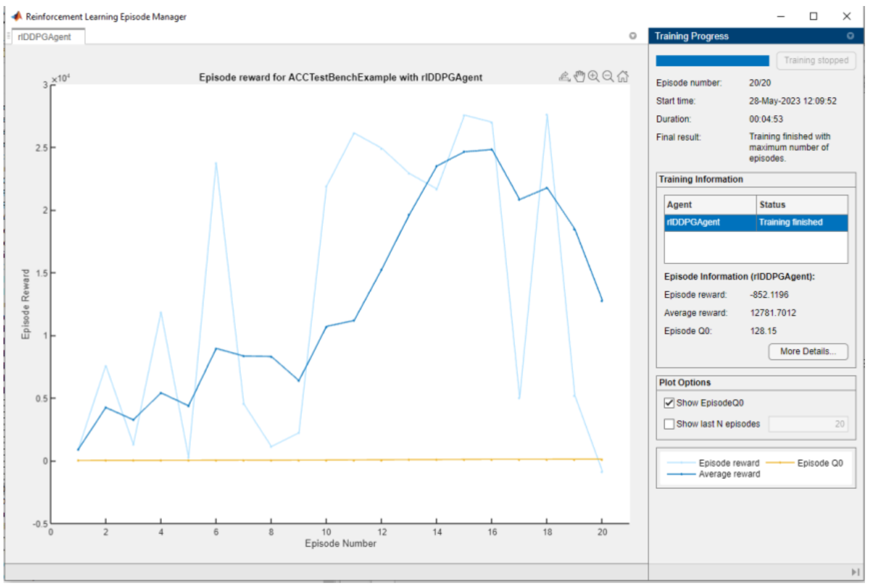
As a example, I wanted the learning process to be terminated when the average reward reaches 10,000 (See Figure 5).



```
%train agent
maxepisodes = 3000;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',10000);
```

Figure 5 RL training options

When the training of the RL agent has successfully started you will see the "RL Episode Manager" appear (See Figure 6), which will help you track the progress of the training of your RL agent. If you want additional information regarding observation signals or acceleration actions during training, you can connect these signals to a scope block from the Simulink Library. To visualize the driving behavior of your car (See Figure 8) you can open the Bird's-Eye Scope located in the Simulation tab (See Figure 7).



Episode number: 20/20
Start time: 28-May-2023 12:09:52
Duration: 00:04:53
Final result: Training finished with maximum number of episodes.

Training Information

Agent	Status
rlDDPGAgent	Training finished

Episode Information (rlDDPGAgent):
Episode reward: -852.1196
Average reward: 12781.7012
Episode Q0: 128.15

Figure 6 RL Episode Manager

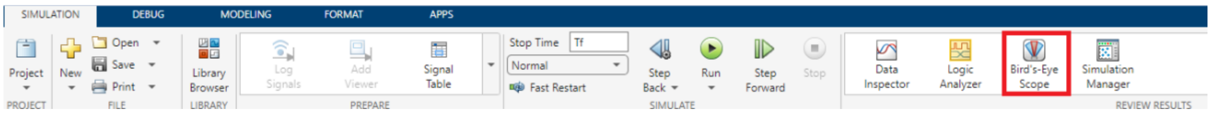


Figure 7 Bird's-Eye Scope Location

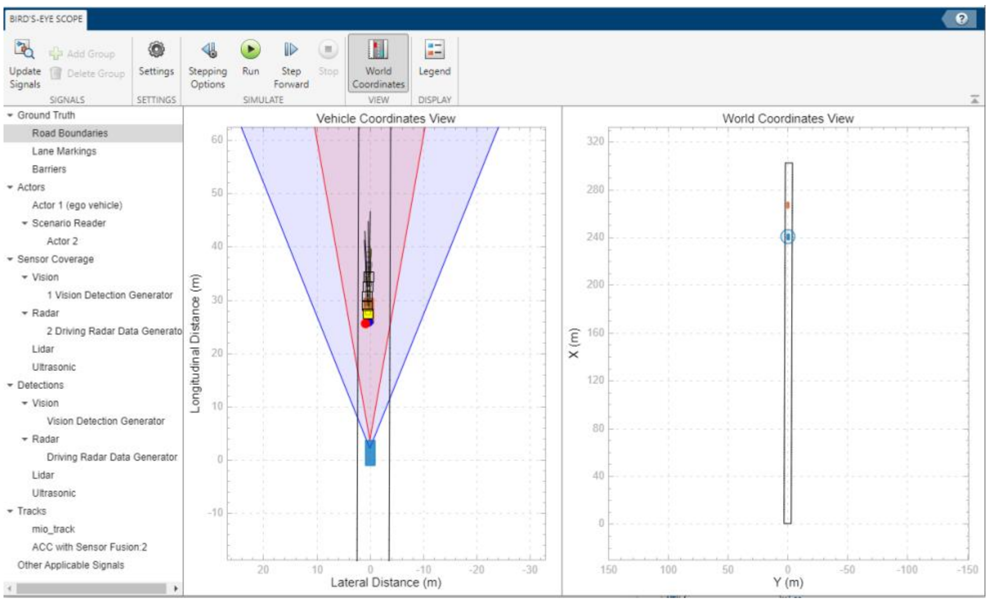


Figure 8 Bird's-Eye Scope

3. Hyperparameter tuning

To fine-tune hyperparameters, you can utilize the Experiment Manager App located in the Apps section of Matlab. After opening this app, you can create a project and develop a custom training function. Fortunately, I have already completed this step. By clicking the "Open" button within the Experiment Manager, you can select the ACCHyperParameterTuning.prj file.

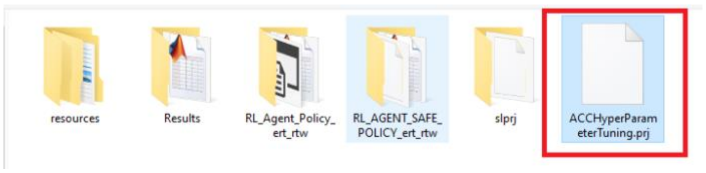
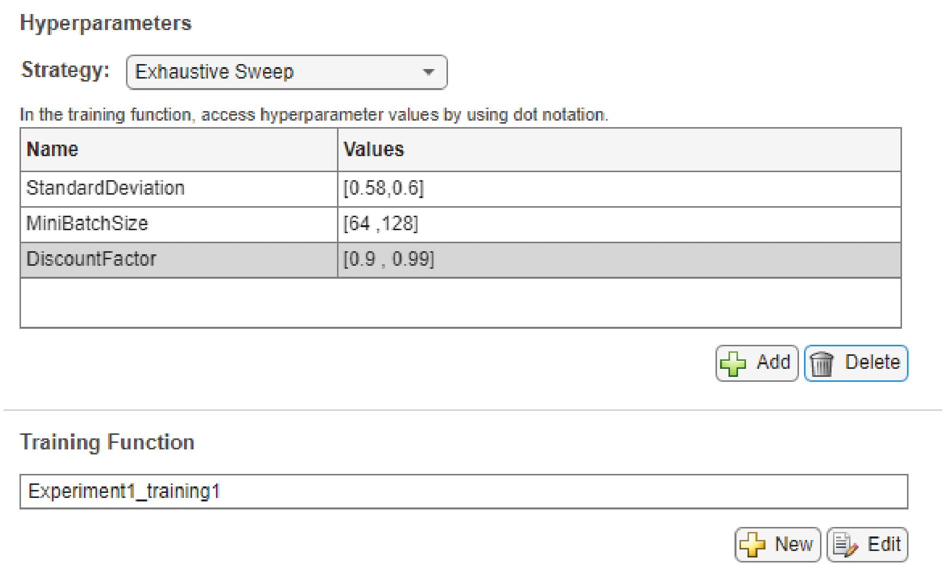


Figure 9 ACCHyperParameterTuning.prj file

When you open the .prj file, the Experiment Manager will appear as a pop-up window. In the Hyperparameter section of the Experiment Manager, you have the option to add or remove hyperparameters that require fine-tuning. Additionally, you can specify the values that you want the Experiment Manager to test for you (See Figure 10). There are various strategies available for Hyperparameter tuning, but in this case, I have only utilized the Exhaustive Sweep Strategy. This strategy systematically tests all possible combinations of the provided hyperparameter values. For example, if you have three parameters to tune, each with two values, you will have a total of 8 combinations and 8 trials. During a trial, a specific combination of hyperparameters is tested with your Reinforcement Learning agent.



Hyperparameters

Strategy: Exhaustive Sweep

In the training function, access hyperparameter values by using dot notation.

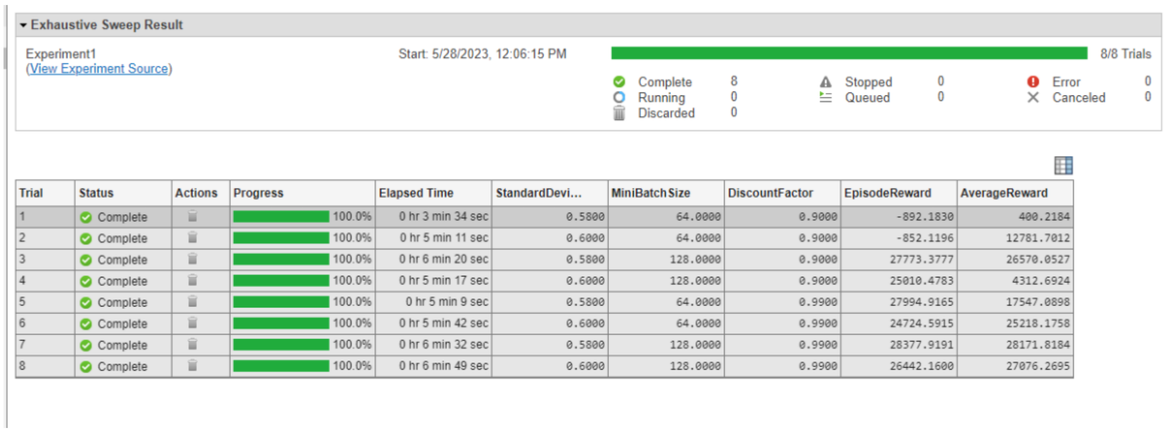
Name	Values
StandardDeviation	[0.58,0.6]
MiniBatchSize	[64 ,128]
DiscountFactor	[0.9 , 0.99]

Training Function

Experiment1_training1

Figure 10 Experiment Manager

As an example, we conducted a hyperparameter tuning session, focusing on tuning three hyperparameters with two values each. Out of the multiple trials, Trial 7 demonstrated the highest average reward, indicating that the specific set of hyperparameters used in that trial yielded the best performance. It is these hyperparameters that we intend to employ. The EpisodeReward column tracks the reward obtained in each episode.



▾ Exhaustive Sweep Result

Experiment1
(View Experiment Source)

Start: 5/28/2023, 12:06:15 PM

8/8 Trials

Complete	8	Stopped	0	Error	0
Running	0	Queued	0	Canceled	0
Discarded	0				

Trial	Status	Actions	Progress	Elapsed Time	StandardDevi...	MiniBatch Size	DiscountFactor	EpisodeReward	AverageReward
1	✓ Complete		100.0%	0 hr 3 min 34 sec	0.5800	64.0000	0.9000	-892.1830	400.2184
2	✓ Complete		100.0%	0 hr 5 min 11 sec	0.6000	64.0000	0.9000	-852.1196	12781.7012
3	✓ Complete		100.0%	0 hr 6 min 20 sec	0.5800	128.0000	0.9000	27773.3777	26570.0527
4	✓ Complete		100.0%	0 hr 5 min 17 sec	0.6000	128.0000	0.9000	25010.4783	4312.6924
5	✓ Complete		100.0%	0 hr 5 min 9 sec	0.5800	64.0000	0.9900	27994.9165	17547.0898
6	✓ Complete		100.0%	0 hr 5 min 42 sec	0.6000	64.0000	0.9900	24724.5915	25218.1758
7	✓ Complete		100.0%	0 hr 6 min 32 sec	0.5800	128.0000	0.9900	28377.9191	28171.8184
8	✓ Complete		100.0%	0 hr 6 min 49 sec	0.6000	128.0000	0.9900	26442.1600	27076.2695

Figure 11 Expirement manager trail results

3.1. How to add a hyperparameter?

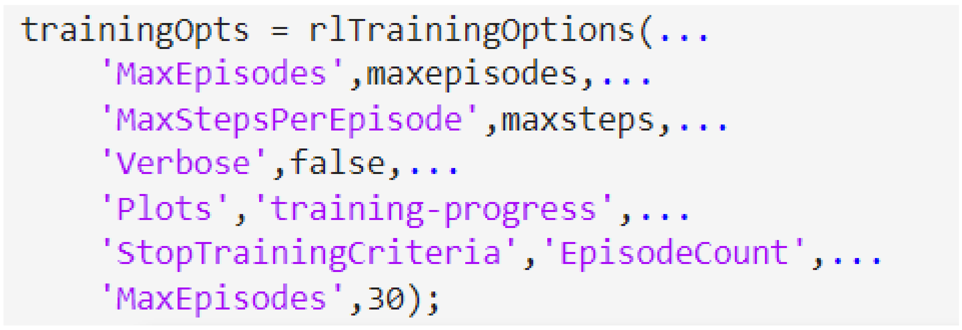
To add a hyperparameter, navigate to the Training Function, which can be found at the bottom of Figure 10. Clicking on "Edit" will open the training function in the MATLAB text editor. To incorporate a hyperparameter, simply replace the constant value of the hyperparameter with ***params.Hyperparametername***. As demonstrated in Figure 12, the hyperparameters already added to the Experiment Manager follow this format. After modifying the training function, save the changes and return to the Experiment Manager. In the Hyperparameters section, click on "Add" and specify the name of the hyperparameter (the name used after ***params.***). Then, enclose the desired values within brackets, separating each value with a comma.

```
%specify agent options
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',params.DiscountFactor,...
    'MiniBatchSize',params.MiniBatchSize);

agentOptions.NoiseOptions.StandardDeviation = params.StandardDeviation;
agentOptions.NoiseOptions.StandardDeviationDecayRate = 1e-5;
```

Figure 12 Add Hyperparameter to Experiment Manager

3.2. How to control length of a Trial?

To determine the duration of each trial in the Experiment Manager, you need to edit the Training Function once again. Within the training function, navigate to the training options and modify the 'StopTrainingCriteria' parameter. This adjustment is similar to controlling the training duration in regular RL training, but instead of modifying it in the agentsetupDDPG.m file, you make the change within the *Experiment1_training1.mlx* file (Training Function) using the edit button. As shown, I have set the 'StopTrainingCriteria' to an episode count of 30. If you wish to extend the training duration, such as to 100 episodes per trial, simply adjust the value from 30 to 100. Now each trial will run 100 episodes.

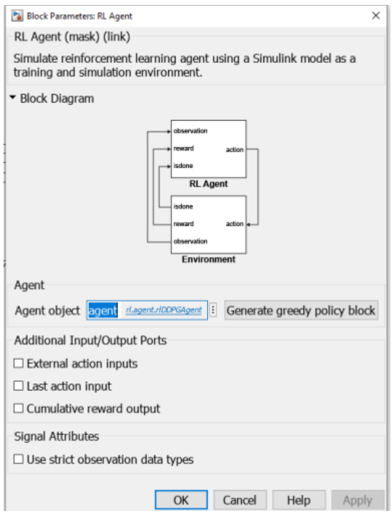


```
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeCount',...
    'MaxEpisodes',30);
```

Figure 13 Change Training duration of eacht Trial

4. Environmental Robustness Testing.

Once you have trained a RL agent, you can double-click on the RL block, which provides you with the option to generate a greedy policy block based on your trained RL agent. This generated greedy policy block can then be utilized in the *ACCTestBenchPolicy.slx* file, which serves as the testing setup.


Block Parameters: RL Agent

RL Agent (mask) (link)

Simulate reinforcement learning agent using a Simulink model as a training and simulation environment.

Block Diagram

Agent

Agent object: agent — rl.agent.rlDDPGAgent — Generate greedy policy block

Additional Input/Output Ports

☐ External action inputs

☐ Last action input

☐ Cumulative reward output

Signal Attributes

☐ Use strict observation data types

Figure 14 Menu after double clicking RL block

It is worth noting that the IsDone block and reward function block, as used in the Reinforcement Learning setup, are not present in this context. However, you still need to employ a MUX block to create an observation signal vector using your observation signals. It is crucial to ensure that you connect the observation signals to the MUX block in the same manner as in the RL setup. Otherwise, the merging of the observation signals within the MUX block will be altered, potentially resulting in a deviation from the intended behavior of your RL agent.

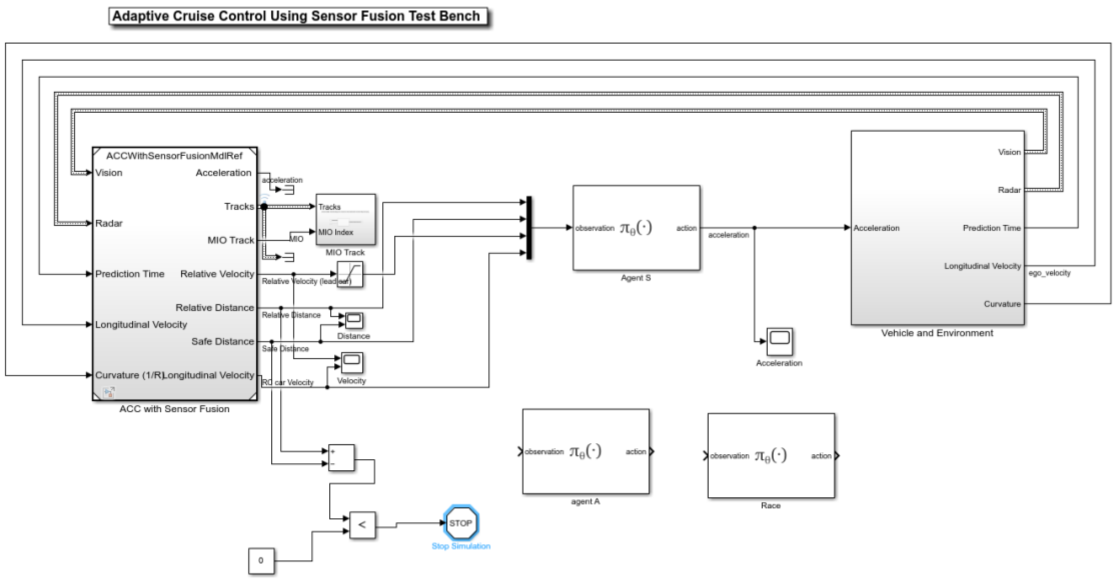


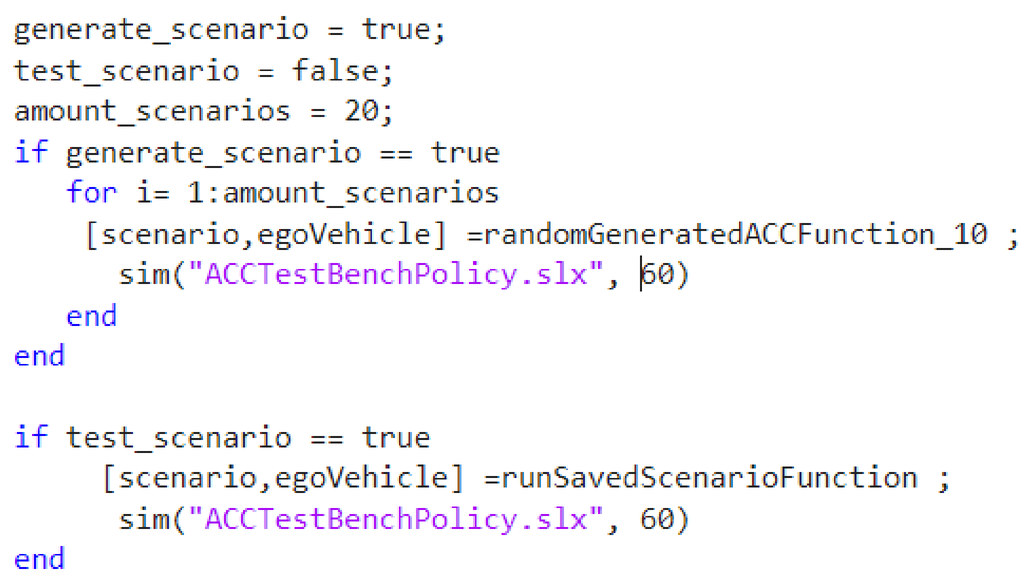
Figure 15 ACCTestBenchPolicy.slx Test Setup

In Figure 15, it is evident that I have already incorporated the policies of Agent_A and Agent_S (discussed in the thesis). Additionally, I have included a third policy for an RL agent specialized in racing. This particular agent drives very close to the lead car sometimes even crossing the safe distance. It is not a really safe agent but can surely be fun to use in F1Tenth races.

After connecting the desired policy for testing, you can open the *runTester.m* file. If you want to generate scenarios, set the *generate_scenario* boolean to true, and then run the tester. Before executing the runTester.m file, make sure to have the observation/action signals scopes and bird's-eye scopes open. If you wish to modify the number of scenarios generated by the tester, simply adjust the value of the *amount_scenarios* variable according to your desired quantity.

The runTester script generates a random scenario, loads it, and initiates the testing setup for a simulation period of 60 seconds. If you wish to modify the simulation duration, you can adjust the value of 60 to your preferred length. When running this file, you will observe the RL agent being tested in different scenarios. To determine whether the agent failed or succeeded in each scenario, I analyzed the relative and safe distances, and also monitored the Bird's-Eye Scope.

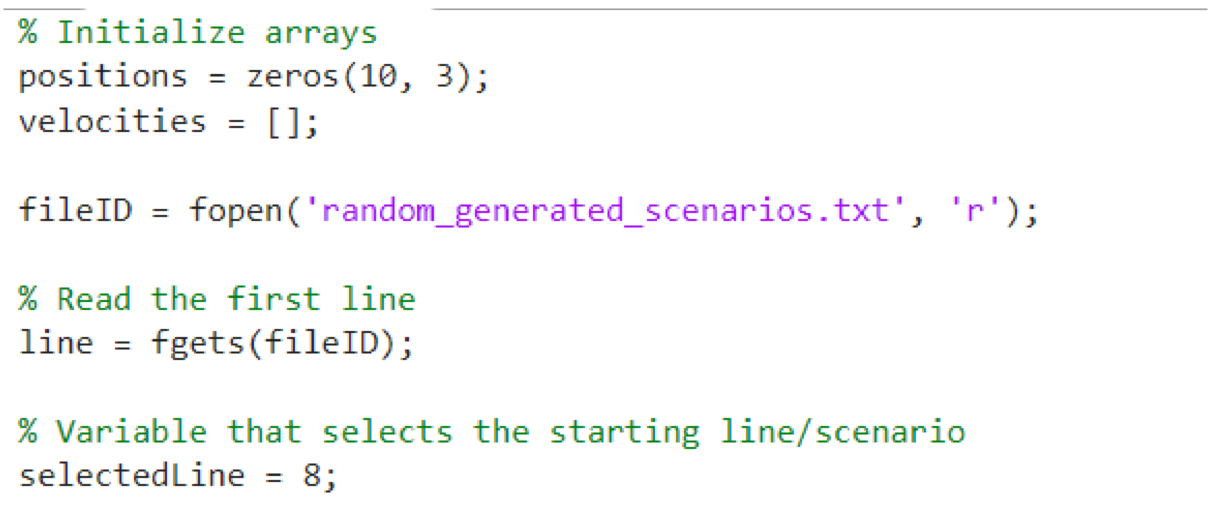
In the event of a safety violation, I manually halted the runTester program. It is worth noting that this approach is not particularly efficient for environmental robustness testing, especially when dealing with a large number of scenarios, as it requires constant manual monitoring. However, I chose this method to ensure that the trained RL agent successfully passed each scenario. In the future, the program can be made more efficient by implementing automation that eliminates the need for manual monitoring. As a step towards automation, I added a stop block (See Figure 15) in the testing setup to halt the simulation if the relative distance becomes smaller than the safe distance. This enabled me to detect even minor safety violations that may not have been visible in the scopes.



```
generate_scenario = true;
test_scenario = false;
amount_scenarios = 20;
if generate_scenario == true
   for i= 1:amount_scenarios
    [scenario,egoVehicle] =randomGeneratedACCFunction_10 ;
      sim("ACCTestBenchPolicy.slx", 60)
   end
end

if test_scenario == true
     [scenario,egoVehicle] =runSavedScenarioFunction ;
      sim("ACCTestBenchPolicy.slx", 60)
end
```

Figure 16 runTester.m Script

All the randomized scenarios are stored in the *random_generated_scenarios.txt* file. Additionally, I have implemented scripts that can read and load these scenarios for further testing. This allows for comparisons between different RL policies. To load a scenario from the text file, set the *generate_scenario* boolean to false and the *test_scenario* boolean to true. Then, navigate to the *runSavedScenarios.m* file (Figure 17) and modify the selectedLine parameter according to the scenario you wish to test. The value of the selectedLine parameter corresponds to the number assigned to the specific scenario you want to evaluate.



```
% Initialize arrays
positions = zeros(10, 3);
velocities = [];

fileID = fopen('random_generated_scenarios.txt', 'r');

% Read the first line
line = fgets(fileID);

% Variable that selects the starting line/scenario
selectedLine = 8;
```

Figure 17 runSavedScenarios.m

Once you have properly configured the boolean variables and selected the scenario you wish to load, simply execute the runTester.m file. This will run the testing setup using the specified scenario.