MANCHESTER 1824
The University of Manchester

AUTONOMOUS VEHICLE DRIFTING

Third Year Individual Project — Final Report

April 2019

Zadiq Oguntimehin

9769331

Supervised by:

Dr. Joaquin Carrasco

Last Updated: 09-04-2019

School of Electronic and Electrical Engineering

Abstract

Various control methods have been explored and applied in the field of robotics for autonomous control tasks. Some of these methods utilize hard-proven scientific knowledge to design algorithms and solutions that enables robots to perform required autonomy tasks. Robots utilizing such solutions are hard-coded to behave in certain ways under certain conditions. While the robots might perform well under these conditions, their performance are limited to the conditions foreseen by the implementer. Attempts to develop solutions that accommodate all possible conditions lead to complexities, making it rather unfeasible to manually develop such solutions. Another approach to finding control solutions for autonomy tasks is the use of data-driven algorithms. These algorithms extract information and features from historical behavioural data of the robot, attempting to detect patterns that correlate with the desired solutions. The use of historical data allows robots to apply "experience" when in an unfamiliar scenario. This project explores the implementation of a Reinforcement Learning (a data-driven approach) algorithm on dynamic models of a vehicle for optimal drift control. The Deep Deterministic Policy Gradient reinforcement learning algorithm was implemented to control 2D and 3D vehicle dynamic models to achieve drifting. Optimal drifting were achieved on the 2D models but not on the 3D models. However, some drifting behaviour were attained on the 3D models which were further analysed.

Keywords

reinforcement-learning, autonomous-control, simulation, dynamic-modelling, deep-learning.

Acknowledgement

I would like to show appreciations to my supervisor, Dr. Joaquin Carrasco, for his support, guidance and encouragement throughout the phase of the project. I would also like to thank my parents, especially my mother, for their parental support throughout the phase of my studies at the University of Manchester.

Nomenclature

Abbreviations	Descriptions
2D	Two Dimension
3D	Three Dimension
RL	Reinforcement Learning
DDMR	Differential Drive Mobile Robots
2WD	Two-Wheel Drive
4WD	Four-Wheel Drive
PMF	Pacejka's Magic Formula
RCV	Radio Controlled Vehicle
DDPG	Deep Deterministic Policy Gradient
DL	Deep Learning
DNN	Deep Neural Network
DQN	Deep Q-Learning
COG	Center of Gravity
URDF	Unified Robot Description Format
DPG	Deterministic Policy Gradient
RCV	Radio Controlled Vehicle
MSE	Mean Squared Error
LR	Learning Rate

Symbols	Description	Units
Λ_2	2D dynamic models	[—]
$\dot{\mathrm{X}}$	Longitudinal velocity in reference to the global frame	$[\mathrm{ms}^{-1}]$
$\dot{\mathrm{Y}}$	Lateral velocity in reference to the global frame	$[\mathrm{ms}^{-1}]$
$\dot{v}_x$	Longitudinal acceleration a vehicle in reference to its frame	$[\mathrm{ms}^{-2}]$
$\dot{v}_y$	Lateral acceleration a vehicle in reference to its frame	$[\mathrm{ms}^{-2}]$
$\ddot{\varphi}$	Yaw acceleration a vehicle around it's centre of mass	$[\mathrm{rads}^{-2}]$
$f_{R,x}$	Rear wheel longitudinal frictional force	[N]
$f_{F,y}$	Fore wheel lateral frictional force	[N]
$f_{R,y}$	Rear wheel lateral frictional force	[N]

d	Duty cycle	[—]
C_{m1}	2D vehicle's modelled motor constant	[—]
C_{m2}	2D vehicle's modelled motor constant	[—]
C_r	2D vehicle's modelled motor constant	[—]
C_d	2D vehicle's modelled motor constant	[—]
D	Pacejka's Magic Formula constant	[—]
C	Pacejka's Magic Formula constant	[—]
B	Pacejka's Magic Formula constant	[—]
δ	Steering angle	[rad]
m	2D vehicle's mass	[kg]
l_f	Distance from fore wheel of a 2D vehicle to it's C.O.G	[m]
l_r	Distance from rear wheel of a 2D vehicle to it's C.O.G	[m]
I_z	2D vehicle's moment of inertia	[kg m^2]
R	2D vehicle's rear-wheel	[—]
F	2D vehicle's fore-wheel	[—]
r_d	Drift radius	[m]
r_r	Drift reward	[—]
Λ_3	3D dynamic models	[—]
μ_1	Frictional coefficient of a wheel in x-axis	[—]
μ_2	Frictional coefficient of a wheel in y-axis	[—]
θ_t	Tilt angle of a vehicle	[rad]
s_t	Time step states	[—]
s_d	Desired state	[—]
δ_2	2D vehicle's steering	[rad]
δ_3	3D vehicle's steering	[rad]
d_2	Duty cycle of 2D vehicles	[—]
d_3	Acceleration control of 3D vehicles	[ms^{-2}]
$\Lambda_{e,k}$	A vehicle dynamic model with a unique dynamic configuration	[—]
r_t	Time step reward	[—]
$\Re$	Reward function	[—]
σ	Reward function's sensitivity factor	[—]
r_T	Total sum of time step reward acquired in an episode	[—]

T	Total length of an episode	[s]
t	Time step	[s]
$\mathcal{A}$	Actor network	[—]
Q	Critic network	[—]
a_t	Time step action	[rad]
$\theta^{\mathcal{A}}$	Actor's parameters	[—]
θ^{Q}	Critic's parameters	[—]
q_t	Time step Q-value	[—]
π	Policy	[—]
γ	Discount Factor	[—]
L	Critic's network loss function	[—]
τ	Parameter update weights	[—]
$\mathfrak{B}$	Replay buffer	[—]
b_s	Mini-batch size	[—]
$\mathbb{n}$	Action noise	[—]
$\sigma_{\mathbb{n}}$	Action noise centre point parameter	[—]
$\mathcal{A}^{'}$	Target actor or DDPG model	[—]
$Q^{'}$	Target critic	[—]
$\alpha_{\mathcal{A}}$	Actor's learning rate	[—]
α_{Q}	Critic's learning rate	[—]
$u_{n,1}$	First hidden layer unit size	[—]
$u_{n,2}$	Second hidden layer unit size	[—]
j	Episode	[—]
$r_{T,inf}$	Total sum of time step reward acquired in an inference episode	[—]

Contents

Content's Word Count: 10, 686

1 Introduction

Autonomous control is an important and interesting topic in the field of robotics. It is increasingly becoming necessary that robots should have certain degrees of autonomy. This is essential not only for efficiency both also for safety measures. Take a vehicle for example, it is currently a norm to have a human manually controlling the vehicle. Under normal circumstances, humans are 'sufficient' controllers for controlling a vehicle. However given a scenario where the vehicle becomes uncontrollable due to physical changes in the environment, the safety of the human is at risk. An example of physical change in environment is when the road becomes slippery due to snowfall. An apparent solution to handling the risk imposed by a slippery road is to design a controller that aids the driver when navigating such roads. The solution can be implemented using knowledge-based control algorithms that are scientifically proven to work in certain scenarios. The problem with such type of solutions is that the design must be robust to accommodate various level of road slipperiness which requires high design complexity. An alternative approach is to use algorithms that can apply experience in situations that has been not previously encountered or defined. An example of such algorithm is the Reinforcement Learning (RL) algorithm.

This project explores the use of a RL algorithm to develop a controller that autonomously controls dynamic models of a vehicle to achieve drifting. By achieving drifting, the controller can also be applied for controlled navigation of a slippery road. Some implementations of RL algorithms are briefly discussed in Literature Review but the RL algorithm implemented in this project is Deep Deterministic Policy Gradient (DDPG) algorithm. While also implementing control algorithms, 2D and 3D dynamic models of a vehicle are also implemented for control experiments. Some analysis are also performed to analyse the performance and robustness of the control algorithms.

The main aim of this project is to demonstrate how a data-based algorithm such as RL can be used for autonomous task control. This demonstration can motivate further investigation of implementing the algorithm in other control problems.

The project delivers a successful implementation of a DDPG algorithm that achieves optimal drifting on 2D dynamic models. Other deliverables include a 2D and 3D implementation of vehicle dynamic models. While the DDPG algorithm could not achieve

optimal drifting on the 3D models, analysis are performed to highlight the reasons and what further steps can be taken to achieve drifting.

To the best of knowledge, this project is the first to implement DDPG for autonomous drifting control.

2 Literature Review

This review explores different methods for modelling a vehicle and various RL algorithms. Similar works involving optimal drift control are also discussed. Note that the words robot, vehicle and dynamic model are interchangeably synonyms used in this report.

2.1 Vehicle Modelling

It is important that the implementation of the vehicle's model entails enough information to infer near-reality behaviours and dynamics of a real-world vehicle. These behaviours and dynamics are important for substantial feature extraction by the control algorithm to successfully achieve drifting. It is also important that the modelled vehicle is able to drift (i.e. there exists a solution for the control algorithm to find).

While a dynamic model of a vehicle is used in this project, kinematic modelling is first explored to understand the fundamental concepts of modelling a mobile robot.

2.1.1 Kinematic Modelling

Kinematic modelling provides information about the motion and structure of the vehicle without taking into consideration the causes of such motions.

The first approach in kinematic modelling is the structural definition of the vehicle. This involves the definition of wheels and bodies of the vehicle. Different approaches have been taken in defining a vehicle's structure depending on the required motion and complexity of control. A commonly used structural type of mobile robot is the Differential Drive Mobile Robots (DDMR). The structural representations of DDMRs usually include two independently controlled side wheels and a front or back free-turning wheel [1], [2]. While the structural representation of a DDMR is relatively simple, there exist a high control complexity for drifting tasks. It is important that the steering control of the robot is utmost independent of the velocity control for simpler drift control. DDMR steering and velocity are both controlled by the direction and speed of the two independent wheels.

Alternative structural setups of mobile robots are the two-wheels and four-wheels non-differential drive configurations, both of which include four wheels linked to the body. The velocity of the robot is controlled by two wheels for a two-wheel drive (2WD) and four wheels for a four-wheel drive (4WD) while the steering is usually controlled by two wheels for both configurations. The separation in velocity and steering control provided by 2WD and 4WD configurations make them a suitable configuration for drifting control.

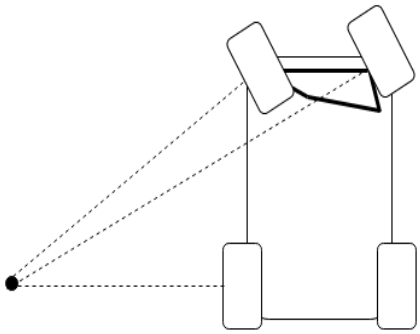

Figure 2.1: Ackermann's steering configuration on a four-wheeled vehicle.

Vehicle's steering wheels configuration are also important in defining its motion. Various geometric configurations of steering wheels have been implemented for 2WD and 4WD mobile robots. A popular geometry configuration used in vehicles is the Ackermann's geometry [2], [3]. In this configuration, steering wheels are geometrically configured to avoid sideways slips when performing a curvature motion. This is done by having the pivot of each steering wheel turn such that the axle points of all wheels meet at a common point as illustrated in Figure 2.1. Bell-crank, Rack-and-Pinion and Davis Steering are other steering wheels configurations that are also used for 2WD and 4WD mobile robots. Each configuration provides different level of complexity and steering controllability [4]–[6]. The Ackermann's configuration provides enough controllability for drifting task and its complexity can be further simplified as further discussed in Dynamic Modelling.

The geometrical motion of the robot is ultimately defined by the choices of body and steering configuration. Kinematic models of mobile robots are usually constricted to the lateral (planar) motions of their environment where the longitudinal motions are ignored for simplicity [7]. Further simplification can be made by ignoring motions along the vehicle's lateral axis (i.e. vehicle slipping are ignored). However, motion along the vehicle's lateral axis is required for drifting.

Rajamani in [7], provides an in-depth analysis of kinematic modelling, focusing mainly on the effect of wheels and steering configuration on the vehicle's motion. The author provides

a kinematic model of a 2WD vehicle that takes slipping into account. Seigwart and Nourbakhsh in [2], provide a simplified 2WD model by assuming no slipping and skidding. However, Rajamani argues that such assumptions do not hold at high velocity. An overview of wheeled mobile robots structures is provided in [8], where the authors discuss different wheels and steering configurations.

While the kinematic model has provided some basic overview of the robot's structure and geometry, the dynamics of the robot's physical interaction with its environment must also be defined.

2.1.2 Dynamic Modelling

Dynamic modelling provides information about the forces that causes motions. It also entails information about how the robot interacts with its physical environment.

The most direct contacts of a wheeled robot with its environment occur at its wheels. There are frictional forces interactions between the wheels and the floor that the robot travels on. 2WD and 4WD robots have frictional forces acting on the lateral and longitudinal axis of each wheel. These forces define the skidding and slipping effect on the vehicle during motion. To effectively represent the effect of the forces on the vehicle's motion, a good dynamic description of the wheels forces is required. Pacejka's Magic Formula (PMF) in [9], is a widely used formula for calculating the frictional forces of wheels. The formula uses the vehicle's slip angle and three empirical curve constants for the forces' computation.

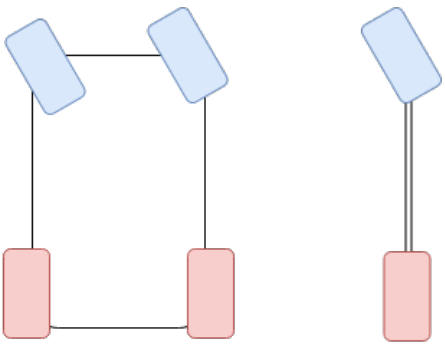

Figure 2.2: A four-wheeled 2WD vehicle on the left and a Bicycle model on the right.

While PMF can calculate the forces for all four wheels in a 2WD and 4WD mobile robots, the complexity of calculating eight interdependent forces (4 lateral and 4 longitudinal forces) exists. The complexity can be reduced by simplifying the structural representation of the model to a Bicycle Model [10]. The Bicycle Model, as illustrated in Figure 2.2, does

the simplification by combining the two fore and rear wheels of a four-wheeled mobile robot to represent the robot as a single rigid body with two wheels. The simplification reduces the number of wheels' frictional forces from eight to four.

There are other forces that are in relation with the vehicle's wheels' motion. The load and gravitational forces on each wheel define the behaviour of the vehicle's motion with respect to load change and distribution. The two forces can also be ignored in a Bicycle Model as performed by the authors in [11] and [12], thereby, further decreasing the complexity of the dynamic model.

Liniger et al. in [11], provide a simple dynamic description of a vehicle using a Bicycle Model. The load and gravitational forces of the two wheels are ignored and the longitudinal frictional forces are modelled using an electrical motor model. Velenos et al. in [13], provide a more complex 2WD model, where the frictional forces for all four wheels are described using PMF. Likewise, the load and gravitational forces on the wheels are also neglected. An in-depth analysis on PMF and its variations is performed by the authors in [14].

2.2 Control Algorithms

The basic concepts of reinforcement learning (RL) are introduced in this section. Some implementations of RL algorithms are explored through the discussion of their advantages and weaknesses. Discussions are also made on how deep learning (DL) can be used to enhance the performance of RL algorithms.

2.2.1 Introduction to Reinforcement Learning

Reinforcement Learning [15] is a subfield of machine learning used to train an agent/robot to interact with its environment to achieve maximum reward through the use of historical observational data of the robot's interactions. RL is commonly applied to robotics problem where the desired final/steady state of the robot is known but prior actions resulting to the state are not explicit. Autonomous path planning is a common example of such problems. The starting and final location (state) of the robots are usually known but the actions that will drive the robot to the final state are not explicitly known and can be stochastic. The state of the robot while completing the autonomy task depends on the environment of the robot. Attempt to map every possible states in the robot's environment is sometimes unfeasible. RL extracts/learns features from observational data and can apply learnt

features (experience) to perform actions when in an unfamiliar state. The features learnt by the RL algorithm is driven by a reward function that determines the value of an action taken in a given state. The terminology used for one of the learnt features is called policy. A policy provides a mapping of states to action (i.e. it is used to determine the action to take when in a state). A good policy is able to determine actions that receives maximum reward from the environment. While a reward function is used to determine the value of taking an action, a function is also needed to measure the value of being in a state. The Value function provides the total future reward that will be acquired from a given state up to the terminal state while following a policy [16]. The value of a state provided by a Value function is usually wrong at initial state and the Value function is a feature that is also learnt by the RL algorithm. An optimal policy is ultimately extracted from a learnt optimal Value function [16]. Value iteration is an iterative process of computing the optimal Value function by estimating the value of states.

Value iteration is used for deterministic environments because the agent knows the full state-state transition of the environment. A deterministic environment will always transition an agent from a given state to the same new state when the same action is applied. This is not always the case for complex robotics problems within complex environments. There usually exists some randomness properties in states transitions of the robotic environments. An environment that exhibit this property is called a stochastic environment. A function that can be used measure a robot's performance in a stochastic environment is the Q-function. The Q-function is used to calculate a Q-value of an agent's action in a given state (i.e. Q-function map state-action pairs to Q-values) when interacting with a stochastic environment. In principle, Q-function utilizes both the state and action of an agent while the Value Function uses only the state of the agent. The concept of using Q-function for learning is called Q-Learning.

Value Function and Q-function are mathematically represented using the Bellman's equation [16].

2.2.2 Model-based and Model-Free Learning

In a stochastic environment where the states transition are not fully predictable, value iteration method can still be applied [15]. This is done by learning a model of the environment based on the observations of the agent's interactions with the environment. Value iteration is subsequently applied on the modelled environment for learning. The

concept of modelling an environment is called model-based learning. A learning procedure where the optimal-policy is learnt directly from observational data without modelling the environment is called model-free learning. Q-learning is a type of model-free learning.

2.2.3 Continuous Space Learning

In most cases the Value function and Q-function can be represented as a lookup table that maps an index to a reward value. The index is a state and state-action pairs for Value function and Q-function respectively. The size of the lookup table grows as the action and state space grows. This growth imposes an obvious problem in a continuous-space task where it might be impossible to map all possibilities on the table. Attempt to discretize the continuous space will lead to loss of information that might be required for features extraction.

A modern approach to solving the size issue of a lookup table is through the use of Neural Network [17]. The Q-function or Value function can be replaced with a Deep Neural Network (DNN) that predicts the action based on a given input. These type of development allows RL algorithm to harness the power of DNN. Complex representation of a robot states can be provided to the DNN model to predict an action. An example of this approach is using images from a robot's camera sensor as states to predict steering direction (action) during autonomous navigation. Mnith et al. in [17] developed a reinforcement learning algorithm called Deep Q-Learning (DQN) that replaces the Q-function with DNN layers.

2.2.4 Policy-based Learning

Prior discussion of Q-learning and Value iteration were necessary in understanding the concepts of RL. Both learning methods are classified as value-based learning methods since the optimal learnt policy is based on the estimated values of their respective functions. Policy-based learning, a new type of learning method, will be now be discussed.

During the process of learning with a value-based method, the optimal policy is extracted from the learnt value function. A problem occurs when the learnt optimal value function doesn't correlate with the optimal policy. A better approach is to directly learn the optimal policy by searching a policy space. This approach is classified as a Policy-based learning method [15]. Sutton and Barton in [15] discussed the advantages of using policy-based methods over value-based methods through policy parameterization. One of those advantages includes assured convergence. Sutton and Barton showed that policy-based

method is able to converge provided there exists a solution in the parameter space. A local maximum solution can be found in the worst case. In contrast, value-based methods do not share this convergence properties. Other advantages of policy-based method include the continuous space support and elimination of perceptual aliasing. A policy-based method can be applied to continuous space problems depending on the parameterization of the policy. Perception aliasing occurs when two states appear to be the same or are the same but require different actions. Inevitably a value-based method will always predict the same action when faced with perception aliasing because of its table mapping property.

2.2.5 Deep Deterministic Policy Gradient

A policy-based learning method can be either deterministic or stochastic. A deterministic policy predicts an action given a state while a stochastic policy provides a probability over all possible actions. While a stochastic policy is suitable for a stochastic environment, the continuous space problem still prevails. It is unfeasible to provide a probability for all actions in a large continuous space without rigorous discretization. A determinist policy-based method is more suitable for continuous space problems.

The disadvantages of value-based over policy-based method have been discussed, however, value-based method does have some advantages over policy-based method. A score function is required to measure the performance of policies during training. The total reward of an episode is generally used to score a policy. An episode is defined as the sequential step transition of an agent from an initial state to a terminal state. Some actions within an episode can yield low rewards while the total reward averages out to be high. Such occurrence have negative impact on the learning rate of a policy-based method. Consequently, more data are required by the policy-based model to find an optimal policy (a policy where all rewards are high). The issue can also introduce another problem where the policy-based model gets stuck in a local maximum.

A good approach in resolving the issues imposed by policy and value based methods is by combining both. An actor-critic approach provides a framework for combining both policy and value based methods. An actor predicts the actions using a policy-based methods while the critic scores the performance of the actor using a value-based method. The policy parameters of the actor are updated based on the gradients of the critic's score values.

As discussed in Continuous Space Learning, DL can be applied to replace a value function in value-based methods. Similar approach can be applied in policy-based methods. The policy can be parameterized by defining the policy as DNNs.

Deep Deterministic Policy Gradient (DDPG) [18] is an algorithm that combines the actor-critic framework with Deep Learning. The actor and critic are defined as separate but similar DNNs. The authors of the DDPG algorithm showcased implementations of the DDPG algorithm on various robotics problems in a simulation environment. Some of these problems included high dimensional observation space environment. They were able to show the robustness and performance of the implementation by benchmarking against the original Deterministic Policy Gradient (DPG) [19]. The original DPG is a policy-based method that utilizes the actor-critic framework without defining the actor and critic as DNNs. The authors of the DDPG algorithm were inspired by the successful implementation of Deep Learning on Q-learning (DQN). A major positive observed in the DDPG implementation is the robustness of the hyper-parameter selection. The authors defined a standard hyper-parameter choice that were applied to almost all problems without further tuning.

The DDPG algorithm is used in this project to develop a control model. Discussion about the implementation can be found in DDPG Implementation.

2.2.6 Related Algorithm

The PILCO algorithm from [20] is a model and policy based algorithm. It is however not a RL algorithm but rather a policy search algorithm designed from scratch by the authors. It utilizes non-parametric probabilistic Gaussian process for modelling the dynamics of the agent. The performance of the modelled dynamics is then taken into account during policy search. PILCO 's authors claim that the algorithm is more data-efficient compared to RL algorithms.

The PILCO algorithm is notably mentioned because it was manually surveyed before the implementation of DDPG for this project. An implementation of PILCO by Rontsis in [21] was tested on similar robotic tasks benchmarked by the DDPG authors. The tests were performed on the OpenAI Gym environments [22] (a standard environment for benchmarking reinforcement learning algorithms). While the tests were not in-depth, the PILCO algorithm performed worse compared to the DDPG algorithm on the tests. It was able to solve some robotics tasks like the "*Inverted Pendulum*" more data efficiently

compared to the DDPG algorithm but for more complex continuous space problem like the "*Mountain Car*", the PILCO algorithm couldn't find an optimal policy within a "reasonable" time. The main reason for its poor performance is because of the exponential increase in policy search time after every iteration. The excessive time consumption rendered it unsuitable for the project given the constrained period for the project completion.

2.3 Related Work

Various works have been performed on developing controllers that achieve optimal drifting control of a vehicle. Bhattacharjee et al. in [23] provide the most related work to this project. They explored the use of RL algorithms such as DQN and PILCO for optimal drifting control of a 3D dynamic model. The authors successfully applied PILCO to achieve optimal drift but were unsuccessful with DQN. The authors also attempted to apply the controller to a real-world radio-controlled vehicle (RCV) but could not achieve optimal drifting. On the other hand, Velenis et al. in [13] utilize manually coordinated steering and drive torque control to achieve steady-state drifting on a modelled 4WD robot. The coordinated values were obtained from observational data collected during drifting manoeuvrers performed by an experienced driver on a real vehicle. While the method express that there exists a relationship between the real-world vehicle and the 4WD robot, it is important to emphasis that experienced controllers (humans) are not always available for creating real-world observable data for most robotic problems. The method inevitably required a complex dynamic model to ensure coordinated value correlation between the real-world and modelled vehicle. The authors of the PILCO algorithm in [24] were able to achieve optimal drift control on a 2D and 3D model and a real-world RCV. These results were achieved through the use policy transfer from each level of environment complexity. No related work was found to have used DDPG for optimal drifting control.

3 Methodology

This section covers the general methods utilized in the project. First, the chosen equations of motion of the 2D robot are discussed. The 3D robot is also briefly discussed before diving into the implementation of the DDPG algorithm.

3.1 2D Dynamic Model

The criteria for choosing a dynamic model was discussed in the Literature Review where the importance of having a simple but information-rich model was emphasized. Various implementations of dynamic modelling of a vehicle were reviewed from different literatures. The Bicycle model described in [11] offers a simple dynamic model that is suitable for the purpose of this project. The Bicycle model is used to describe the 2D mobile robots used for 2D simulations in this project. Let the 2D mobile robots be notated as Λ_2.

The equations of motion of Λ_2 are described using five state-space variables notated as $\dot{X}$, $\dot{Y}$, $\dot{v}_x$, $\dot{v}_y$ and $\ddot{\varphi}$. Aside, there is a major difference between the case notation used to describe coordinate axes. The world's fixed-frame axes are notated as X and Y while the moving frame of Λ_2 are notated as x and y. Let $\dot{X}$ and $\dot{Y}$ be the velocity of Λ_2 in the world's frame x-axis (longitudinal) and y-axis (lateral) respectively. While $\dot{v}_x$ and $\dot{v}_y$ are the velocity of Λ_2 in the x-axis and y-axis at its centre of gravity (COG). The model is controlled with a steering angle δ and duty cycle d input.

The structural properties of Λ_2 are described with the mass m, distance from the fore-wheel to the COG l_f, distance from the rear-wheel to the COG l_r and the inertia at the COG I_z.

It was stated in Dynamic Modelling that the complexity of computing eight frictional forces is reduced by using a Bicycle model which requires four forces. (i.e. two forces per wheel). To further simplify the model, the longitudinal frictional force of the fore-wheel can be neglected to assume that the fore-wheel does not skid in the x-axis direction of Λ_2. This assumption further reduces the number of frictional forces to three. Let $f_{R,x}$ and $f_{R,y}$ be the frictional forces of the rear-wheel in the x-axis and y-axis respectively. While the frictional force of the fore-wheel in the y-axis is notated as $f_{F,y}$. Where R and F in the three frictional forces notations, represent the rear-wheel and fore-wheel of Λ_2 respectively.

The slipping frictional forces, $f_{R,y}$ and $f_{F,y}$, are calculated using the Pacejka's Magic Formula given as:

$$f_{(i,y)} = D\sin\left(C\arctan\left(B\alpha_i\right)\right); i \in \{F, R\}, \tag{1}$$

where D, C and B are fitting constants that define the shape of the semi-empirical curve while α_i is the slipping angle of the wheels during motion.

A dynamic model of a motor which is dynamically controlled with a duty cycle d is used to drive the vehicle through $f_{R,x}$. The behaviour of the modelled motor is described with four constants: C_{m1}, C_{m2}, C_r and C_d. The modelled motor expression for finding $f_{R,x}$ is given as:

$$f_{R,x} = \left(C_{m1} - C_{m2} v_x\right) d - C_r - C_d v_x^2. \tag{2}$$

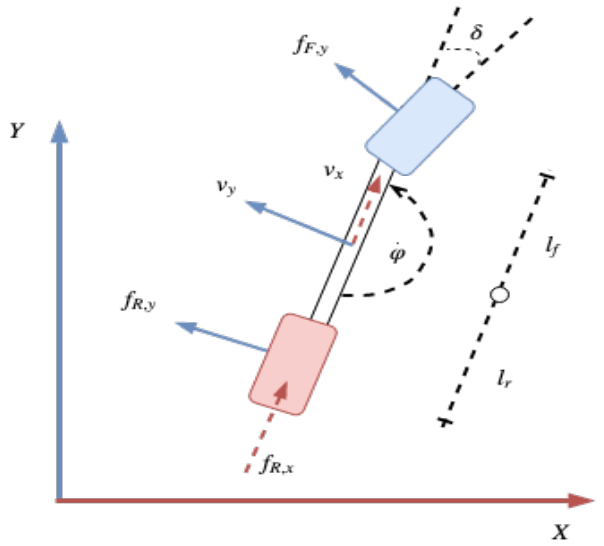

Figure 3.1: Bicycle model used to describe 2D dynamic models.

Finally, the equations of motion used to describe Λ_2 as illustrated in Figure 3.1 is given as:

$$\dot{X} = v_x \cos(\varphi) - v_y \sin(\varphi), \tag{3}$$

$$\dot{Y} = v_x \sin(\varphi) - v_y \cos(\varphi), \tag{4}$$

$$\dot{v}_x = \frac{1}{m}\left(f_{R,x} - f_{F,y}\sin(\delta) + m v_y \dot{\varphi}\right), \tag{5}$$

$$\dot{v}_y = \frac{1}{m}\left(f_{R,y} + f_{F,y}\cos(\delta) + m v_x \dot{\varphi}\right), \tag{6}$$

$$\ddot{\varphi} = \frac{1}{I_z}\left(f_{F,y} l_f \cos(\delta) - f_{R,y} l_r\right). \tag{7}$$

3.2 3D Dynamic Model

To introduce more complexity, a 3D model of a vehicle was also implemented in a 3D environment. The introduced complexity provides more information for analysing the performance and limitations of the implemented control model.

A pre-existing 3D dynamic model of a vehicle with Ackermann's steering configuration in [25] was used for the implementation. The dynamics of the robot are described by the authors in a Unified Robot Description Format (URDF) as provided in 3D Model URDF. URDF is a standard format used for describing robotic models with a markup language. The format allows portable and shareable description of models that can be rendered in multiple environments. The 3D model was implemented in the Gazebo (Gazebo, Ver. 10) 3D simulation environment [26].

Let the 3D models be notated as Λ_3. The model is described as a 2WD vehicle with an Ackermann's steering configuration at the fore-wheels. It also features shock absorbers on all a four wheels which introduces some complexity for drifting control. The description of Λ_3 contains a lot of dynamic and structural parameters which are provided in 3D Model Parameters under the Appendices section. However, only two parameters are relevant to this project: the two friction coefficients of each wheel. Let μ_1 and μ_2 be the friction coefficient of each wheel in the x-axis and y-axis of individual wheel respectively. These coefficients control the interaction of the wheels with the base floor. It is necessary to tune the variables to ensure optimal drifting control can be achieved by the control model.

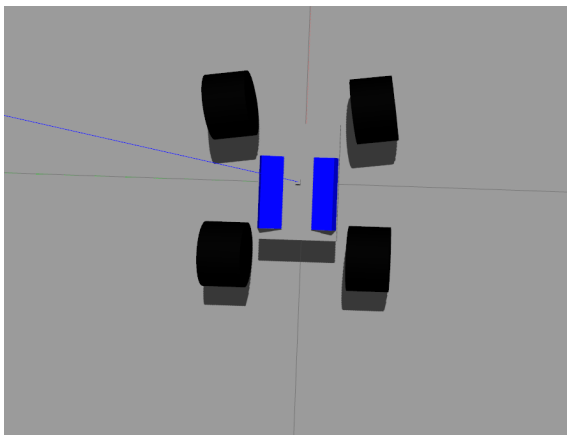

Figure 3.2: 3D model in the Gazebo environment.

The image in Figure 3.2 shows Λ_3 rendered in the Gazebo 3D environment. The 3D model is controlled with a steering and acceleration value.

3.3 Optimal Drifting

Before diving into the discussion of how the control model was implemented, it is important to define what optimal drifting or generally what a drift is and how they can be achieved. Naturally, when driving a vehicle and turning around a corner the vehicle faces the direction of motion. This behaviour is due to the interaction between the tyres and the road, and in some advanced cases, the control module of the vehicle. This behaviour holds under the condition that there is enough friction between the tyre and road to avoid the vehicle from moving sideways (drifting).

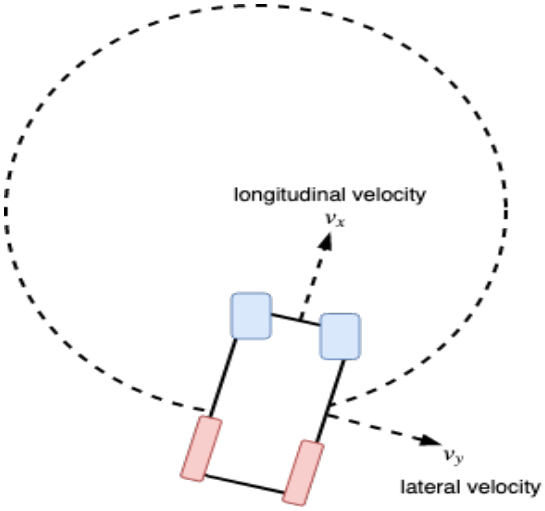

Figure 3.3: 2D representation of a vehicle drifting in a circular path.

As illustrated in Figure 3.3, a vehicle drifts when it doesn't face the path of its motion. Drifting occurs because there in enough velocity in the vehicle's lateral direction to cause sliding. The drifting radius which defines the drifting curve or path is determined by the linear and angular velocity of the vehicle. The aim of this project is to control the vehicles to achieve drifting with a certain radius and maintain it (optimal drifting).

There are various methods that can be used to attain drifting during motion. A common method is to make a sharp turn while travelling at a high velocity. The frictional forces of the wheels are affected by the velocity of the wheels and tend to reduce at high velocity. Another method is to make the road or wheels slippery. The slippery interaction between the wheels and the road allows the vehicle to move more freely in lateral and longitudinal direction. In this project, the dynamics of Λ_2 and Λ_3 are manually tuned using heuristic methods to achieve a slippery interaction between their wheels and the "road" in respective environments. The frictional forces of Λ_2 define this slippery interaction which is described by the PMF fitting and modelled motor constants. In Λ_3, the interaction is

defined by the frictional coefficients μ_1 and μ_2. However, using heuristic method won't achieve the optimal drifting required and that is where the control algorithm comes in. In retrospect of tuning the dynamics of the vehicle to achieve slippery motion, the control model is then supposed to "learn" the optimal control of steering the vehicles to attain optimal drifting. In conclusion, this project has utilized both method of contact interaction (slipperiness) and steering control to achieve optimal drifting.

3.4 Desired State Definition

The dynamic variables that define when a vehicle is drifting have been briefly discussed in Optimal Drifting. They are the lateral, longitudinal and angular velocity of the vehicle in reference to the vehicle's COG.

For the control model to achieve drifting, a desired state must be defined for the control model. A desired drifting state should describe the direction, speed and radius of drifting. The mathematical relationship between the variables v_y, v_x and $\dot{\varphi}$ provides the information for defining a desired drift state.

The mathematical expression is given as:

$$|v| = \sqrt{v_y^2 + v_x^2} \;\; ; \;\; r_d = \frac{|v|}{\dot{\varphi}}, \tag{8}$$

where r_d is the drift radius and $|v|$ is the absolute linear velocity of the model. By setting the values of v_y, v_x and $\dot{\varphi}$, the desired direction, speed and radius of drifting can be defined. When enough velocity is applied in the lateral direction of the vehicle (v_y) and there is not enough longitudinal velocity v_x to counteract the force imposed by the former velocity, the vehicle will start to move sideways. The values of v_x and v_y also determine the tilt angle θ_t of the vehicle during drifting as expressed by:

$$\theta_t = \arctan\left(\frac{v_y}{v_x}\right), \tag{9}$$

while $\text{sign}(v_y)$ determines the direction of drifting. The vehicle drifts in an anti-clockwise direction when $\text{sign}(v_y)$ is negative and in a clockwise direction when $\text{sign}(v_y)$ is positive.

In conclusion, let a vehicle's drift state at a time step be s_t and the desired drift state be s_d. The expression for representing both states is given by:

$$s_i = [v_x, v_y, \dot{\varphi}] \;;\; i \in \{t, d\}. \tag{10}$$

The aim of the control model is to control the vehicle to achieve and maintain the values specified in s_d (i.e. $s_t = s_d$).

3.5 Clarifying Notations

For onward reference to Λ_2 and Λ_3 models and their parameters, new notations have to be introduced for clarification. Both models share similar input control variables and the same dynamic states. A new variable e is introduced to represent the environment of a model with $e \in \{2,3\}$. Where the values 2 and 3 of e represent the 2D and 3D environment respectively. The steering control is now defined as δ_e, where δ_2 and δ_3 are the steering control variables of 2D and 3D models respectively. The 2D and 3D models are driven by a duty cycle d_2 and acceleration value d_3 respectively.

A new notation of dynamic models is also introduced to accommodate implementations of different models with different dynamic parameters. Let the dynamic model of a vehicle be $\Lambda_{2,k}$ and $\Lambda_{3,k}$ for a 2D and 3D model respectively, where the notation k in $\Lambda_{2,k}$ and $\Lambda_{3,k}$ represents a unique configuration of dynamic parameters for respective models.

The new notations are introduced mainly to distinguish between variables referring a 2D or 3D model. However, for general reference to both models, the variables will sometimes be referred to without e and k variables. This exception will be obvious in context.

3.6 Dynamic Models Analysis

Some analysis were performed to ensure that optimal drifting can be achieved by the dynamic models. The process of the analysis involved manual observation of the dynamic model's behaviours to parameter changes. By observing the states of the dynamic models during the analysis, the right configuration of dynamic parameters that achieves drifting can be used to describe the models. The analysis echoes the discussion in Optimal Drifting about manually tuning a vehicle's interaction with the road to enable slippery behaviour.

The main objective of the analysis is to find the combination of dynamic parameters that emulates a vehicle on a slippery surface (i.e. the vehicle can drift easily). This ensures that a solution exists for the control model to find. The configuration of dynamic parameters that showed potentials of achieving optimal drift during the analysis were recorded.

The analysis were performed by running multiple simulations of Λ_2 and Λ_3 models in their respective environments with different dynamic parameters. During simulations, the steering controls of the models were systematically changed within intervals while the velocity controls were kept constant. Varying the steering at short intervals simulates aggressive steering that normally leads to drifting of a vehicle. The applied steering values varied between extreme left steering angle and no steering for the respective model. The simulations were performed for 20s in both the 2D and 3D environment with a simulation time step of 0.01s. Input variables, steering angle and velocity control, were applied at each time step and the resulting state variables s_t were recorded for drifting behaviour analysis. For each state recorded at each time step, the tilt angle θ_t is calculated and observed. The vehicle is considered to be drifting if $|\theta_t| \geq 0.26$ rad(15^0). Through the observation of θ_t and s_t the dynamic parameters of the models are tuned to achieve the drifting criteria at some time steps. As previously stated, this observation is necessary to ensure a solution exists for the control model to find.

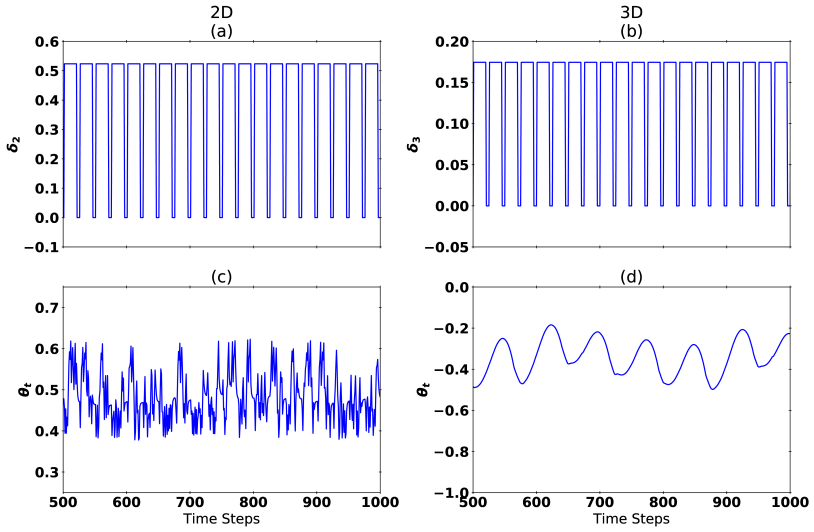

Figure 3.4: Illustrations of applied steering angles and the tilting behaviour of the vehicles.

For the 2D simulations, the steering angle δ_2 was alternated between a value of 0.52 rad (30°) for 20 time steps and a 0 rad steering value for 5 time steps while the drive value d_2 was kept constant at value of 1. The values of δ_2 during the simulation period are

illustrated in Figure 3.4 (a). During each simulation, the dynamic parameters manually tuned are the PMF's fitting constants (D , C and B) and the modelled motor constants (C_{m1} , C_{m2} , C_r and C_d).

2D Dynamic Models	D	C	B	C_{m1}	C_{m2}	C_r	C_d
$\Lambda_{2,1}$	8	9.5	2500	1250	3	100	45
$\Lambda_{2,2}$	6	10.49	1942	1101	15	132	38
$\Lambda_{2,3}$	8	9.5	2500	1000	5	80	40
$\Lambda_{2,4}$	6	10.49	1942	1101	3	100	45

Table 3.1: Different dynamic configuration of the 2D dynamic model.

The configurations of dynamic parameters that achieved the drifting criteria in the 2D environment were recorded as shown in Table 3.1. Figure 3.4 (c) shows the tilt angle of $\Lambda_{2,1}$ in response to alternating steering angles. It can be observed that the tilt angle of $\Lambda_{2,1}$ is above the 0.26 rad threshold at majority of the simulation period. The effects of tuning each parameter were also observed. While the effects of each parameter are not independent, it was observed that C_{m1} and C_{m2} affect the velocity of the models while C affects the drifting behaviour. The parameter B controls the angular velocity (how quickly the vehicle can turn) while D controls the magnitude of lateral and angular velocity. The recorded 2D models in Table 3.1 will be used for further control model training and analysis.

Similarly, the 3D model analysis were performed in the same format. However, different type of dynamic parameters were manually tuned for the 3D model since they are described differently. The dynamic parameters used for 3D models' analysis are the frictional coefficients μ_1 and μ_2 . The steering value δ_3 was alternated between a value of 0.17 rad (10°) for 20 time steps and a 0 rad steering value for 5 time steps while the drive value d_3 was kept constant at a value of 3.5 ms^{-2} . Figure 3.4 (c) illustrates the steering control. No further observations were required for observing the effect of the changes to the parameters μ_1 and μ_2 as they have predefined effects. The parameters μ_1 and μ_2 define the frictional interaction of each wheel in the x-axis and y-axis respectively. The same values of μ_1 and μ_2 are applied to all four wheels of the vehicle. Like in the 2D

simulations, the states s_t at each time step were recorded and analysed for drifting behaviour for different parameter configurations.

3D Dynamic Models	μ_1	μ_2
$\Lambda_{3,1}$	1	0.5
$\Lambda_{3,2}$	0.4	0.4

Table 3.2: Different dynamic configurations of the 3D dynamic model.

Recorded values of the different configurations are provided in Table 3.2. The tilt angles of $\Lambda_{3,2}$ recorded during the analysis simulation are shown in Figure 3.4 (d). It can be observed that the model achieves the tilt angle criteria. Similarly, the model $\Lambda_{3,1}$ listed in Table 3.2 achieved these criteria.

The analysis performed on the 2D and 3D models helped define and select the dynamic models that will be used for control model training and analysis. Further reference of dynamic models with notation $\Lambda_{e,k}$ will refer to models described in Table 3.1 and Table 3.2.

3.7 Reward Function

The concept of a reward function and value was discussed in Control Algorithms when introducing Reinforcement Learning algorithms. A reward function is used to direct the control model towards finding the optimal solution. This is done by rewarding the policy-maker (control model) every time it performs an action. In other words, the reward function defines the value of taking an action in a given state.

The states of the dynamic models required for drifting purpose have been defined in Desired State Definition. A drift reward function will provide a reward value by comparing the current vehicle's state s_t against a desired drift state s_d. Let $\Re$ be the reward function that computes the drift reward value r_t at a time step when provided with the vehicle state and desired state. The mathematical expression for reward value relationship is given by:

$$r_t = \Re(s_t, s_d). \tag{11}$$

The drift reward function $\Re$ needs to compute a reward value that clearly distinguishes between a drifting state and non-drifting state. An obvious method for defining the reward

function is by directly using the negative absolute difference between the current state and desired state. This definition will provide a reward value of zero when $s_t = s_d$ and a negative value when otherwise. However, the velocities in the drift state definition are not of the same magnitude. This might introduce an imbalance in computed reward values where some drift state variables hold more weight over others. An alternative approach is to use a negated mean squared error (MSE) function. However, the MSE function still has the problem of computing a biased reward for at each time step. These problems hold as long as the reward value is not globally unified across all time steps.

Cutler and How in [24] defined a drift cost function used in their PILCO algorithm. The cost value computed by the cost function is bounded between a value of 0 and 1. The function features a squared error that is normalized by a sensitivity control variable. However, RL algorithms optimize to maximize a value in contrast to minimizing a cost value like the PILCO algorithm. Therefore, the cost function is negated and used to define the drift reward function. Finally, the reward function expression is given by:

$$r_t = \Re(s_t, s_d) = -\left(1 - \exp\left(\frac{-\sum (s_t - s_d)^2}{2\sigma^2}\right)\right) \in [-1, 0], \tag{12}$$

where $\sigma \in [0,1]$ is a sensitivity factor for adjusting the tolerance of error between the current state and desired state values. The higher the value of σ, the higher the tolerated error value.

With the reward function definition in equation (12), the reward at each time step is bounded between -1 and 0. However, the performance of the control model will be measured using the total reward accumulated during an episode of simulation. Let r_T be the total sum of rewards accumulated during an episode. The expression for finding r_T is given by:

$$r_T = \sum_{t=0}^{T} r_t = \sum_{t=0}^{T} \Re(s_t, s_d), \tag{13}$$

where T is the total length of an episode simulation and t is a time step. An optimal drift controller will achieve a r_T value of 0 (i.e. the controller achieves a perfect optimal drift).

3.8 DDPG Implementation

The concepts about the DDPG algorithm was discussed in Deep Deterministic Policy Gradient. It was mentioned that the DDPG algorithm was chosen because it mitigates some problems introduced by other RL algorithms. It is suitable for high dimensional space model control and evidently suitable for this project's purposes.

DDPG is policy based control models that is based on the actor-critic framework. The actor is the policy-maker that predicts the action or so to say, controls the vehicle to achieve optimal behaviour. Observing the actions of the actor is the critic which measures the performance of the actor and provides feedback through sampled gradient. The actor and critic are implemented using a Neural Network architecture which provide parameterization for both networks. Once more, the actor is a policy-based method while the critic is a value-based method. The differences between both methods have been discussed in-depth in the Control Algorithms section.

In this section, the mathematical definitions of the DDPG algorithm are discussed using combinations of previously and newly defined notations.

Let the actor network be $\mathcal{A}(s_t|\theta^{\mathcal{A}})$ which maps a state s_t to an action a_t at a given time step. In the context of this project, the action a_t is the steering variables δ_2 and δ_3 for 2D and 3d models respectively. The variable $\theta^{\mathcal{A}}$ is the parameter that must be learnt for optimal mapping of states to action. The learning of $\theta^{\mathcal{A}}$ is performed through gradient updates sampled from the critic network $Q(s_t, a_t|\theta^{Q})$. Note that the critic network is also parameterized with a parameter θ^{Q} which must also be learnt. The critic network maps a state-action pair to a Q-value q_t.

3.8.1 Actor and Critic Network Optimization

As previously stated, the Q-value quantifies the value of performing an action in a given state. To be more specific, the value of taking an action in a given state is not only limited to the current state but consequently the future states. In other words, the Q-value is the expected long term or future reward of taking an action in given state following a given policy. The definition of the Q-function can be mathematically expressed using the Bellman's equation as given by:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[\Re(s_t) + \gamma\, \mathbb{E}[Q^{\pi}(s_{t+1}, a_{t+1})]], \tag{14}$$

where π is a given policy and $\gamma \in [0,1]$ is a discount factor. The discount factor is used to weight the effect future rewards. It is used as a traded off mechanism between acquiring earlier small rewards or future larger rewards. Note that the expression is a recursive function as it contains the expected computation of future rewards $\mathbb{E}[Q^{\pi}(s_{t+1}, a_{t+1})]$. The authors of the DDPG algorithm state that inner recursion can be avoided if the target policy is deterministic (i.e. the policy maps a state to an action value and not to the probability over all actions in the action space (stochastic policy)). Using the policy-maker $\mathcal{A}(s_t|\theta^{\mathcal{A}})$ and critic $Q(s_t, a_t|\theta^{Q})$, the Q-value q_t can finally be computed using the expression given by:

$$q_t = \Re(s_t) + \gamma\, Q(s_{t+1}, \mathcal{A}(s_{t+1})|\theta^{Q}). \tag{15}$$

A loss function $L(\theta^{Q})$ is defined for learning and optimizing the parameterized critic network. The function is defined using the squared error which measures the output of the critic network against a true value. The true value is the Q-value q_t. Note that the computation of q_t as expressed in equation (15) depends on a parameterized critic network. The authors of the DDPG algorithm state that the parameterization can be ignored. Finally, the loss function of the critic Network is given by:

$$L(\theta^{Q}) = (q_t - Q(s_t, a_t|\theta^{Q}))^2. \tag{16}$$

For learning and optimizing the parameter of the actor network, the parameter $\theta^{\mathcal{A}}$ is updated using the policy gradient. The computation of the policy gradient from [19] is given by the expression:

$$\nabla_{\theta^{\mathcal{A}}}\mathcal{A} = \nabla_a Q(s_t, \mathcal{A}(s_t)|\theta^{Q})\ \nabla_{\theta_{\mathcal{A}}}\mathcal{A}(s_t|\theta^{\mathcal{A}}). \tag{17}$$

3.8.2 Target Networks

As expressed in equation (15) and (16), the true Q-value q_t used to optimize the parameter q_t also depends on the critic network $Q(s_t, a_t|\theta^{Q})$. This could lead to a problem of divergence or oscillatory and unstable learning performance. The authors of the DDPG algorithm proposed a solution where they made a copy of each network and slowly

update their parameters using the original networks. Let $Q'(s_t, a_t|\theta^{Q'})$ and $\mathcal{A}'(s_t|\theta^{\mathcal{A}'})$ be the copied critic and actor's network respectively. The copied networks will be referred to as the target networks while the original networks $Q(s_t, a_t|\theta^{Q})$ and $\mathcal{A}(s_t|\theta^{\mathcal{A}})$ are referred to as the online networks. The online networks are used for learning and optimization of their respectively parameters while the target networks are slowly updated using a weight value τ. Subsequently, the target critic and actor networks are used to calculate the true Q-value q_t. In other words, the critic and actor networks in equation (15) are replaced with the target networks notations. The expression for updating the parameters of the target networks is given by:

$$\theta^{n'} = \tau\theta^{n} + (1-\tau)\theta^{n'}, \tag{18}$$

with $n \in \{\mathcal{A}, Q\}$ and $\tau \ll 1$. Although the learning of the optimal parameters may be slow, the authors of the DDPG algorithm states that the trade-off is out-weighted by the learning stability of the network.

3.8.3 DDPG Algorithm

When training the neural networks with relevant variables at each time step, the networks sees the variables as independent variables. However, this is not true as each variable (state, action and reward) at each time step is co-dependent with past and future variables. Instead of training the networks with a single sample of the variables at each time step, the authors of DDPG algorithm utilized mini-batch sampling of data from a replay buffer. A replay buffer $\mathfrak{B}$ is a fixed sized buffer used to store the history of transitions in sequential order. A transition is defined as a tuple (s_t, a_t, r_t, s_{t+1}), where s_{t+1} is the transitioned new state after taking an action a_t in state s_t. The samples are randomly sampled from the buffer at a fixed mini-batch size b_s during time step iterations. When the buffer is full, the oldest stored transition is removed and the new transition is stored.

One problem normally faced in reinforcement learning is the problem of policy exploration. During the learning phase, a good policy might be found but the policy might not necessarily be the optimal policy. If the algorithm does not explore other possible solutions by gathering more information, then it can get stuck in a local maximum (i.e. it assumes the "good" policy is the optimal policy). A good exploration implementation is usually

accompanied by exploitation trade-off in some RL algorithms. Exploitation is when the algorithm decides to find the best solution using the information at hand (i.e. tries to get the "best" policy from available information). In DDPG, the exploration can be handled independently of the exploitation. The exploitation is handled by the learning process of the neural networks (i.e. through hyper-parameters tuning). The authors of the DDPG algorithm provide a method of handling the exploration by adding noise to the actor's policy. Let $\mathbb{n}$ be the action noise and the expression for adding the noise be given by:

$$a_t = \mathcal{A}(s_t|\theta^{\mathcal{A}}) + \mathbb{n}. \tag{19}$$

The Ornstein-Uhlenbleck process [27] was used by the DDPG's authors to define the action noise. The process is a stochastic process that methodically returns to a defined centre point as time continues. A parameter $\sigma_{\mathbb{n}}$ is used to adjust the centre point of the noise. Adding noise allows the actor to explore more action possibilities during the training phase. The action noise is not added during inference.

Algorithm 1: DDPG Training Algorithm

Randomly initialize the parameters of the online networks $Q(s_t, a_t|\theta^Q)$ and $\mathcal{A}(s_t|\theta^{\mathcal{A}})$.
Initialize target networks $\mathcal{A}'$ and Q' with $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mathcal{A}'} \leftarrow \theta^{\mathcal{A}}$.
Initialize replay buffer $\mathfrak{B}$.
Initialize vehicle dynamic model $\Lambda_{e,k}$.
for episode **in** 1:J **do**:
  Initialize noise $\mathbb{n}$.
  Observe the initial state s_1 of the vehicle $\Lambda_{e,k}$.
  for t **in** 1:T **do**:
    Retrieve noised action a_t using the online actor: $a_t = \mathcal{A}(s_t|\theta^{\mathcal{A}}) + \mathbb{n}$.
    Apply action a_t and observe transitioned state s_{t+1} and reward r_t:

$$(s_t, a_t, r_t, s_{t+1}) \leftarrow \Lambda_{e,k}(a_t).$$

    Store the transition in the replay buffer: $\mathfrak{B}(s_t, a_t, r_t, s_{t+1})$.
    if sizeof($\mathfrak{B}$) $\geq b_s$:
      Randomly sample a mini-batch of b_s transitions from $\mathfrak{B}$.
      Compute the true Q-value q_t with the target networks:

$$q_t = r_t + \gamma\, Q'(s_{t+1}, \mathcal{A}'(s_{t+1})|\theta^{Q'}).$$

      Batch update the online critic by minimizing the batch loss:

$$L(\theta^Q) = \frac{1}{b_s}\sum_i (q_i - Q(s_i, a_i|\theta^Q))^2.$$

      Batch update the online actor's policy using sampled gradient:

$$\nabla_{\theta^{\mathcal{A}}}\mathcal{A} = \frac{1}{b_s}\sum_i \nabla_a Q(s_i, \mathcal{A}(s_i)|\theta^Q)\ \nabla_{\theta_{\mathcal{A}}}\mathcal{A}(s_i|\theta^{\mathcal{A}}).$$

      Update the target networks' parameters:

$$\theta^{Q'} = \tau\theta^Q + (1-\tau)\theta^{Q'},$$
$$\theta^{\mathcal{A}'} = \tau\theta^{\mathcal{A}} + (1-\tau)\theta^{\mathcal{A}'}.$$

  end for
  Reset the dynamics of $\Lambda_{e,k}$ to initial state.
end for

With majority of DDPG concepts and implementation discussed, the algorithm for training a dynamic model is provided in Algorithm 1. The provided algorithm is a rectified version of the original DDPG algorithm published by its authors. It has been rectified to include the vehicle dynamic model and further clarifications were made for easier apprehension.

3.9 Control Model Training and Inference

Given that the actor and critic networks utilize a neural network architecture, the network include hyper-parameters that can be tuned for optimization. These hyper-parameters

include parameters for describing the network architecture itself and also for adjusting the learning performance.

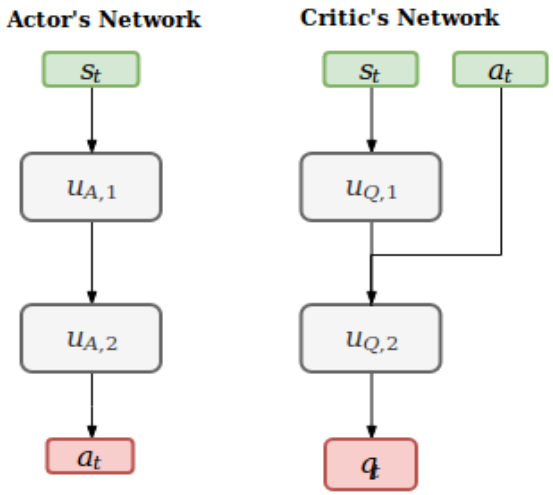

Figure 3.5: Architecture of the actor-critic network in the DDPG algorithm.

For both networks, the Adam optimizer [28] was used to optimize the loss function and sampled gradients of the critic and actor networks respectively. The Adam optimizer features a learning rate hyper-parameter for adjusting the learning behaviour of the optimizer. Let the learning rates (LR) be notated as α_A and α_Q for the actor and critic networks respectively. The architecture of the network is defined by the number of hidden layers and their respective units. The authors of the DDPG algorithm used 2 hidden layers for both networks in all their low-dimensional experiments. While the authors use a default unit size of 400 and 300 for both hidden layers, in this project the number of units are tuned while keeping to the default number of hidden layers (i.e. all trained DDPG models have two hidden layers). Let the units size of each layer be notated as $u_{n,1}$ and $u_{n,2}$ for the first and second hidden layer respectively with $n \in \{A, Q\}$. The output of the actor is bounded with a tanh function to an action bound value of a_b. By bounding the output, the action computed by an actor at a time step is represented with $a_t \in [-a_b, a_b]$. This is useful for left and right angle steering control. The architecture of the actor and critic networks is illustrated in Figure 3.5.

The 2D and 3D dynamic models defined in Table 3.1 and 3.2 were used to train DDPG models in various training experiments. The experiments were performed in respective model's environment for varying numbers of episodes. During the training experiments, the performance of the models were manually observed using the total reward per episode r_T

and the training process was manually intercepted when a decent performance is observed or the model is making not further progress. For simplicity purpose, a trained DDPG model will be defined as the target actor $\mathcal{A}'_{e,k}(s_t|\theta^{\mathcal{A}'})$ with learnt parameter $\theta^{\mathcal{A}'}$. The target actor's parameter $\theta^{\mathcal{A}'}$ were stored at every episode for later use in inference experiments.

While the training experiments of the 2D and 3D models share similar procedures, different set of hyper-parameters were tuned during respective experiments. However, the value of the target parameter update weight τ was kept constant at 0.001 for all experiments. Attempts to change τ to other values had negative effect on the learning performance of the DDPG models.

3.9.1 2D Model Training

The 2D training experiments was performed for 20s per episode with a time step of 0.01. This means that there are 2000 transitions in every episode and the minimum sum of an episode reward is -2000 with a maximum of 0 reward. In all 2D experiments, an action bound value of 0.2 rad was used while the action noise parameter $\sigma_{\mathbb{n}}$ was set to 0.005.

2D DDPG Models	$\Lambda_{2,k}$	γ	b_s	$\alpha_{\mathcal{A}}$	α_Q	$u_{\mathcal{A},1}$	$u_{\mathcal{A},2}$	$u_{Q,1}$	$u_{Q,2}$
$\mathcal{A}'_{2,1}$	$\Lambda_{2,1}$	0.99	20	$1e^{-4}$	$1e^{-3}$	400	300	400	300
$\mathcal{A}'_{2,2}$	$\Lambda_{2,2}$	0.99	20	$1e^{-4}$	$1e^{-3}$	400	300	400	300
$\mathcal{A}'_{2,3}$	$\Lambda_{2,1}$	0.99	30	$9e^{-5}$	$1e^{-3}$	400	300	400	300
$\mathcal{A}'_{2,4}$	$\Lambda_{2,2}$	0.89	30	$1e^{-4}$	$1e^{-3}$	400	300	450	300
$\mathcal{A}'_{2,5}$	$\Lambda_{2,3}$	0.99	40	$1e^{-4}$	$1e^{-3}$	405	300	405	300
$\mathcal{A}'_{2,6}$	$\Lambda_{2,2}$	0.99	30	$9e^{-5}$	$1e^{-3}$	400	300	400	300
$\mathcal{A}'_{2,7}$	$\Lambda_{2,4}$	0.99	35	$1e^{-4}$	$1e^{-3}$	400	300	400	300

Table 3.3: List of 2D DDPG models trained on different 2D dynamic models with different hyper-parameters configurations.

By using the 2D dynamic models previously defined in Table 3.1 and different configurations of hyper-parameters, 7 DDPG models were trained with each achieving varying performance. Each DDPG model is listed in Table 3.3 which shows the choice of hyper-parameters for each trained model. Note that the notation of the trained DDPG model follows the same notations used dynamic models $\Lambda_{e,k}$ (i.e. variable e is the

environment and k is a unique configuration identifier). All 2D models were trained to optimize on the same desire state with $s_d=[4.1,-2.1,2.3]$. The desired state describes a drift radius of 2m with the vehicle travelling in the clockwise direction at a velocity of 4.6 ms^{-1} while being tilted at an absolute angle of 0.47 rad. The drive variable d_2 was set to 1 for all 2D training experiments.

3.9.2 3D Model Training

Similar experiment procedures used in 2D training experiment were also used in 3D experiments. The experiments were run for a simulation time of 10s per episode with a time step of 0.01. This results in 1000 state transitions at each episode and a minimum total reward score of -1000. In contrast to the 2D experiments, each 3D DDPG models were trained to achieve different desired states.

3D DDPG Models	$\Lambda_{3,k}$	s_d
$\mathcal{A}'_{3,1}$	$\Lambda_{3,1}$	$[3.6,-2.2,2.1]$
$\mathcal{A}'_{3,2}$	$\Lambda_{3,2}$	$[2,-2,2]$
$\mathcal{A}'_{3,3}$	$\Lambda_{3,2}$	$[1,-1,2]$

Table 3.4: List of 3D DDPG models trained on different 3D dynamic models with different hyper-parameters configurations.

All other hyper-parameters were assigned for all 3D experiments as follows: $u_{n,1}=0.2$, $u_{n,2}=200$, $\gamma=0.99$, $\alpha_{\mathcal{A}}=1\mathrm{e}^{-4}$, $\alpha_Q=1\mathrm{e}^{-3}$, $d_3=4$, $b_s=20$, $\sigma_{\mathbb{n}}=1\mathrm{e}^{-3}$ and $a_b=0.2$. All trained 3D models are listed in Table 3.4 with the desired state the models were trained to achieve.

3.9.3 Model Inference

As stated earlier, the parameter $\theta^{\mathcal{A}'}$ of the target actors were saved at the end of every episode. Referring back to Algorithm 1, it should be noted that the learnt parameters changes at every iteration during an episode of training. This means that the total sum of an episode reward r_T acquired during training does not reflect the performance of the parameter saved at the end of an episode. This implies that the r_T of the saved parameter may be different. For every trained DDPG model, each episodic parameter is re-evaluated to calculate its actual r_T. This is performed by reconstructing the actor's Network with the same hyper-parameters used to train $\theta^{\mathcal{A}'}$ and initializing its parameter with the

corresponding episode parameter. Let a saved episode parameter be $\theta_{(j)}^{\mathcal{A}'}$ with j being the episode number.

Algorithm 2: DDPG Inference Algorithm

Initialize a DDPG model $\mathcal{A}'_{e,k}$ with corresponding hyper-parameters.
Initialize corresponding dynamic model $\Lambda_{e,k}$.
for episode j **in** 1:J **do**:
  Initialize the $\mathcal{A}'_{e,k}$ with saved episode parameter: $\theta^{\mathcal{A}'} \leftarrow \theta_{(j)}^{\mathcal{A}'}$.
  Observe the initial state s_1 of the vehicle $\Lambda_{e,k}$.
  for t **in** 1:T **do**:
    Retrieve action a_t: $a_t = \mathcal{A}'_{e,k}(s_t|\theta^{\mathcal{A}'})$.
    Apply action a_t and observe step reward r_t:
    $$r_t \leftarrow \Lambda_{e,k}(a_t).$$
  end for
  Calculate the episode total reward $r_{T(j)}$ from all observed step rewards.
  Reset the dynamics of $\Lambda_{e,k}$ to initial state.
end for

The procedure for re-evaluating an episode's learnt parameter is presented in Algorithm 2. The saved parameter with the best performance is the one with the maximum total sum of reward r_T (i.e. the best parameter is given as $\theta_{(j)}^{\mathcal{A}'} \leftarrow \text{argmax}_j(r_{T(j)})$).

4 Evaluation and Results

The results of all experiments performed are presented in this section. Optimal drifting was achieved on the 2D dynamic models but not on the 3D models. Plausible causes of not achieving optimal drifting on the 3D model are discussed. Some further evaluation analysis are also performed to test the performance and robustness of the trained control models.

4.1 2D and 3D Results

After training all DDPG models with corresponding dynamic models using Algorithm 1, all models saved during an episode were used in inference experiments to retrieve inference score $r_{T,inf}$. The inference scores are the scores used to evaluate the performance of the models.

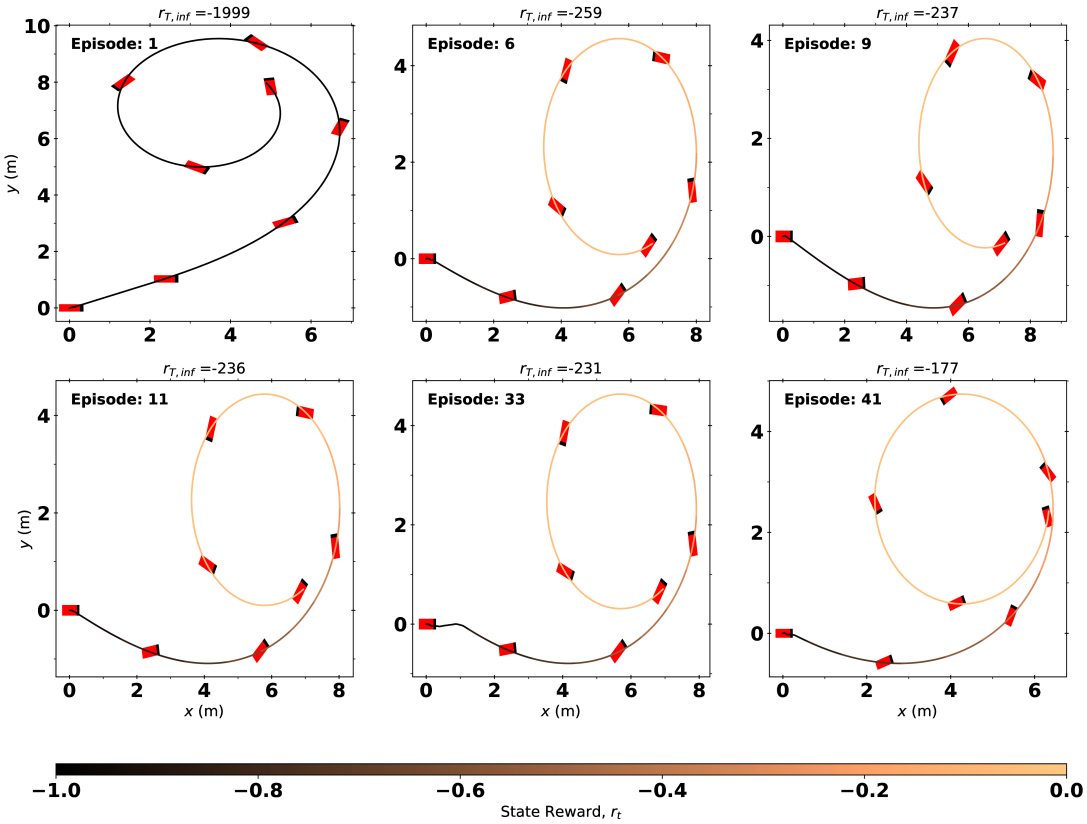

Figure 4.1: 2D representation of 2D dynamic model's ($\Lambda_{2,4}$) trajectory during episodic training of $\mathcal{A}_{2,7}^{'}$.

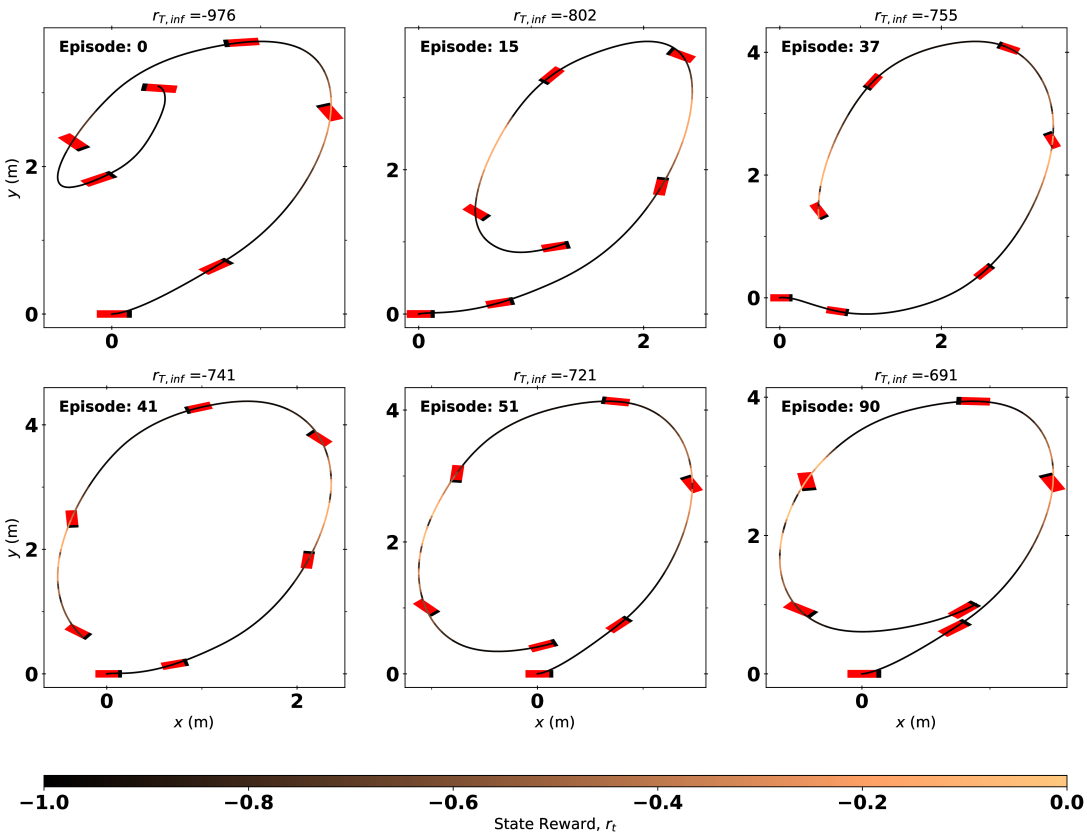

Figure 4.2: 2D representation of 3D dynamic model 's ($\Lambda_{3,1}$) trajectory during episodic training of $\mathcal{A}_{3,1}^{'}$.

Figure 4.1 and 4.2 shows the trajectory paths of dynamic models $\Lambda_{2,4}$ and $\Lambda_{3,1}$ respectively during episodic training of their respective models $\mathcal{A}_{2,7}^{'}$ and $\mathcal{A}_{3,1}^{'}$. The trajectory representations shows the reward achieved at every time steps by the use of colour gradients, where the colours black to yellow map low to high rewards respectively. The yellow region represents when the vehicle is drifting. It can be observed that the vehicle faces the centre of the drift circle during the drifting phase. The representations do not show the full length of simulation for clarity.

The control model $\mathcal{A}_{2,7}^{'}$ is able to optimally control dynamic model $\Lambda_{2,4}$ as shown Figure 4.1. Note that the model performs gravely at initial episode but increases performance quickly by episode 6. The best episode performance can be observed in episode 41 where the control model achieved a total reward score of -177. By definition, this score does not represent optimal drifting however, it can be observed that most of the low rewards in episode 41 were attained during the initial states before drifting. An explanation to this is that the dynamic model required some response time before reaching steady state of drifting. After this initial phase, the control model is able to continuously maintain the vehicle in a drifting state (optimal drifting). A model that was able to achieve a better performance compared to model $\mathcal{A}_{2,7}^{'}$ is model $\mathcal{A}_{2,4}^{'}$ which achieved an inference score of -165 on its corresponding dynamic model.

In contrast to the good control performance observed in 2D, none of the 3D DDPG model achieve desired drifting on the 3D dynamic models. The best performance observed was by $\mathcal{A}_{3,1}^{'}$ which achieved an inference score of -691 on $\Lambda_{3,1}$. It was observed that 3D vehicle was controlled to continuously accelerate in a forward direction and then drift for a short period. This behaviour can be carefully observed in some episodes in Figure 4.2. Notice that the vehicle travels in forward direction and then drifts for a short period in almost the same regions across episodes. The low performance of the DDPG model is credibly due to the complexity of the 3D model and its dynamic description. The dynamic parameters that greatly affects the performance are the frictional coefficients which requires manual tuning and observation to get right. Given that this project's major focus was on the 2D models and there was not enough time, further experiments couldn't be performed to find the right set of coefficient values to achieve optimal drifting. However, giving the learning progress observed in Figure 4.2, it is certain that the DDPG algorithm can achieve optimal drifting control on the 3D dynamic models.

Provided in Training Results are the best training and inference scores achieved by each DDPG model on their respective dynamic model. The table also include the episode at which the scores were achieved.

4.2 Reward Analysis

Some training and inference reward analysis were performed to further understand the learning process of the DDPG algorithm.

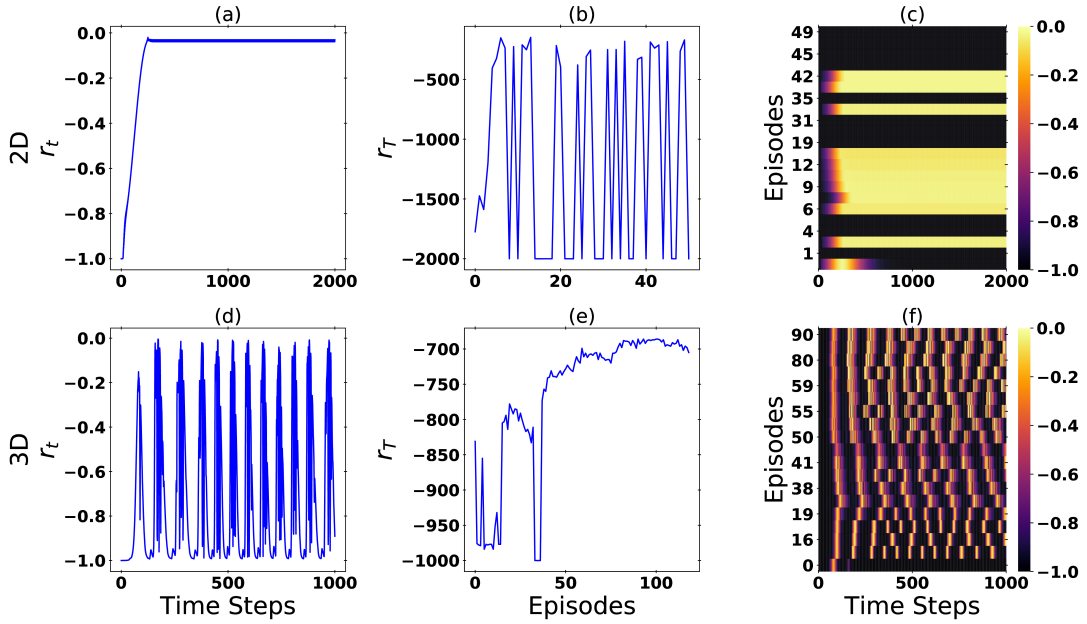

Figure 4.3: Acquired rewards during training and inference of DDPG models.

The analysis looks at three representations of rewards acquired during training and inference of the 2D and 3D DDPG models. Figure 4.3 (a) and (d) show the step reward r_t of control model $\mathcal{A}'_{2,7}$ and $\mathcal{A}'_{3,1}$ respectively during an episode inference while Figure 4.3 (b) and (e) show the total reward r_T acquired at every episode during training of the two models. It can be observed that the step reward is relatively steady for the 2D model while it is the otherwise for the 3D model. This observation reflects the discussion about the behaviour of the control model discussed in 2D and 3D Results. However, in the episodic training phase, the learning process is not stable for the 2D model but reasonably stable for the 3D model. An explanation for this is that the DDPG algorithm explores more possible solutions when making positive progress but tries to exploit a certain region when otherwise. This reflects in the learning behaviour of the 2D model where the DDPG algorithm continues exploring other solutions even when relatively close to the optimal

solution. This type of behaviour can have a positive and negative impact on the learning performance as it can divert the model from getting stuck in a local maximum but can also slow down learning process. The illustrations in Figure 4.3 (c) and (f) shows the reward per step per episode of the two models $\mathcal{A}_{2,7}^{'}$ and $\mathcal{A}_{3,1}^{'}$ respectively. Again, the illustrations shows the learning and control behaviour of the DDPG model. Figure 4.3 (c) shows that the 2D model is able to maintain the drifting state over the whole period in some episodes. Likewise, Figure 4.3 (f) shows the cyclic behaviour of the 3D model where it only achieves drifting at intervals. The colour gradients in Figure 4.3 (c) and (f) follow the same format as in Figure 4.1 and 4.2 where the black-yellow gradient maps to low-high state reward.

4.3 Further Analysis

Evaluating the performance of the DDPG models on just the inference reward is not enough for full analysis. Other analysis are performed to measure and evaluate the performance and robustness of the DDPG algorithm. All further model analysis are performed in the 2D environment using the 2D models.

4.3.1 Control Robustness on Dynamics

The trained 2D models listed in Table 3.3 were all trained on dynamic models with distinctive configurations. To test the robustness of the DDPG algorithm, some cross inference experiments were performed. This involved the use of a single 2D model to perform inference experiments on vehicle dynamics it was not trained on. The $\mathcal{A}_{2,7}^{'}$ was used to control all other 2D dynamic models described in Table 3.1 that it was not trained on.

2D Dynamic Models	r_T
$\Lambda_{2,1}$	-144
$\Lambda_{2,2}$	-1999
$\Lambda_{2,3}$	-392

Table 4.1: Inferences score of $\mathcal{A}_{2,7}^{'}$ controlling dynamic models it was not trained on.

Surprisingly, the model $\mathcal{A}_{2,7}^{'}$ performed better on a dynamic model it was not trained on compared to the one it was trained on. Recall that the best score achieved by $\mathcal{A}_{2,7}^{'}$ on $\Lambda_{2,4}$ is -177, the model achieves a higher score of -144 on $\Lambda_{2,1}$. In contrast, it achieves a bad score on $\Lambda_{2,2}$. By observing the dynamic parameters used to describe the dynamic

models, the models $\Lambda_{2,2}$ and $\Lambda_{2,4}$ have more in common compared to $\Lambda_{2,1}$ and $\Lambda_{2,4}$. Therefore, it would be expected that $\mathcal{A}_{2,7}^{'}$ performs better on $\Lambda_{2,2}$ but this is not the case. By also observing $\Lambda_{2,3}$, it can be assumed that the cause of this behaviour is the large difference of parameter C_{m2} in $\Lambda_{2,2}$ and $\Lambda_{2,4}$. This means that the DDPG model is robust to changes in B, C and D as shown in $\Lambda_{2,1}$ and $\Lambda_{2,3}$ results but does not share this property with parameter C_{m2}.

4.3.2 Optimal Dynamic Parameters

So far, searches have been performed to find the optimal controller that can control a dynamic model to achieve optimal drifting. In retrospect, an analysis was performed to find the "optimal" dynamic vehicle that a controller can control to achieve optimal drifting. This was performed by searching for the optimal combination of dynamic parameters that a trained DDPG model can control to achieve optimal drifting. A Bayesian Search function from [29] was used to perform the search.

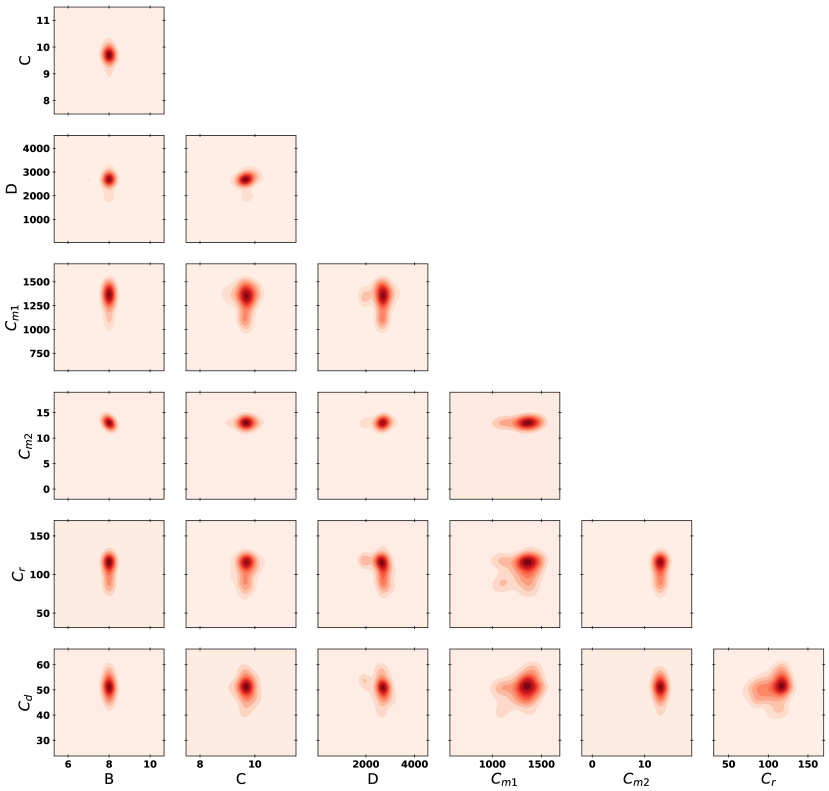

Figure 4.4: Contour plot of sampled points during Bayesian Search.

Figure 4.4 shows the contour plot of sampled points during Bayesian Search. The intensity of the red colour in the map represents the number of sampled points within the search space. It is expected that as the search iteration increases, the closer the sampled points

are to the optimal point. A total reward score of -43 was achieved by $\mathcal{A}_{2,7}^{'}$ on the optimal dynamic parameters found by the Bayesian search. The "optimal" dynamic parameters are given as: $D=2733, C=9.71, B=8, C_{m1}=1305, C_{m2}=13, C_{r}=90$ and $C_{d}=53$. The search spaces of individual dynamic parameters are provided in Dynamic Parameter Search Space.

5 Conclusion

In general, the project was a success and the main aim of the project was achieved. The fact that optimal drifting was not achieved on the 3D models might seem like a negative progress but it should be noted that the project mainly focused on the 2D implementations. The project initially started out on the 2D models and then moved on 3D models after observed success on the 2D models. Given enough time, the 3D models can be improved to achieve optimal drifting.

The learning behaviours of the DDPG algorithm were recognised through the experiments and analysis performed. It was observed that the learning process is not always progressive as the algorithm tries to explore other possible solutions. It was concluded that this behaviour can prevent the models from getting stuck in a local minimum. Further analysis were performed to benchmark the robustness of the control models. This involved the use of a trained DDPG model to control dynamic models it was not trained on. The results from this analysis showed that the DDPG model is robust to some parameter changes but not all. Additional analysis were also performed to locate the optimal combination of dynamic parameters that a trained model can control. The performance observed on the optimal dynamics outclassed all performance score obtained during generic training and inference.

The results and observation made in this project highlights that the DDPG algorithm has potential applications to other control problems. It also reinforces the use of data-driven algorithm for autonomous control tasks.

However, further analysis and improvements can still be implemented. Potential further work could be to initialize the 3D learning parameters with the 2D's before training. This can accelerate learning as there are some dynamic correlation between the 2D and 3D models. Additionally the algorithm can be implemented on a RCV for optimal drifting. This can be used to benchmark DDPG against PILCO which already has a successful implementation on a RCV.

6 References

[1] R. D. Ahmad Abu Hatab, 'Dynamic Modelling of Differential-Drive Mobile Robots using Lagrange and Newton-Euler Methodologies: A Unified Framework', *Advances in Robotics & Automation*, vol. 02, no. 02, 2013 [Online]. Available: https://www.omicsgroup.org/journals/dynamic-modelling-of-differentialdrive-mobile-robots-using-lagrange-and-newtoneuler-methodologies-a-unified-framework-2168-9695.1000107.php?aid=19201. [Accessed: 17-Mar-2019]

[2] R. Siegwart and I. R. Nourbakhsh, *Introduction to autonomous mobile robots*. Cambridge, Mass: MIT Press, 2004.

[3] D. King-Hele, 'Erasmus Darwin's Improved Design for Steering Carriages--And Cars', *Notes and Records of the Royal Society of London*, vol. 56, no. 1, pp. 41–62, 2002.

[4] V. Singh and S. Sharma, 'A Review on Mechanical Design of a Four Wheel Steering System', p. 7.

[5] A. Kolekar, S. Mulani, A. Nerkar, and S. Borchate, 'Review on Steering Mechanism', *IJSART*, vol. 3, no. 4, Apr. 2017.

[6] C. Raut, R. Chettiar, N. Shah, C. Prasad, and D. Bhandarkar, 'Design, Analysis and Fabrication of Steering System Used in Student Formula Car', vol. 5, p. 12, 2017.

[7] R. Rajamani, *Vehicle dynamics and control*, 2. ed. New York, NY: Springer, 2012.

[8] L. Deepak, D. Parhi, and A. K. Jha, 'Kinematic Model of Wheeled Mobile Robots', *Recent Trends in Engineering & Technology,*.

[9] E. Bakker, L. Nyborg, and H. B. Pacejka, 'Tyre Modelling for Use in Vehicle Dynamics Studies', SAE International, Warrendale, PA, SAE Technical Paper 870421, Feb. 1987 [Online]. Available: https://www.sae.org/publications/technical-papers/content/870421/. [Accessed: 20-Mar-2019]

[10] H. Peng and M. Tomizuka, 'Vehicle Lateral Control for Highway Automation', in *1990 American Control Conference*, 1990, pp. 788–794.

[11] A. Liniger, A. Domahidi, and M. Morari, 'Optimization-Based Autonomous Racing of 1:43 Scale RC Cars', *Optimal Control Applications and Methods*, vol. 36, no. 5, pp. 628–647, Sep. 2015.

[12] E. Velenis, E. Frazzoli, and P. Tsiotras, 'On steady-state cornering equilibria for wheeled vehicles with drift', in *Proceedings of the 48h IEEE Conference on Decision*

and Control (CDC) held jointly with 2009 28th Chinese Control Conference, 2009, pp. 3545–3550.

[13] E. Velenis, D. Katzourakis, E. Frazzoli, P. Tsiotras, and R. Happee, 'Steady-state drifting stabilization of RWD vehicles', *Control Engineering Practice*, vol. 19, no. 11, pp. 1363–1376, Nov. 2011.

[14] R. T. Uil, 'Tyre models for steady-state vehicle handling analysis', p. 111.

[15] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, Second edition. Cambridge, Massachusetts: The MIT Press, 2018.

[16] M. E. Harmon and S. S. Harmon, 'Reinforcement Learning: A Tutorial.':, Defense Technical Information Center, Fort Belvoir, VA, Jan. 1997 [Online]. Available: http://www.dtic.mil/docs/citations/ADA323194. [Accessed: 21-Mar-2019]

[17] V. Mnih *et al.*, 'Human-level control through deep reinforcement learning', *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[18] T. P. Lillicrap *et al.*, 'Continuous control with deep reinforcement learning', *arXiv:1509.02971 [cs, stat]*, Sep. 2015 [Online]. Available: http://arxiv.org/abs/1509.02971. [Accessed: 21-Mar-2019]

[19] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, 'Deterministic Policy Gradient Algorithms', p. 9.

[20] M. P. Deisenroth and C. E. Rasmussen, 'PILCO: A Model-Based and Data-Efficient Approach to Policy Search', p. 8.

[21] N. Rontsis, *PILCO: A Python implementation of the PILCO algorithm in TensorFlow*. 2019 [Online]. Available: https://github.com/nrontsis/PILCO. [Accessed: 22-Mar-2019]

[22] OpenAI, *GYM: A toolkit for developing and comparing reinforcement learning algorithms.* OpenAI, 2019 [Online]. Available: https://github.com/openai/gym. [Accessed: 22-Mar-2019]

[23] S. Bhattacharjee, K. D. Kabara, and R. Jain, 'AUTONOMOUS DRIFTING RC CAR WITH REINFORCEMENT LEARNING', Final Report, May 2018.

[24] M. Cutler and J. P. How, 'Autonomous drifting using simulation-aided reinforcement learning', in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 5442–5448.

[25] J.-B. PASSOT, *ROS packages for simulating a vehicle with Ackermann steering: jbpassot/ackermann_vehicle*. 2019 [Online]. Available: https://github.com/jbpassot/ackermann_vehicle. [Accessed: 27-Mar-2019]

[26] Open Source Robotics Foundation, 'Gazebo - A dynamic multi-robot 3D simulator'. [Online]. Available: http://gazebosim.org/. [Accessed: 27-Mar-2019]

[27] G. E. Uhlenbeck and L. S. Ornstein, 'On the Theory of the Brownian Motion', *Phys. Rev.*, vol. 36, no. 5, pp. 823–841, Sep. 1930.

[28] D. P. Kingma and J. Ba, 'Adam: A Method for Stochastic Optimization', *arXiv:1412.6980 [cs]*, Dec. 2014 [Online]. Available: http://arxiv.org/abs/1412.6980. [Accessed: 29-Mar-2019]

[29] *Sequential model-based optimization.* scikit-optimize, 2019 [Online]. Available: https://github.com/scikit-optimize/scikit-optimize. [Accessed: 04-Apr-2019]

[30] Z. Oguntimehin, *Autonomous Vehicle Drifting using DDPG Reinforcement Learning algorithm.: zadiq/auto_drift*. 2019 [Online]. Available: https://github.com/zadiq/auto_drift. [Accessed: 05-Apr-2019]

7 Appendices

7.1 Further Results and Analysis

7.1.1 Training Results

Provided below in Table 7.1 are the best scores achieved by each dynamic model during training and inference. The table also provides the episode j at which the scores were attained.

3D DDPG Models	Training		Inference	
	j	r_T	j	r_T
$\mathcal{A}'_{2,1}$	38	-171	86	-228
$\mathcal{A}'_{2,2}$	**48**	**-103**	53	-246
$\mathcal{A}'_{2,3}$	60	-145	20	-271
$\mathcal{A}'_{2,4}$	69	-270	**69**	**-165**
$\mathcal{A}'_{2,5}$	73	-229	27	-348
$\mathcal{A}'_{2,6}$	76	-110	15	-199
$\mathcal{A}'_{2,7}$	13	-148	41	-177
$\mathcal{A}'_{3,1}$	90	-686	90	-691
$\mathcal{A}'_{3,2}$	23	-765	13	-773
$\mathcal{A}'_{3,3}$	20	-948	20	-968

Table 7.1: List of total sum of rewards attained during training and inference of the DDPG models.

7.1.2 Dynamic Parameter Search Space

During the optimal dynamic parameter search discussed in Optimal Dynamic Parameters, the search space for each parameter were provided to the Bayesian search function.

Dynamic Parameters	Search Space	Optimal Found Value
B	$x \in \mathbb{Z} \mid 6 \leq x \leq 10$	8
C	$x \in \mathbb{R} \mid 8 \leq x \leq 11$	9.71
D	$x \in \mathbb{Z} \mid 500 \leq x \leq 4000$	2733
C_{m1}	$x \in \mathbb{Z} \mid 750 \leq x \leq 1500$	1305

C_{m2}	$x \in \mathbb{Z} \mid 1 \leq x \leq 17$	13
C_r	$x \in \mathbb{Z} \mid 50 \leq x \leq 150$	90
C_d	$x \in \mathbb{Z} \mid 30 \leq x \leq 60$	53

Table 7.2: Lists of dynamic search space provided to the Bayesian algorithm for optimal search. The optimal dynamic values that the model $\mathcal{A}'_{2,7}$ can control is provided under Optimal Found Value.

7.1.3 Action and State Analysis

The actions predicted by two DDPG models during inference are analysed for understanding how the model attains drifting.

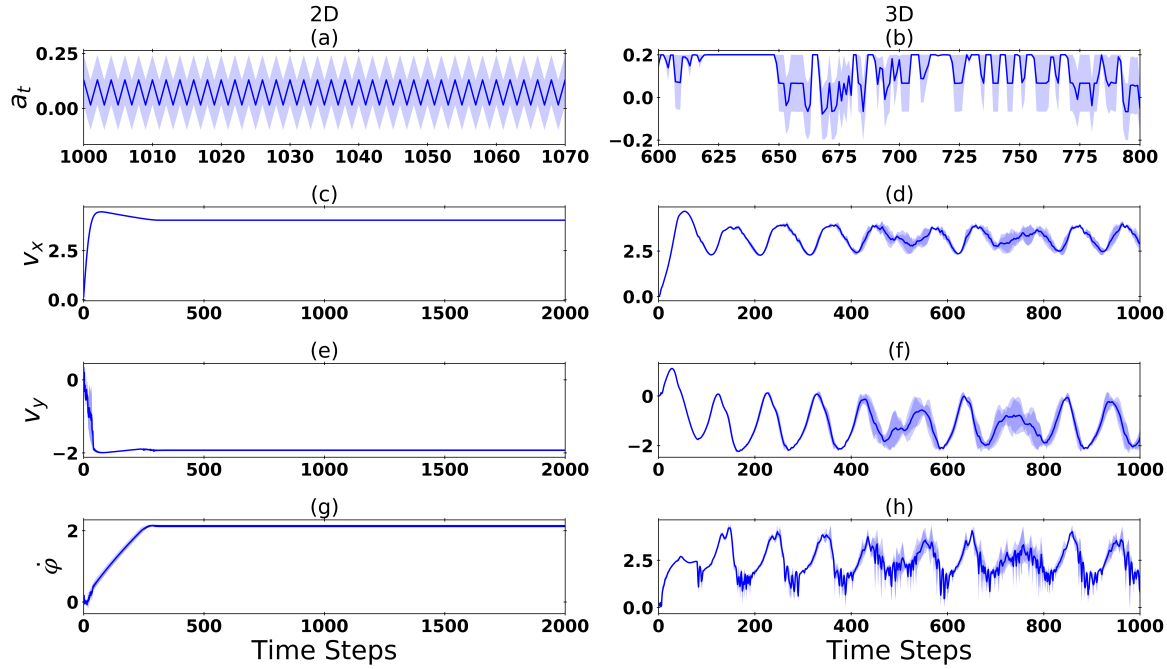

Figure 7.1: Illustrations of the actions predicted by DDPG models during inference and the resulting state transitions of the dynamic models they were applied on.

On left column of Figure 7.1 are the predicted actions (a) of the DDPG model $\mathcal{A}'_{2,7}$ and the states of the dynamic model $\Lambda_{2,4}$ which are illustrated in graph (c), (e) and (g). On the right column are the related values of 3D DDPG model $\mathcal{A}'_{3,1}$ controlling dynamic model $\Lambda_{3,1}$. It can be observed in graph (a) that $\mathcal{A}'_{2,7}$ provides a high frequency alternating steering control to achieve drifting. The steering values are periodic and this is during the time that the dynamic model has reached steady drifting state. Observe the values of the dynamic state in graph (c), (e) and (g), a steady state of the desired state $s_d = [4.1, -2.1, 2.3]$ is achieved. This is the desired state defined for the DDPG model to find as discussed in 2D Model Training.

In contrast, the actions predicted by the 3D as illustrated in graph (b) is not periodic and irregular. It is likely that the model is struggling to find the perfect sequence of alternating the steering values. Recall from Table 3.4 that the DDPG model $\mathcal{A}_{3,1}^{'}$ was trained to achieve a drift state of $[3.6, -2.2, 2.1]$. By observing the dynamic states in graph (d), (f) and (h), the model is able to reach this states sometimes but it is unable to maintain it. However, further observation of graph (h) shows that the angular velocity greatly exceeds the desired state of 2.1. This could be because the vehicle wheels are too slippery and the vehicle drifts further than anticipated by the model. For the DDPG model to easily control the vehicle, the friction coefficients of the wheels need to be tuned and defined accordingly.

Another behaviour observed in the 3D model which is not illustrated in the graphs is that some wheels sometimes leave the floor when travelling due to the suspension on each wheels. This introduces further complexity for the DDPG model. For future experiments and analysis, the suspensions can be disabled to reduce the learning complexity.

7.2 3D Model Parameters

The URDF file used to describe the 3D model contains the structural and dynamic parameters of the model.

Parameter	Value	Units
Body Length	0.258	m
Body Width	0.168	m
Body Height	0.01	m
Body Mass	2.788	kg
Wheel Diameter	0.14	m
Wheel Mass	0.29	kg

Table 7.3: Dynamic and structural parameters of 3D models as described in the URDF file.

Some of the parameters and their values have been extracted from the URDF file in [25] and filled out in Table 7.3 for reference purposes.

7.3 Codes

For easier navigation, the full implementations of all codes can be found in [30]. Otherwise, all codes are provided under relevant sub-sections. Most of the codes are implemented in the Python programming language.

7.3.1 2D Model Implementation

Provided below is the code implementation of the 2D dynamic models.

```
import numpy as np
from drift.commons import (
    Vector, Sequence,
    GroupVector, Scale,
    extract_returns
)
from drift.metrics import get_drift_reward


class ETHParameters:
    """
A class used to set ETH model parameters with different implementations.
Different implementations exhibit different behaviours.
"""
    keys = [
        'B', 'C', 'D',
        'Cm1', 'Cm2', 'Cr', 'Cd'
    ]

    def __init__(self, variable, which):
        getattr(self, '_' + which)()
        [setattr(variable, k, getattr(self, k)) for k in self.keys]

    def __iter__(self):
        yield from self.keys

    def _normal(self):
        """
- travels at 7.2 m/s on a straight line
- doesn't drift when driving at angle
"""
        self.B = 10
        self.C = 0.8
        self.D = 2700

        self.Cm1 = 1250
        self.Cm2 = 1
        self.Cr = 100
        self.Cd = 45

    def _drifty(self):
        """
- hard to control
- madness
- B:C{comments}
- 1:6{moves weird @15deg}
- 7:9.5{kinda okay}
"""
        self.B = 8
```

```
        self.C = 9.5  #
        self.D = 2500

        self.Cm1 = 1250
        self.Cm2 = 3  # this must be an odd number
        self.Cr = 100
        self.Cd = 45


class ETHVariables:
    """
Variables Description
---------------------
vx_dot:         linear acceleration in longitudinal direction
vy_dot:         linear acceleration in latitude direction
omega_dot:      angular acceleration in reference to the global frame
X_dot:          linear X velocity in reference to the global frame
Y_dot:          linear Y velocity in reference to the global frame
yaw_dot|omega:  angular velocity in reference to the global frame
yaw:            angular rotation around the z axis
vx:             linear velocity in longitudinal direction
vy:             linear velocity in latitude direction
delta:          steering applied to front wheel
duty_cycle:     duty cycle use to apply velocity to the vehicle
v:              velocity of vehicle
"""

    def __init__(self, imp="normal", dt=0.01, scale=43):
        """
:param imp: parameter implementation
:param dt: delta time
:param scale: value for scaling down model's physical attributes
"""

        variables = [
            'vx_dot', 'vy_dot', 'omega_dot',
            'vx', 'vy', 'omega', 'yaw',
            'X_dot', 'Y_dot',
            'F_fy', 'F_ry', 'F_rx',
            'X', 'Y', 'v', 'radius'
        ]

        self.space_variables = Vector([0] * len(variables), keys=variables)
        self.control_variables = Vector(0, 1, keys=['delta', 'duty_cycle'])

        self.group = GroupVector(self.space_variables, self.control_variables)
        _ = Scale(scale)

        self.mass = _(2500)
        self.Iz = _(300.9)
        self.lf = _(1.1)
        self.lr = _(1.59)

        self.B = None
        self.C = None
        self.D = None
        self.Cm1 = None
        self.Cm2 = None
        self.Cr = None
        self.Cd = None

        # can base dynamic variables directly as dict or a string for
        # choosing a dynamic profile
        if type(imp) == dict:
```

```
            self.__dict__.update(imp)
        else:
            ETHParameters(self, imp)

        self.dt = dt

    def __getitem__(self, item):
        return self.space_variables[item]

    def __setitem__(self, key, value):
        self.space_variables[key] = value

    @staticmethod
    def assert_which(which):
        assert which in ['f', 'r'], ValueError("Invalid which provided")

    @property
    def front_wheel_y_slip(self):  # alpha_f
        """equation (2a)."""
        delta = self.control_variables['delta']
        omega_dot = self.space_variables['omega_dot']
        vy = self.space_variables['vy']
        vx = self.space_variables['vx']

        return delta - np.arctan2((omega_dot*self.lf) + vy, vx)

    @property
    def rear_wheel_y_slip(self):  # alpha_r
        """equation (2b)."""
        omega_dot = self.space_variables['omega_dot']
        vy = self.space_variables['vy']
        vx = self.space_variables['vx']

        return np.arctan2((omega_dot*self.lr) - vy, vx)

    def wheel_y_force(self, which):
        """
Lateral forces of front and rear wheels
equation (2a, 2b)
"""
        self.assert_which(which)
        if which == 'f':
            slip, var_key = self.front_wheel_y_slip, 'F_fy'
        else:
            slip, var_key = self.rear_wheel_y_slip, 'F_ry'

        force = self.D * np.sin(self.C * np.arctan(self.B * slip))
        self.space_variables[var_key] = force

        return force

    @property
    def front_wheel_y_force(self):  # F_fy
        return self.wheel_y_force('f')

    @property
    def rear_wheel_y_force(self):  # F_ry
        return self.wheel_y_force('r')

    @property
    def rear_wheel_x_force(self):
        """
Longitudinal forces on the rear wheel
equation (2c).
```

```
"""
        d = self.control_variables['duty_cycle']
        vx = self.space_variables['vx']

        force = ((self.Cm1 - (self.Cm2 * vx)) * d) - self.Cr - (self.Cd *
(vx**2))
        self.space_variables['F_rx'] = force
        return force


class ETHDesiredStates:

    def __init__(self):
        self.keys = ['vx', 'vy', 'omega']

    def __getitem__(self, item):
        return getattr(self, item)

    @property
    def one(self):
        """
Desired average velocity of 5, tilted around 23.96 deg
:return:
"""
        return Vector(4.5, 2, 3.125, keys=self.keys)

    @property
    def two(self):
        """
Desired average velocity of 5, tilted around 36.87 deg
:return:
"""
        return Vector(4, 3, 3.125, keys=self.keys)

    @property
    def three(self):
        """[NEW]
Desired average velocity of 4.6, tilted around 27.12 deg
:return:
"""
        return Vector(4.1, -2.1, 2.3, keys=self.keys)


class ETHModel:
    """
Based On: Optimization-based autonomous racing of 1:43 scale RC cars
"""

    def __init__(self, ds="one", reward_params=None, angle_mode="r",
                 reward_version="v2", **kwargs):
        """
:param angle_mode: (r, radian | d, degree) to specify the angle mode
of steering values to be provided when using __call__.
:param reward_params: other parameters passed to reward function
:param ds: a string mapped to one of ETHDesiredStates properties.
Used to choose a desired state
:param kwargs: other arguments passed to ETHVariables
"""
        mode_choices = ['r', 'radian', 'd', 'degree']
        assert angle_mode in mode_choices, ValueError(f'Invalid angle '
                                                      f'mode choose from
{mode_choices}')
        self.angle_mode = angle_mode
        self.reward_params = reward_params or {}
```

```
        self.desired_state = ETHDesiredStates()[ds]
        self.reward_func = get_drift_reward(reward_version)
        self.kwargs = kwargs
        self._build()

    def _build(self):
        self.variables = ETHVariables(**self.kwargs)
        self.sequence = Sequence()
        self.action = None

    def update_sequence(self):
        self.sequence(self.variables.group)

    def reset(self):
        self._build()

    def save(self, path):
        """
TODO implement this, reset before saving
:param path:
:return:
"""

    def resolve_angle(self, angle):
        if self.angle_mode in ['r', 'radian']:
            return angle
        return np.radians(angle)

    @extract_returns
    def __call__(self, steering, duty_cycle=None):
        self.action = steering = self.resolve_angle(steering)
        self.variables.control_variables['delta'] = steering

        if duty_cycle is not None:
            self.variables.control_variables['duty_cycle'] = duty_cycle

        self.solve_space_vector()
        self.update_space_variables()

    def simulate(self, func=None, *args, **kwargs):
        if func is not None:
            """Use custom simulation function"""
            return func(self, *args, **kwargs)
        else:
            t = 0
            while t <= 100:
                self.sequence(self.variables.group)
                s = 0.01 if t < 50 else 0
                self(s)
                t += self.variables.dt

    def solve_space_vector(self):
        delta = self.variables.control_variables['delta']

        vx = self.variables['vx']
        vy = self.variables['vy']
        yaw = self.variables['yaw']
        omega = self.variables['omega']
        m = self.variables.mass
        iz = self.variables.Iz
        lf = self.variables.lf
        lr = self.variables.lr
        f_fy = self.variables.front_wheel_y_force
        f_rx = self.variables.rear_wheel_x_force
```

```
        f_ry = self.variables.rear_wheel_y_force

        self.variables['X_dot'] = (vx * np.cos(yaw)) - (vy * np.sin(yaw))
        self.variables['Y_dot'] = (vx * np.sin(yaw)) + (vy * np.cos(yaw))

        self.variables['vx_dot'] = (f_rx - (f_fy * np.sin(delta)) + (m * vy *
omega)) / m
        self.variables['vy_dot'] = (f_ry + (f_fy * np.cos(delta)) - (m * vx *
omega)) / m
        self.variables['omega_dot'] = ((f_fy * lf * np.cos(delta)) - (f_ry *
lr)) / iz

    def update_space_variables(self):
        dt = self.variables.dt

        self.variables['vx'] += (dt * self.variables['vx_dot'])
        self.variables['vy'] += (dt * self.variables['vy_dot'])
        self.variables['v'] = np.sqrt((self.variables['vx']**2) +
(self.variables['vy']**2))
        self.variables['omega'] += (dt * self.variables['omega_dot'])
        self.variables['radius'] = self.variables['v'] / self.variables['omega']
        self.variables['yaw'] += (dt * self.variables['omega'])

        self.variables['X'] += (dt * self.variables['X_dot'])
        self.variables['Y'] += (dt * self.variables['Y_dot'])

    def get_state(self):
        """
Returns a Vector of variables used to represent the
state of the vehicle.
"""
        return self.variables['vx', 'vy', 'omega']

    @staticmethod
    def sample_steering(bound=30):
        """
Returns a random steering value within a range.
"""
        return np.radians(np.random.uniform(-bound, bound))

    def get_reward(self):
        new_state = self.get_state()  # get new_state after performing an action
        return self.reward_func(self.desired_state, new_state,
**self.reward_params)

    @property
    def info(self):
        """
dimension of action and state
:return:
"""
        return Vector([1, 3], keys=['action', 'state'])
```

7.3.2 DDPG Model Implementation

The DDPG algorithm was implemented using the Tensorflow library (Tensorflow, r1.13).

The code is provided below.

```
"""
Implementation of Deep Deterministic Policy Gradient

Based On: CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING
Link: https://arxiv.org/pdf/1509.02971.pdf
Implementation aided with:
https://github.com/pemami4911/deep-rl/blob/master/ddpg/ddpg.py
"""
import tensorflow as tf
import tflearn
import numpy as np
from tflearn.layers import fully_connected
from tflearn.layers.normalization import batch_normalization


class DDPGLayer:
    """Abstract class for a neural layer"""

    def __init__(self, dim, batch=False, activation="relu"):
        self.dim = dim
        self.batch = batch
        self.activation = activation


def build_layers(x, layers):
    """Construct neural layers"""

    for l in layers:
        x = fully_connected(x, l.dim)
        if l.batch:
            x = batch_normalization(x)
        if l.activation:
            x = getattr(tflearn.activations, l.activation)(x)
    return x


def assert_name(name):
    """Check DDPG name"""
    assert name in ['online', 'target']


class DDPGActorCriticBase:
    """Base class for DDPG Actor and Critic"""

    def __init__(self):
        self.weight_params = None
        self.online_network = None
        self.target_network_weights = []

    def set_weight_params(self, weight_params, online_network=None):
        _ = tf.multiply
        self.weight_params = weight_params

        # create tensor for updating target weights using tau
        if online_network:
            self.online_network = online_network
            for i in range(len(self.weight_params)):
```

```
                self.target_network_weights.append(
                    self.weight_params[i].assign(
                        _(self.online_network.weight_params[i], self.params.tau)
+
                        _(self.weight_params[i], 1. - self.params.tau)
                    )
                )

    def update_weights(self):
        """Update a target actor's weight with online weights"""
        self.sess.run(self.target_network_weights)


class DDPGParams:
    """DDPG architecture params"""

    action_dim = 1  # steering
    state_dim = 3  # vx, vy, w
    tau = 0.001  # target update tau
    batch_size = 20
    gamma = 0.99  # discount factor

    # actor's params
    actor_network = [
        (400, False, "relu"),  # dim, batch, activation
        (300, False, "relu"),
    ]
    actor_activator = "tanh"  # output layer activation
    action_bound = np.radians(np.radians(80))  # double rad
    actor_lr = 0.0001

    # critic's params
    critic_network = [
        (400, False, "relu"),  # dim, batch, activation
    ]
    critic_last_layer = 300
    critic_activator = "relu"
    critic_lr = 0.001

    # weights initialisation params
    layer_weights_min = -3 -3
    layer_weights_max = 3 -3


class DDPGActor(DDPGActorCriticBase):
    """
DDPG Actor's Network
An actor takes states as input and predicts an action that leads
to the next state.
"""

    def __init__(self, params: DDPGParams, sess=None, name="online"):
        self.params = params
        self.sess = sess
        assert_name(name)
        self.name = name
        self._build_network()

        super().__init__()
        self.grads = None
        self.optimise = None

        # this will be provided by the critic network
```

```
        self.action_gradient = tf.placeholder(tf.float32, [None,
self.params.action_dim])

    def _build_network(self):
        layers = [DDPGLayer(*l) for l in self.params.actor_network]
        self.states = tflearn.input_data([None, self.params.state_dim])
        x = build_layers(self.states, layers)
        weights_init = tflearn.initializations.uniform(
            minval=self.params.layer_weights_min,
            maxval=self.params.layer_weights_max
        )
        self.action = tflearn.fully_connected(
            x, self.params.action_dim, self.params.actor_activator,
            weights_init=weights_init, name="actor"
        )
        self.scaled_action = tf.multiply(
            self.action, self.params.action_bound,
            name="scaled-actor"
        )

    def set_info(self, weight_params, online_actor=None):
        """
used to set actor variable info
:param weight_params: learning parameters after building network
:param online_actor: provide online actor when setting info for target actor
:return:
"""
        self.set_weight_params(weight_params, online_actor)

        if not online_actor:
            unnorm_gradients = tf.gradients(
                self.scaled_action, self.weight_params,
                -self.action_gradient
            )
            self.grads = list(map(lambda x: tf.div(x, self.params.batch_size),
unnorm_gradients))
            # optimiser = tf.train.AdamOptimizer(self.params.actor_lr)
            self.optimise =
tf.train.AdamOptimizer(self.params.actor_lr).apply_gradients(
                zip(self.grads, self.weight_params)
            )

    def train(self, states, action_grads):
        self.sess.run(self.optimise, feed_dict={
            self.states: states,
            self.action_gradient: action_grads
        })

    def predict(self, states):
        """Predict action based on provided states"""
        return self.sess.run(self.scaled_action, feed_dict={
            self.states: states
        })


class DDPGCritic(DDPGActorCriticBase):
    """
DDPG Critic Network
A critic takes in states and action(s) as inputs and predicts a q_value given
a the old state and action.
"""

    def __init__(self, params: DDPGParams, sess, name="online"):
        self.params = params
```

```
        self.sess = sess
        assert_name(name)
        self.name = name
        self._build()
        super().__init__()

    def _build(self):
        self.states = tflearn.input_data(shape=[None, self.params.state_dim])
        self.actions = tflearn.input_data(shape=[None, self.params.action_dim])
        layers = [DDPGLayer(*l) for l in self.params.critic_network]
        x = build_layers(self.states, layers)

        # As stated in the paper, the action is not added to the layers
        # until the last one for low dimensional action space. Separate
        # last layer is created for actions and then the weights are then
        # combined through addition.
        t1 = fully_connected(x, self.params.critic_last_layer)
        t2 = fully_connected(self.actions, self.params.critic_last_layer)
        _ = tf.matmul
        combined = _(x, t1.W) + _(self.actions, t2.W) + t2.b
        x = tflearn.activation(combined,
activation=self.params.critic_activator)

        weights_init = tflearn.initializations.uniform(
            minval=self.params.layer_weights_min,
            maxval=self.params.layer_weights_max
        )
        self.q_values = fully_connected(x, 1, weights_init=weights_init,
name="critic")

        if self.name == "online":
            # Estimated y value (the future expected reward value also known).
            self.est_y_values = tf.placeholder(tf.float32, [None, 1])
            self.loss = tflearn.mean_square(self.est_y_values, self.q_values)
            optimiser = tf.train.AdamOptimizer(self.params.critic_lr)
            self.optimise = optimiser.minimize(self.loss)
            self.action_gradient = tf.gradients(self.q_values, self.actions)

    def set_info(self, weights_params, online_critic=None):
        self.set_weight_params(weights_params, online_critic)

    def train(self, states, actions, est_y_values):
        return self.sess.run(
            [self.q_values, self.optimise],
            feed_dict={
                self.states: states,
                self.actions: actions,
                self.est_y_values: est_y_values
            }
        )

    def predict(self, states, actions):
        """Predict q_values given state and actions"""
        return self.sess.run(self.q_values, feed_dict={
            self.states: states,
            self.actions: actions
        })

    def get_action_gradients(self, states, actions):
        return self.sess.run(self.action_gradient, feed_dict={
            self.states: states,
            self.actions: actions
        })
```

```
class DDPGSummary:
    """
Handler class for handling summaries
and saving and loading models.
"""

    def __init__(self, sess, sum_path=None, max_to_keep=100):
        """
:param sum_path: path for saving checkpoints
"""
        self.sess = sess
        self.writer = None
        if sum_path:
            self.summary_vars, self.summary_ops = self.build_summaries()
            self.writer = tf.summary.FileWriter(sum_path, sess.graph)
        self.saver = tf.train.Saver(max_to_keep=max_to_keep)

    @staticmethod
    def build_summaries():
        episode_reward = tf.Variable(-1999, trainable=False)
        tf.summary.scalar("Total Reward", episode_reward)
        eps_ave_max_q = tf.Variable(0., trainable=False)
        tf.summary.scalar("Qmax Value", eps_ave_max_q)
        max_reward = tf.Variable(0., trainable=False)
        tf.summary.scalar("Max Reward", max_reward)

        summary_vars = [episode_reward, eps_ave_max_q, max_reward]
        summary_ops = tf.summary.merge_all()

        return summary_vars, summary_ops

    def write_summary(self, ep, values: list, sess=None):
        """Write an episode summary to file"""
        if not self.writer:
            AttributeError("write_summary is only available "
                           "when sum_path is provided")
        sess = sess or self.sess
        feed = dict(zip(self.summary_vars, values))
        summary_str = sess.run(self.summary_ops, feed_dict=feed)
        self.writer.add_summary(summary_str, ep)
        self.writer.flush()

    def save_model(self, path, sess=None):
        sess = sess or self.sess
        self.saver.save(sess, path)

    def load_model(self, path, sess=None):
        sess = sess or self.sess
        self.saver.restore(sess, path)


class DDPG:
    """DDPG main class"""

    def __init__(self, params: DDPGParams, sess):
        self.params = params
        self.var_count = 0  # tf variables count
        self.ep_avg_max_q = 0

        # create Actor(s)
        self.online_actor = DDPGActor(params, sess, "online")
        self.online_actor.set_info(self.get_tf_variables())
```

```
        self.target_actor = DDPGActor(params, sess, "target")
        self.target_actor.set_info(self.get_tf_variables(), self.online_actor)

        # create Critic(s)
        self.online_critic = DDPGCritic(params, sess, "online")
        self.online_critic.set_info(self.get_tf_variables())

        self.target_critic = DDPGCritic(params, sess, "target")
        self.target_critic.set_info(self.get_tf_variables(), self.online_critic)

    def reset(self):
        self.ep_avg_max_q = 0

    def get_tf_variables(self):
        t_var = tf.trainable_variables()[self.var_count:]
        self.var_count += len(t_var)
        return t_var

    def predict(self, states, target=True):
        """
Predict actions given states using either
target or online actor
"""
        if target:
            return self.target_actor.predict(states)
        return self.online_actor.predict(states)

    def train(self, batch):
        """
An episodic train step using data provided as a batch
Batch should have __get_item__ with the following keys:
- current_state
- new_state
- action
- reward
- terminal
- gamma
"""
        current_states = batch['current_state']
        new_states = batch['new_state']
        actions = batch['action']
        rewards = batch['reward']
        terminals = batch['terminal']
        gamma = self.params.gamma

        t_act = self.target_actor.predict(new_states)
        t_q_values = self.target_critic.predict(new_states, t_act)

        # calculate y values for the batch set
        est_y_values = []
        for r, t, t_q in zip(rewards, terminals, t_q_values):
            if t:
                est_y_values.append(r)
            else:
                est_y_values.append(r + (gamma * t_q))
        est_y_values = np.reshape(est_y_values, (len(batch), 1))

        o_q_values, _ = self.online_critic.train(
            current_states, actions,
            est_y_values
        )

        # Update the actor policy using the sampled gradient
        self.ep_avg_max_q += np.amax(o_q_values)
```

```
        a_outs = self.online_actor.predict(current_states)
        grads = self.online_critic.get_action_gradients(current_states, a_outs)
        self.online_actor.train(current_states, grads[0])

        self.update_target_weights()

    def update_target_weights(self):
        self.target_actor.update_weights()
        self.target_critic.update_weights()
```

7.3.3 3D Model Interface

The code provide below is used to interface the URDF in the Gazebo environment for running experiments.

```
class ServerHandler(socketserver.BaseRequestHandler):

    def handle(self):
        raw_data = self.receive_data()
        data = json.loads(raw_data.decode(       ))
# print(data)
        self.server.processor(data)

    def receive_data(self):
        raw_data_len = self.receive_all(4)
        if not raw_data_len:
            return None
        data_len = struct.unpack('>I', raw_data_len)[0]
        return self.receive_all(data_len)

    def receive_all(self, n):
        data = b
        while len(data) < n:
            packet = self.request.recv(n - len(data))
            if not packet:
                return None
            data += packet
        return data


class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass


class Ackermann:
    """
Communicates with the ros package controlling the Ackermann mode
on Gazebo
"""

    def __init__(self, host="localhost", port=9190,
                 action_dim=1, reward_v="v1", desired_state=0,
                 reward_params=None, thr_lim=0):

# create server
        self.server = ThreadedTCPServer((host, port), ServerHandler)
        ip, host = self.server.server_address
        self.server.processor = self.process_data
        server_thread = Thread(target=self.server.serve_forever)
        server_thread.daemon = True
        server_thread.start()
```

```
        print(f"Serving Ackermann interface at ({ip}, {host})")

# controller server address
        self.con_server_address = ("localhost", 9091)

        self.states = None
        self.can_send_actions = False
        self.new_state = None
        self.coord = None
        self.action_dim = action_dim
        self.reward_func = get_drift_reward(reward_v)
        self.reward_params = reward_params or {}
        self.desired_state = self.get_desired_state[desired_state]
        self.sequence = Sequence()
        self.dt = 0.01
        self.throttle_lim = thr_lim

    @property
    def get_desired_state(self):
        keys = ['vx', 'vy',        ]
        return [                                    # ix    mu1     mu2
            Vector(2.1, -1.17, 1.6, keys=keys),     # 0
            Vector(3.5, -2, 2, keys=keys),          # 1
            Vector(4.1, -2.1, 2.3, keys=keys),      # 2
            Vector(3.6, -2.2, 2.1, keys=keys),      # 3
            Vector(4.1, -1.9, 2.1, keys=keys),      # 4
            Vector(2.1, -3.8, 2, keys=keys),        # 5
            Vector(1, -2, 2, keys=keys),            # 6
            Vector(1.2, -2.6, 2, keys=keys),        # 7     0.2     0.4
            Vector(2, -2, 2, keys=keys),            # 8     0.4     0.4
            Vector(1, -1, 2, keys=keys),            # 9     0.4     0.4
        ]

    @extract_returns
    def __call__(self, steering=0.15, throttle=4):
        """Apply actions"""
        if self.action_dim > 1:
            throttle += self.throttle_lim
            self.action = [steering, throttle]
else:
            self.action = steering
# self.action = steering if self.action_dim == 1 else [steering, throttle]
        while not self.can_send_actions:
            """Wait until controller is ready to receive action"""
            time.sleep(.0001)

# disable sending action until the controller is ready
        self.can_send_actions = False
        data = {
                    : {
                "throttle": throttle,
                "steering": steering
            }
        }
        self.send_data(data)

        while not self.can_send_actions:
            """wait until controller receives data"""
            time.sleep(.0001)

    def send_data(self, data):
        ros_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ros_client.settimeout(3)
        ros_client.connect(self.con_server_address)
```

```
        data = bytes(json.dumps(data),         )
        data = struct.pack('>I', len(data)) + data
        ros_client.sendall(data)
        ros_client.close()

    def get_state(self):
        if self.states is None:
# print("hit here")
            self.send_data(
                                : True
             )
            while not self.states:
                time.sleep(.0001)

        states = self.states
        self.states = None
# self.coord = None
        return states

    def reset(self):
        self.sequence.clear()
        self.states = None
        self.send_data(
                  : True
         )
        time.sleep(0.1)

    def process_data(self, data):
        """Process incoming data from Ackermann Controller"""
        response = {
                  : False
        }

        if response.get("error"):
            print(response[       ])

        if data.get("request_action"):
            """Store states"""
            self.can_send_actions = data[               ]

        if data.get("states"):
            """store states when it is sent by the controller"""
            states = data.get("states")
            self.states = Vector(
                [states['vx'], states['vy'], states[       ]],
                keys=['vx', 'vy',        ]
            )
            self.coord = Vector(
                [states['x'], states['y'], states['yaw']],
                keys=['x', 'y', 'yaw']
            )

    @staticmethod
    def sample_action(bound):
        return np.random.uniform(-bound, bound)

    def simulate(self, steering=.15, throttle=4):
        _ = self.sample_action
        dt = 0.01
        t = 0
        i = 0
        start = time.time()
        while t <= 5:
            self.sequence(self(steering, throttle))
```

```
# print(i, self.get_reward())
            t += dt
            i += 1
            time.sleep(1/60)
        print(f"Finish time: {time.time() - start}")

    def simulate_drift(self, steering=0.15, throttle=4):
        intervals = 20, 5
        t = 0
        i = 0
        s = 0
        dt = 0.01
        while t <= 5:
            if i <= intervals[0]:  # left turn
                s = steering
            elif intervals[0] < i < np.sum(intervals):  # right turn
                s = -steering
else:  # reset
                i = 0
            self.sequence(self(s, throttle))
            i += 1
            t += dt

    def get_reward(self):
# new_state = self.get_state() can get new_state filled in
# call_return instead
        new_state = self.new_state
        return self.reward_func(self.desired_state, new_state,
**self.reward_params)


if __name__ ==           :
    ack = Ackermann()
```

7.3.4 3D Model URDF

The code below describes the dynamic model of the 3D model in the Unified Robot Description Format.

```
<?xml version="1.0"?>

<!-- em_3905.urdf.xacro

This file defines a model of a Traxxas(R) E-Maxx(R) #3905 RC (Radio Controlled)
truck.

Lengths are measured in meters, angles are measured in radians, and masses are
measured in kilograms. All of these values are approximations.

To work with Gazebo, each link must have an inertial element, even if
the link only serves to connect two joints. To be visible in Gazebo, a link
must have a collision element. Furthermore, the link must have a Gazebo
material.

Traxxas(R), E-Maxx(R), and Titan(R) are registered trademarks of Traxxas
Management, LLC. em_3905.urdf.xacro was independently created by Wunderkammer
Laboratory, and neither em_3905.urdf.xacro nor Wunderkammer Laboratory is
affiliated with, sponsored by, approved by, or endorsed by Traxxas Management,
LLC. Mabuchi Motor(R) is a registered trademark of Mabuchi Motor Co., Ltd.
Corporation Japan. All other trademarks and service marks are the property of
their respective owners.
```

```
Copyright (c) 2011-2014 Wunderkammer Laboratory

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
-->

<robot name="em_3905" xmlns:xacro="http://www.ros.org/wiki/xacro">
  <!-- Degree-to-radian conversions -->
  <xacro:property name="degrees_45" value="0.785398163"/>
  <xacro:property name="degrees_90" value="1.57079633"/>

  <!-- chassis_length is measured along the x axis, chassis_width
along the y axis, and chassis_height along the z axis. -->
  <xacro:property name="chassis_length" value="0.258"/>
  <xacro:property name="chassis_width" value="0.168"/>
  <xacro:property name="chassis_height" value="0.01"/>
  <xacro:property name="chassis_mass" value="2.788"/>

  <!-- battery_dist is the distance between the inner edges of the
batteries. battery_comp_depth is the battery compartment depth.
battery_length is measured along the x axis, battery_width
along the y axis, and battery_height along the z axis. -->
  <xacro:property name="battery_x_offset" value="0.055"/>
  <xacro:property name="battery_dist" value="0.065"/>
  <xacro:property name="battery_comp_depth" value="0.02"/>
  <xacro:property name="battery_comp_angle" value="0.34906585"/>
  <xacro:property name="battery_length" value="0.16"/>
  <xacro:property name="battery_width" value="0.047"/>
  <xacro:property name="battery_height" value="0.024"/>
  <xacro:property name="battery_mass" value="0.5025"/>

  <!-- hub_dia and tire_dia are the diameters of the hub and tire,
respectively. hex_hub_depth is the distance that the hex hub is
inset from the outer edge of the tire. It is set so that each wheel
is a "zero offset" wheel. hex_hub_depth = tire_width / 2 -
axle_length. -->
  <xacro:property name="hub_dia" value="0.09652"/>
  <xacro:property name="tire_dia" value="0.14605"/>
  <xacro:property name="tire_width" value="0.0889"/>
  <xacro:property name="hex_hub_depth" value="0.01445"/>
  <xacro:property name="wheel_mass" value="0.29"/>

  <!-- hex_hub_dist is the distance between left and right hex hubs when
the shock absorbers are fully extended. axle_length is the distance
from a U joint to the corresponding hex hub. wheel_travel is the
vertical wheel travel. -->
  <xacro:property name="wheelbase" value="0.335"/>
  <xacro:property name="hex_hub_dist" value="0.365"/>
  <xacro:property name="axle_length" value="0.03"/>
  <xacro:property name="wheel_travel" value="0.084"/>
  <xacro:property name="shock_z_offset" value="0.0655"/>

  <!-- shock_eff_limit is 2 * ((shock_stroke / 2) * shock_spring_constant) N.
shock_stroke is 0.028575 meters. shock_spring_constant, an approximation
```

```
of a Traxxas Ultra Shock shock absorber spring's constant, is
437.817 N/m. -->
  <xacro:property name="shock_eff_limit" value="12.5106"/>
  <xacro:property name="shock_vel_limit" value="1000"/>

  <!-- The specifications for a Titan(R) 550 motor could not be found, so the
stall torque of a Mabuchi Motor(R) RS-550VC-7525 motor was used instead.

num_spur_gear_teeth = 68
num_pinion_gear_teeth = 19
final_gear_ratio = (num_spur_gear_teeth / num_pinion_gear_teeth) *
5.22 = 18.68
stall_torque = 0.549 N m
axle_eff_limit = ((2 * stall_torque) * final_gear_ratio) / 4 =
5.12766 N m

max_speed = 40 mph (30+ mph) = 17.8816 m/s
axle_vel_limit = (2 * pi) * (max_speed / (pi * tire_dia)) =
244.8696 rad/s -->
  <xacro:property name="axle_eff_limit" value="5.12766"/>
  <xacro:property name="axle_vel_limit" value="244.8696"/>

  <!-- These constants are used to simulate a Traxxas 2056 servo operated at
6 V. servo_stall_torque is measured in N m. servo_no_load_speed is
measured in rad/s. -->
  <xacro:property name="servo_stall_torque" value="0.5649"/>
  <xacro:property name="servo_no_load_speed" value="4.553"/>

  <material name="battery_mat">
    <color rgba="0 0 1 1"/>
  </material>
  <material name="chassis_mat">
    <color rgba="0.5 0.5 0.5 1"/>
  </material>
  <material name="tire_mat">
    <color rgba="0 0 0 1"/>
  </material>

  <!-- Null inertial element. This is needed to make the model work with
Gazebo. -->
  <xacro:macro name="null_inertial">
    <inertial>
      <mass value="0.001"/>
      <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
    </inertial>
  </xacro:macro>

  <!-- Inertia of a solid cuboid. Width is measured along the x axis, depth
along the y axis, and height along the z axis. -->
  <xacro:macro name="solid_cuboid_inertial"
               params="width depth height mass *origin">
    <inertial>
      <xacro:insert_block name="origin"/>
      <mass value="${mass}"/>
      <inertia ixx="${mass * (depth * depth + height * height) / 12}"
               ixy="0" ixz="0"
               iyy="${mass * (width * width + height * height) / 12}"
               iyz="0"
               izz="${mass * (width * width + depth * depth) / 12}"/>
    </inertial>
  </xacro:macro>

  <!-- Inertia of a thick-walled cylindrical tube with open ends. Height is
measured along the z axis, which is the tube's axis. inner_rad and
```

```
outer_rad are the tube's inner and outer radii, respectively. -->
  <xacro:macro name="thick_walled_tube_inertial"
               params="inner_rad outer_rad height mass">
    <inertial>
      <mass value="${mass}"/>
      <inertia ixx="${(1 / 12) * mass * (3 * (inner_rad * inner_rad +
outer_rad * outer_rad) + height * height)}"
               ixy="0" ixz="0"
               iyy="${(1 / 12) * mass * (3 * (inner_rad * inner_rad +
outer_rad * outer_rad) + height * height)}"
               iyz="0"
               izz="${mass * (inner_rad * inner_rad +
outer_rad * outer_rad) / 2}"/>
    </inertial>
  </xacro:macro>

  <!-- Battery -->
  <xacro:macro name="battery" params="prefix reflect">
    <joint name="chassis_to_${prefix}_battery" type="fixed">
      <parent link="chassis"/>
      <child link="${prefix}_battery_offset"/>
      <origin xyz="${battery_x_offset - battery_length / 2}
${reflect * battery_dist / 2}
0"
              rpy="${reflect * battery_comp_angle} 0 0"/>
    </joint>

    <link name="${prefix}_battery_offset">
      <xacro:null_inertial/>
    </link>

    <joint name="offset_to_${prefix}_battery" type="fixed">
      <parent link="${prefix}_battery_offset"/>
      <child link="${prefix}_battery"/>
      <origin xyz="0
${reflect * battery_width / 2}
${(battery_height / 2) - battery_comp_depth}"/>
    </joint>

    <link name="${prefix}_battery">
      <visual>
        <geometry>
          <box size="${battery_length} ${battery_width} ${battery_height}"/>
        </geometry>
        <material name="battery_mat"/>
      </visual>

      <collision>
        <geometry>
          <box size="${battery_length} ${battery_width} ${battery_height}"/>
        </geometry>
      </collision>

      <xacro:solid_cuboid_inertial
         width="${battery_length}" depth="${battery_width}"
         height="${battery_height}" mass="${battery_mass}">
        <origin/>
      </xacro:solid_cuboid_inertial>
    </link>

    <gazebo reference="${prefix}_battery">
      <material>Gazebo/Blue</material>
    </gazebo>
  </xacro:macro>
```

```
  <!-- Shock absorber -->
  <xacro:macro name="shock"
               params="lr_prefix fr_prefix lr_reflect fr_reflect child">
    <joint name="${lr_prefix}_${fr_prefix}_shock" type="prismatic">
      <parent link="chassis"/>
      <child link="${child}"/>

      <origin xyz="${fr_reflect * wheelbase / 2}
${lr_reflect * ((hex_hub_dist / 2) - axle_length)}
${(wheel_travel / 2) - shock_z_offset}"/>
      <axis xyz="0 0 -1"/>
      <limit lower="${-wheel_travel / 2}" upper="${wheel_travel / 2}"
             effort="${shock_eff_limit}" velocity="${shock_vel_limit}"/>
    </joint>
    <transmission name="${lr_prefix}_${fr_prefix}_shock_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="${lr_prefix}_${fr_prefix}_shock">
        <hardwareInterface>EffortJointInterface</hardwareInterface>
      </joint>
      <actuator name="${lr_prefix}_${fr_prefix}_shock_act">
        <!-- This hardwareInterface element exists for compatibility
with ROS Hydro. -->
        <hardwareInterface>EffortJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>
  </xacro:macro>

  <!-- The "wheel" macro defines an axle carrier, axle, and wheel. -->
  <xacro:macro name="wheel" params="lr_prefix fr_prefix lr_reflect">
    <link name="${lr_prefix}_${fr_prefix}_axle_carrier">
      <xacro:null_inertial/>
    </link>

    <!-- The left and right axles have the same axis so that identical
rotation values cause the wheels to rotate in the same direction. -->
    <joint name="${lr_prefix}_${fr_prefix}_axle" type="continuous">
      <parent link="${lr_prefix}_${fr_prefix}_axle_carrier"/>
      <child link="${lr_prefix}_${fr_prefix}_wheel"/>
      <origin rpy="${degrees_90} 0 0"/>
      <axis xyz="0 0 -1"/>
      <limit effort="${axle_eff_limit}" velocity="${axle_vel_limit}"/>
    </joint>
    <transmission name="${lr_prefix}_${fr_prefix}_axle_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="${lr_prefix}_${fr_prefix}_axle">
        <hardwareInterface>EffortJointInterface</hardwareInterface>
      </joint>
      <actuator name="${lr_prefix}_${fr_prefix}_axle_act">
        <!-- This hardwareInterface element exists for compatibility
with ROS Hydro. -->
        <hardwareInterface>EffortJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>

    <link name="${lr_prefix}_${fr_prefix}_wheel">
      <visual>
        <origin xyz="0
${lr_reflect * (axle_length - (tire_width /
2 - hex_hub_depth))}
0"/>
        <geometry>
```

```
          <cylinder radius="${tire_dia / 2}" length="${tire_width}"/>
        </geometry>
        <material name="tire_mat"/>
      </visual>

      <collision>
        <origin xyz="0
${lr_reflect * (axle_length - (tire_width /
2 - hex_hub_depth))}
0"/>
        <geometry>
          <cylinder radius="${tire_dia / 2}" length="${tire_width}"/>
        </geometry>
      </collision>

      <xacro:thick_walled_tube_inertial
          inner_rad="${hub_dia / 2}" outer_rad="${tire_dia / 2}"
          height="${tire_width}" mass="${wheel_mass}"/>
    </link>

    <gazebo reference="${lr_prefix}_${fr_prefix}_wheel">
      <material>Gazebo/Black</material>
      <mu1>0.4</mu1>
      <mu2>0.4</mu2>
    </gazebo>
  </xacro:macro>

  <!-- Front wheel -->
  <xacro:macro name="front_wheel"
               params="lr_prefix fr_prefix lr_reflect fr_reflect">
    <xacro:shock lr_prefix="${lr_prefix}" fr_prefix="${fr_prefix}"
                 lr_reflect="${lr_reflect}" fr_reflect="${fr_reflect}"
                 child="${lr_prefix}_steering_link"/>

    <link name="${lr_prefix}_steering_link">
      <xacro:null_inertial/>
    </link>

    <joint name="${lr_prefix}_steering_joint" type="revolute">
      <parent link="${lr_prefix}_steering_link"/>
      <child link="${lr_prefix}_${fr_prefix}_axle_carrier"/>
      <axis xyz="0 0 1"/>
      <limit lower="${-degrees_45}" upper="${degrees_45}"
             effort="${servo_stall_torque}" velocity="${servo_no_load_speed}"/>
    </joint>
    <transmission name="${lr_prefix}_steering_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="${lr_prefix}_steering_joint">
        <hardwareInterface>EffortJointInterface</hardwareInterface>
      </joint>
      <actuator name="${lr_prefix}_steering_act">
        <!-- This hardwareInterface element exists for compatibility
with ROS Hydro. -->
        <hardwareInterface>EffortJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>

    <xacro:wheel lr_prefix="${lr_prefix}" fr_prefix="${fr_prefix}"
                 lr_reflect="${lr_reflect}"/>
  </xacro:macro>

  <!-- Rear wheel -->
  <xacro:macro name="rear_wheel"
```

```
                params="lr_prefix fr_prefix lr_reflect fr_reflect">
    <xacro:shock lr_prefix="${lr_prefix}" fr_prefix="${fr_prefix}"
                 lr_reflect="${lr_reflect}" fr_reflect="${fr_reflect}"
                 child="${lr_prefix}_${fr_prefix}_axle_carrier"/>
    <xacro:wheel lr_prefix="${lr_prefix}" fr_prefix="${fr_prefix}"
                 lr_reflect="${lr_reflect}"/>
  </xacro:macro>

  <!-- base_link must have geometry so that its axes can be displayed in
rviz. -->
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.01 0.01 0.01"/>
      </geometry>
      <material name="chassis_mat"/>
    </visual>
  </link>
  <gazebo reference="base_link">
    <material>Gazebo/Grey</material>
  </gazebo>

  <!-- Chassis -->
  <link name="chassis">
    <visual>
      <origin xyz="0 0 ${-chassis_height / 2}"/>
      <geometry>
        <box size="${chassis_length} ${chassis_width} ${chassis_height}"/>
      </geometry>
      <material name="chassis_mat"/>
    </visual>

    <collision>
      <origin xyz="0 0 ${-chassis_height / 2}"/>
      <geometry>
        <box size="${chassis_length} ${chassis_width} ${chassis_height}"/>
      </geometry>
    </collision>

    <xacro:solid_cuboid_inertial
        width="${chassis_length}" depth="${chassis_width}"
        height="${chassis_height}" mass="${chassis_mass}">
      <origin xyz="0 0 ${-chassis_height / 2}"/>
    </xacro:solid_cuboid_inertial>
  </link>
  <gazebo reference="chassis">
    <material>Gazebo/Grey</material>
  </gazebo>

  <joint name="base_link_to_chassis" type="fixed">
    <parent link="base_link"/>
    <child link="chassis"/>
  </joint>

  <!-- Batteries -->
  <xacro:battery prefix="left" reflect="1"/>
  <xacro:battery prefix="right" reflect="-1"/>

  <!-- Wheels -->
  <xacro:front_wheel lr_prefix="left" fr_prefix="front"
                     lr_reflect="1" fr_reflect="1"/>
  <xacro:front_wheel lr_prefix="right" fr_prefix="front"
                     lr_reflect="-1" fr_reflect="1"/>
  <xacro:rear_wheel lr_prefix="left" fr_prefix="rear"
```

```
                    lr_reflect="1" fr_reflect="-1"/>
  <xacro:rear_wheel lr_prefix="right" fr_prefix="rear"
                    lr_reflect="-1" fr_reflect="-1"/>

  <gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so"/>
  </gazebo>
</robot>
```

7.3.5 Miscellaneous Codes

Other supporting functions and classes are provided below.

```
import matplotlib.pyplot as plt
import numpy as np
import pickle
import time
import os
import json
from matplotlib.animation import FuncAnimation
from matplotlib.transforms import Affine2D
from collections import OrderedDict, deque
from functools import wraps


def is_iter(obj):
    """
Check if an object is iterable
"""

    try:
        iter(obj)
        return True
    except TypeError:
        return False


class Vector:
    """
An ordered and named vector
"""

    def __init__(self, values, *args, keys, name=None):

        self.init = True
        if type(values) in [int, float]:
            values = (values,)
        vec = [*values, *args]
        assert len(vec) == len(keys), ValueError("numbers of values do not match
that of keys")
        self.keys = list(keys)
        self.vector = OrderedDict()
        self.update(vec)
        self.name = name
        self.init = False
        self.tolist = self.to_list

    def __getitem__(self, item):
        if type(item) == tuple:
            """Return a Vector instance if requesting for more than one item"""
            return Vector([self.vector[i] for i in item], keys=item)
        return self.vector[item]
```

```
    def __setitem__(self, key, value):
        self.vector[key] = value
        if key not in self.keys:
            self.keys.append(key)

    def __iter__(self):
        yield from self.vector.values()

    def __len__(self):
        return len(self.vector)

    def __sub__(self, other):
        return self.to_array() - other.to_array()

    def items(self):
        for k, v in zip(self.keys, self.to_list()):
            yield k, v

    def append(self, k, v):
        self.keys.append(k)
        self.vector[k] = v

    def update(self, vec):
        if not self.init:
            assert len(vec) == len(self), ValueError("Length of provided
iterable "
                                                     "must match vector")
        for i, k in enumerate(self.keys):
            self[k] = vec[i]

    def to_list(self):
        return list(self.vector.values())

    def to_array(self):
        return np.array(self.to_list())

    def __call__(self, vector=None):
        """
Update if vector is provided and
return Vector values as an array.
:param vector: a iterable containing values to be updated. Must be in order of
keys and have same length as vector
"""
        if vector is not None:
            self.update(vector)
        return self.to_array()

    def __repr__(self):
        return "Vector<{}>".format(self.to_list())


class GroupVector:
    """
Group multiple vectors together
"""

    def __init__(self, *vectors):
        self.vectors = vectors
        self.keys = []
        for i, v in enumerate(vectors):
            self.keys.extend([f"{i}_{k}" for k in v.keys])

    def __iter__(self):
```

```
        yield from self.keys

    def __getitem__(self, item):
        keys = item.split('_')
        i = keys[0]
        k = "_".join(keys[1:])
        return self.vectors[int(i)][k]

    def items(self):
        for k in self:
            yield k, self[k]


class Sequence:
    """
A class used to store and update multiple items' values in time sequence.
Format
------
{
'item_1': [t1, t2, ..., tn],
'item_2': [t1, t2, ..., tn],
'item_3': [t1, t2, ..., tn],
}
- Now supports dynamic lengths for each items (all items do not have to have
the numbers of time steps)
"""
    def __init__(self, max_len=None):
        self.max_len = max_len  # maximum length of item data
        self.sequence = {}
        self.can_strip_key = False

    def clear(self):
        self.sequence.clear()

    def __call__(self, variables):

        if type(variables) == GroupVector:
            self.can_strip_key = True

        for k, v in variables.items():
            if k not in self.sequence:
                self.sequence[k] = deque(maxlen=self.max_len)
            self.sequence[k].append(v)

    def __getitem__(self, item):
        return self.sequence[item]

    def __len__(self):
        if not self.sequence:
            return 0
        # return length of one of the items
        return len(self.sequence[list(self.keys())[0]])

    def __iter__(self):
        yield from self.sequence

    def data_as_array(self):
        s = Sequence()
        for k in self.sequence:
            s.sequence[k] = np.array(self[k])
        return s

    def dump(self, path, prefix="", append_time=False):
        """Save sequence to file"""
```

```
        ap_time = f"-{int(time.time() * 1e6)}" if append_time else ""
        full_path = os.path.join(path, f"{prefix}{ap_time}.sequence")
        with open(full_path, 'wb') as fp:
            pickle.dump(self, fp)

        print(f"Successfully dumped sequence to {full_path}")

    @classmethod
    def load(cls, path):
        """Load sequence from file"""
        with open(path, 'rb') as fp:
            obj: cls = pickle.load(fp)
            return obj

    def get_batch(self, size, stack_len=1, to_stack=("current_state",
"new_state"),
                  has_terminal=True):
        """



:param has_terminal: indicate whether sequence has an ite named terminal
:param size: size of batch
:param to_stack: keys of data to stack, others will single index
:param stack_len: length of previous sequential data to prepend with indexed
data
:return:
"""
        batch = Sequence(max_len=size)
        batch_index = np.random.choice(range(len(self)), size)

        def update_data(keys=self, stacking=False):
            for k in keys:
                _bool = stacking and has_terminal and self['terminal'][oi-1]
                if i < 0 or _bool:
                    o_data = self[k][oi]
                    data = [0] * len(o_data) if is_iter(o_data) else 0
                else:
                    data = self[k][i]

                if is_iter(data):
                    stack[k].extend(data)
                else:
                    stack[k].append(data)

        for i in batch_index:
            stack = Vector([[] for _ in range(len(self.keys()))],
keys=self.keys())
            oi = i  # original index
            update_data()

            for _ in range(stack_len - 1):
                i -= 1
                update_data(to_stack, True)  # stack previous sequence

            batch(stack)

        return batch.data_as_array(), batch_index

    def keys(self):
        return self.sequence.keys()

    def plot(self, fig_size=(20, 50), path=False, sharex="all", plot_ind=False):
```

```
        def strip_key(key):
            """extract name from key"""
            if self.can_strip_key:
                n = key.split('_')
                return "_".join(n[1:])
            return key

        rows = (len(self.sequence) // 2) + (len(self.sequence) % 2)
        fig, axes = plt.subplots(rows, 2, sharex=sharex, figsize=fig_size)
        axes = axes.ravel()

        if plot_ind:
            plt.figure()
            plt.title("Plot of vehicle's trajectory in X-axis")
            plt.plot(self['0_X'])
            plt.xlabel("time (s)")
            plt.ylabel("Distance (m)")

            plt.figure()
            plt.title("Plot of vehicle's trajectory in Y-axis")
            plt.plot(self['0_Y'])
            plt.xlabel("time (s)")
            plt.ylabel("Distance (m)")

            plt.figure()
            plt.title("Plot of vehicle's yaw")
            plt.plot(self['0_yaw'])
            plt.xlabel("time (s)")
            plt.ylabel("Angle (degrees)")

        for i, k in enumerate(self):
            axes[i].set_title(strip_key(k))
            axes[i].plot(self[k], label=strip_key(k))
            axes[i].grid()
            axes[i].legend()
        if path:
            plt.figure()
            plt.title("Path")
            plt.plot(self['0_X'], self['0_Y'],
label="{}".format(max(self['0_X'])))
            plt.legend()
        plt.show()


class Scale:
    """
A simple (unnecessary) class for scaling model
parameters.
"""

    def __init__(self, scale):
        self.scale = scale

    def __call__(self, value):
        return value / self.scale


class Animation:
    """
Used to illustrate the path and movement of models/agents based on
provided sequence
"""

    def __init__(self, sequence: Sequence, imp="normal"):
```

```
        self.yaw = np.array(sequence['0_yaw']) % (2 * np.pi)
        self.x_array = np.array(sequence['0_X'])
        self.y_array = np.array(sequence['0_Y'])
        self.sequence = sequence

        r = {  # robot patch parameters
            'w': 6,  # width
            'h': 3,  # height
            'hw': 1,  # head width
        }
        scale_factor = max(abs(self.x_array).max(), abs(self.y_array).max())
        scale = 150 / scale_factor  # scale down
        robot_shape = np.array([r['w'], r['h']]) / scale
        self.robot = plt.Rectangle((0, 0), *robot_shape, fc="r")
        head_shape = np.array([r['hw'], r['h']]) / scale
        self.robot_head = plt.Rectangle((0, 0), *head_shape, fc="black")

        self.set_pos((0, 0))

        # plot
        self.fig = plt.figure()
        self.ax = plt.axes()
        plt.title(f"2D motion path of {imp} profile")
        plt.xlabel("X")
        plt.ylabel("Y")
        self.fig.set_dpi(100)
        self.fig.set_size_inches(7, 6.5)
        plt.plot(self.x_array, self.y_array)
        plt.plot([self.x_array[0]], [self.y_array[0]], marker='d', color="g",
label="Start Point")
        plt.plot([self.x_array[-1]], [self.y_array[-1]], marker='d', color="r",
label="End Point")
        self.ax.add_patch(self.robot)
        self.ax.add_patch(self.robot_head)
        plt.axis('scaled')
        plt.legend()

    @property
    def params(self):
        return dict(
            fig=self.fig, func=self.animate,
            frames=len(self.yaw), blit=True,
            interval=10,
        )

    def set_pos(self, pos):
        # set robot position
        w, h = self.robot.get_width(), self.robot.get_height()
        x, y = pos[0] - (w / 2), pos[1] - (h / 2)
        self.robot.set_xy((x, y))

        # set robot head position
        self.robot_head.set_xy((x + w, y))

    def transform(self, t):
        self.robot.set_transform(t)
        self.robot_head.set_transform(t)

    def plot_at(self, i):
        """
:param i: index
:return:
"""
        x = self.x_array[i]
```

```
        y = self.y_array[i]
        t = Affine2D().rotate_around(x, y, self.yaw[i]) + self.ax.transData
        self.transform(t)
        self.set_pos((x, y))

    def show(self):
        # necessary to assign to FuncAnimation a variable for it to be displayed
        ani = FuncAnimation(**self.params)
        plt.show()
        return ani

    def save(self, *args, **kwargs):
        animation = FuncAnimation(**self.params)
        animation.save(*args, **kwargs)

    def animate(self, i):
        self.plot_at(i)
        return self.robot, self.robot_head


class Agent:
    """
A wrapper class to wrap models (simulation and real-world
models).
"""

    def __init__(self, v_model):
        """
:param v_model: vehicle model. could be simulation or real world model.
"""
        self.v_model = v_model

    def __call__(self, *args, **kwargs):
        return self.v_model(*args, **kwargs)

    def __getattr__(self, item):
        """
Fallback to model's attributes if not implemented
here.
"""
        return getattr(self.v_model, item)


def extract_returns(call_func):
    """
A decorator function used to wrap vehicle model __call__ function.
It extracts necessary parameters before and after performing an action
and returns them.
"""
    keys = ['current_state', 'action', 'reward', 'new_state', 'terminal']
    call_return = Vector([0] * len(keys), keys=keys)

    @wraps(call_func)
    def wrapper(self, *args, **kwargs):
        current_state = self.get_state()  # get current state before performing
action
        call_func(self, *args, **kwargs)  # perform action
        self.new_state = new_state = self.get_state()
        action = self.action
        call_return([
            current_state, action,
            self.get_reward(), new_state,
            False,  # terminal value will be updated in main loop
        ])
```

```
        return call_return

    return wrapper


class OrnsteinUhlenbeckActionNoise:
    """
Taken from: https://github.com/pemami4911/deep-rl/blob/master/ddpg/ddpg.py
# Taken from
https://github.com/openai/baselines/blob/master/baselines/ddpg/noise.py, which
is
# based on http://math.stackexchange.com/questions/1287634/implementing-ornstein-
uhlenbeck-in-matlab
"""
    def __init__(self, mu, sigma=0.3, theta=.15, dt=1 -2, x0=None):
        self.theta = theta
        self.mu = mu
        self.sigma = sigma
        self.dt = dt
        self.x0 = x0
        self.reset()

    def __call__(self):
        x = self.x_prev + self.theta * (self.mu - self.x_prev) * self.dt + \
                self.sigma * np.sqrt(self.dt) *
np.random.normal(size=self.mu.shape)
        self.x_prev = x
        return x

    def reset(self):
        self.x_prev = self.x0 if self.x0 is not None else np.zeros_like(self.mu)

    def __repr__(self):
        return 'OrnsteinUhlenbeckActionNoise(mu={}, sigma={})'.format(self.mu,
self.sigma)


class ParamsBase:
    """
A base class for Param classes, can save/load params
to/from json.
"""
    excludes = [  # list of attributes to exclude when saving
        "to_json",
        "from_json"
    ]

    def to_json(self, save_path):
        save_path = os.path.join(save_path, "params.json")

        with open(save_path, 'w') as fp:
            data = {}
            for a in dir(self):
                if not a.startswith("__"):
                    data[a] = getattr(self, a)

            [data.pop(x) for x in self.excludes]

            json.dump(data, fp, indent=4)

    @classmethod
    def from_json(cls, save_path):
        save_path = os.path.join(save_path, "params.json")
        obj = cls()
```

```
        with open(save_path, 'r') as fp:
            data = json.load(fp)
            for k, v in data.items():
                setattr(obj, k, v)
        return obj

def resolve_rw(rw):
    if rw in [np.nan, -np.nan]:
        return -999
    return rw


def drift_cost(desired_state, state, sigma=.01, scale=1):
    """
Computes the cost/reward of current state in respect to
desired state.
Based on: Autonomous Drifting using Simulation-Aided Reinforcement
Learning. (Eq. 7)

:param desired_state: A Vector of desired state
:param state: A Vector of current state
:param sigma: A tolerance hyper-parameter. the bigger it is the bigger the
tolerated error gap between desired and real state.
:param scale: A positive scaling factor
:return: cost
"""
    return (1 - np.exp(-np.sum((desired_state - state) ** 2) / (2 *
(sigma**2)))) * scale


def drift_reward(desired_state, state, sigma=.01, scale=1):
    """
Scaled between (-1 and 0) * scal
"""
    return resolve_rw(-1 * drift_cost(desired_state, state, sigma, scale))


def drift_reward_v2(desired_state, state, **_):
    return resolve_rw(-np.sum((desired_state - state) ** 2))


def get_drift_reward(version="v2"):
    func_map = {
        "v1": drift_reward,
        "v2": drift_reward_v2,
    }
    return func_map[version]
```