Software report

AUTONOMOUS DRIFTING RC CAR WITH REINFORCEMENT LEARNING

May 9, 2018

Supervisor: Dr. D. Schnieders
Sourav Bhattacharjee (3035123796)
Kanak Dipak Kabara (3035164221)
Rachit Jain (3035134721)

Written by Kanak Kabara

Abstract

The advent of self-driving cars has pushed the boundaries on the safety of automobiles, but most modern self-driving car systems ignore the possibility of a car slipping resulting from inclement weather or driver error [1]. Passengers and bystanders would benefit heavily if self-driving cars could handle slipping by learning to drift with the turn rather than against it (by applying the brakes, or turning away, which is the instinctive action), preventing many fatalities [2].

Our project is aimed at studying the drifting of an autonomous remote controlled (RC) car using reinforcement learning (RL) techniques. Specifically, we experimented with a model-free approach with dueling double Deep Q-networks (DQN) and a model-based approach with Probabilistic Inference for Learning COntrol (PILCO) for finding an optimal drift controller. Since robotic systems are prone to wear with use, a simulator is used to model the car dynamics and train a preliminary drift controller which is then transferred to the real car.

Using these techniques, we were successful in obtaining an optimal drift controller on the simulator, which was stable and robust to varying physical conditions. Other than the drift controller, this project makes important contributions in the form of novel approaches like using DQN for obtaining a drift controller and using the policy learned from DQN for PILCO initialization. Additionally, this report presents a metric, D_m, to objectively quantify the quality of a sustained circular drift.

Acknowledgement

We would like to thank our supervisor, Dr. Dirk Schnieders, for guiding us throughout the project. We are also grateful for the help received from a friend, Mr. David Ip, who helped us acquire the hardware needed for this project. Finally, we are thankful for the help we received from Dr. Chris R. Roberts with various hardware issues encountered.

Contents

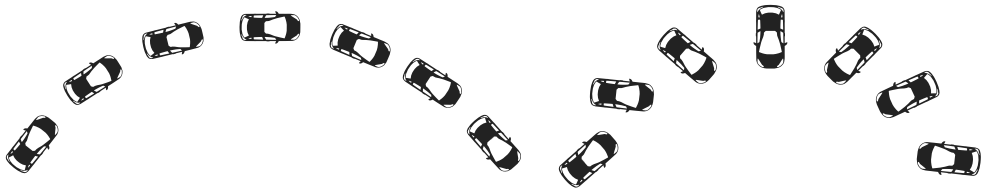
List of Figures

Abbreviations

Abbreviation	**Meaning**
2WD	Two wheel drive
4WD	Four wheel drive
DQN	Deep Q-Networks
EKF	Extended Kalman Filter
HTTP	Hypertext Transfer Protocol
IMU	Inertial Measurement Unit
ODE	Open Dynamics Engine
PID	Proportional–integral–derivative
PILCO	Probabilistic Inference for Learning COntrol
RC	Remote Controlled
RL	Reinforcement Learning
ROS	Robot Operating System
URDF	Unified Robot Description Format
XML	Extensible Markup Language

1 Introduction

Before discussing the implementation details of the project, it is crucial to understand the background of the problem we are trying to solve, and the actual scope. This section addresses that and highlights the need to study and solve the problem of drifting, and outlines how we plan to do so with an approach based on simulation aided reinforcement learning. For the purposes of this report, drifting is defined as the oversteering of a car which results in the loss of traction of the rear wheels. This results in the front wheels pointing in the opposite direction to the turn and the car appears to be moving sideways as shown in Figure 1.



(a) Steering around bend **(b) Drifting around bend**

Figure 1: Drifting is defined as the oversteering of a car which results in the loss of traction of the rear wheels. This results in the front wheels pointing in the opposite direction to the turn and the car appears to be moving sideways. The diagram illustrates the difference between simply turning around a bend and drifting around a bend.

1.1 Background and Motivation

Passenger vehicles usually implement stability control in a number of ways like differential braking [3], active steering [4] [5] or integrated chassis control [6] [7] [8]. Other methods, based on independent wheel torque, have also been developed to make passenger vehicles more stable. However, these methods function by making sure that the tires avoid slipping. In doing so, these methods essentially restrict the operation of the vehicle. Similarly, control algorithms in current self-driving car systems (Anti-lock brake systems, Electronic stability control etc.) try and mitigate the chances of slipping due to its unpredictable nature [1]. Sufficiently lowering the speed of the car and making turns that are not too tight will mostly

prevent slipping, but this does not consider cases where the system must make evasive moves to avoid crashes or when a car is already in a slipping state due to the driver's fault. For example, hydroplaning, which refers to a situation where a layer of water builds up between the car tires and the road, is a major reason for vehicle accidents. According to the United States' Department of Transportation, 15.97% of all vehicle crash fatalities in the United States [2] are attributed to wet and icy roads. An autonomous car system should be prepared for the scenarios outlined above to ensure the safety of the passenger and bystanders, regardless of the weather conditions or the state of the car. To reduce fatalities and ensure that these car systems are as robust and safe as possible, it is essential to study drifting, and eventually deduce how cars can respond quickly to unintentional slipping states as those encountered due to hydroplaning. Not only can drifting be useful to steer out of these unintentional slipping states, but can also be useful in taking full advantage of the capabilities of a vehicle to avoid accidents in emergencies.

Many of the systems discussed above try to tackle the issue of stability control and slipping by approaching it as an optimal control and open looped problem with explicit dynamics model. Approaches using optimal control are often deterministic and use closed-form expressible equations of motions. The resulting policies depends entirely on the model used to compute them. Sometimes, these restrictions on the model neglect parts of the true system either because they are non-linear or they are just not well-enough understood to be expressed in equations. We thus propose a method that does not rely on explicit equations of motion, but rather on an implicit understanding of the world obtained by trial and error.

Another motivation to study drifting is that Paul Frère [9] points out the usefulness of drifting to turn fast around sharp bends. Since high-speed stability is of greater importance to ordinary touring vehicles and competition cars, they tend to understeer, and for a competition car average circuit time is improved by going fast through fast bends while slowing down through sharp ones [9]. However, a car is able to turn sharp bends faster by drifting because the yaw angle formed by the drift brings the vehicle in line with the straight path following the bend even before the vehicle completes the turn [9].

1.2 Objective

The objective of this project is to get a remote controlled car to maintain a sustained circular drift autonomously. This paper proposes a framework for learning the best way to drift using simulation aided reinforcement learning which is one approach to solving the problem without having to input the dynamics of the system explicitly. Then the project aims to transfer the learned optimal drift policy or strategy from the simulation to a physical RC car for further learning.

1.3 Scope

The area of drifting falls into two categories – sustained drift and transient drift. Due to the wide breadth of the two categories and the time and cost constraints, our project will mainly focus on sustained drift, and more specifically steady state circular drift on an RC car with constant forward throttle. Additionally, despite the wide range of reinforcement learning algorithms available, due to reasons elaborated in the *Reinforcement Learning Report*, we investigate two different algorithms to obtain the sustained circular drift controller - DQN and PILCO.

1.4 Deliverables

The complete implementation of the project is available on `https://github.com/kanakkabara/Autonomous-Drifting`. There are a few major deliverables in this project, which are outlined below:

1. Reinforcement Learning (RL) algorithms – Implementation of Double dueling Deep Q-networks for finding an optimal drift controller as well as model based policy search with PILCO.

2. Drift controller - A successful sustained circular drift controller along with tests to prove its robustness and stability.

3. Drift metric - A drift metric to objectively quantify the quality of a drift.

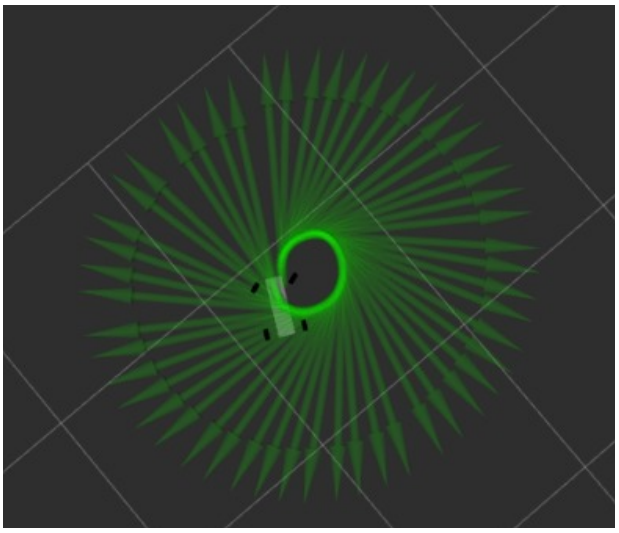


Figure 2: Time-lapsed path traced by the car on the simulator using the successful sustained circular drift controller.

4. Simulator – We trained the RL algorithms on a simulated car that models the RC car in an environment with physics that mimic the real world. The environment was used to test and improve different RL algorithms efficiently and quickly, without causing any wear or damage to the RC car.

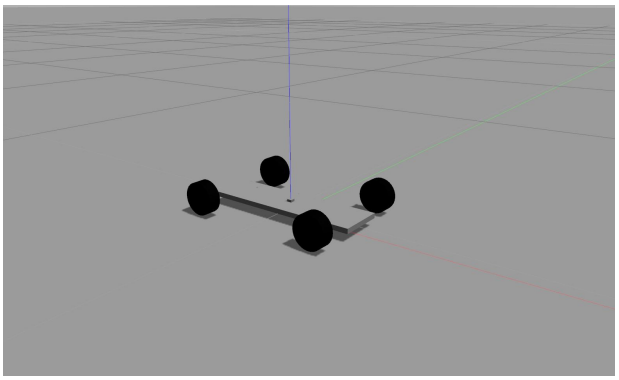


Figure 3: The final simulated car

5. A remote controlled (RC) car – This car is a 1/10th scale model of an actual car, integrated with sensors (Inertial measurement unit combined with magnetometer and optical sensors) for measuring data like the translational and angular velocities of the car to perform steady state circular drifting. The

project then aims to transfer the optimal policy learned in simulator onto the RC car for validation.

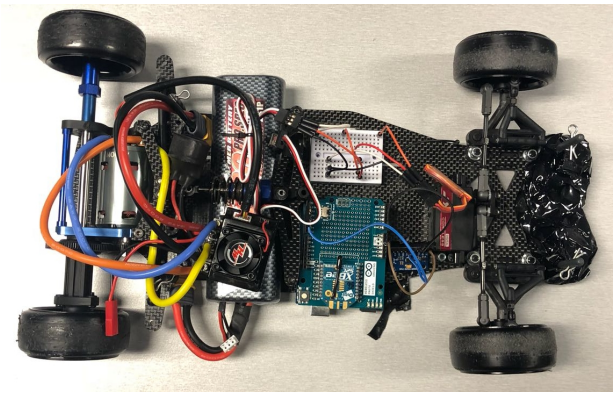


Figure 4: The final RC car

1.5 Contributions

The project introduces the following novel ideas, as elaborated further in the *Reinforcement Learning Report*:

1. Using double dueling Deep Q-networks (DQN) to find an optimal drift controller.
2. Using policy learned from the DQN model to initialize PILCO learning.
3. A drift metric, D_m, to objectively evaluate a sustained circular drift:

$$D_m = \frac{1}{T}\sum_{t=0}^{T} \exp\left(-\frac{(||\boldsymbol{s}_t - \boldsymbol{s}_{target}||^2}{2\sigma^2}\right) \in [0,1]$$

1.6 Outline of Reports

The documentation for this project is divided into three reports. Although the reports share the same background and motivation behind the project, each emphasizes on the methodology, experiments, results and difficulties encountered for different aspects. A reader is thus suggested to refer to all three individual reports to acquire a complete understanding of the project. The three reports are as follows:

Report outlining the Hardware, written by Rachit Jain, highlights the implementation of the RC car and the various challenges faced in indoor localization and velocity estimation.

Report outlining the Reinforcement Learning Algorithms, written by Sourav Bhattacharjee, focuses on the main aspects of the project and contains the detailed description of the two Reinforcement Learning techniques used, and the associated results.

This report outlines the crucial software elements in this project and proceeds as follows. First, the report talks about OpenAI Gym and the motivation behind using it in this project. Then, the implementation details for the simulation environment and the communication stack are discussed. Finally, the report is ended with some concluding remarks.

2 Methodology

This project consists of 3 major components - the hardware, the Reinforcement Learning (RL) agents and the various software elements that bind the entire system together. The subsequent sections talk about the binding software elements, which include OpenAI Gym for Reinforcement Learning, Robotics Operating System (ROS) for communication and Gazebo for simulation.

2.1 OpenAI Gym

The components of the RL algorithm described in Section 3 of the *Reinforcement Learning Report* can be summarized into the [agent, state, action, environment] representation:

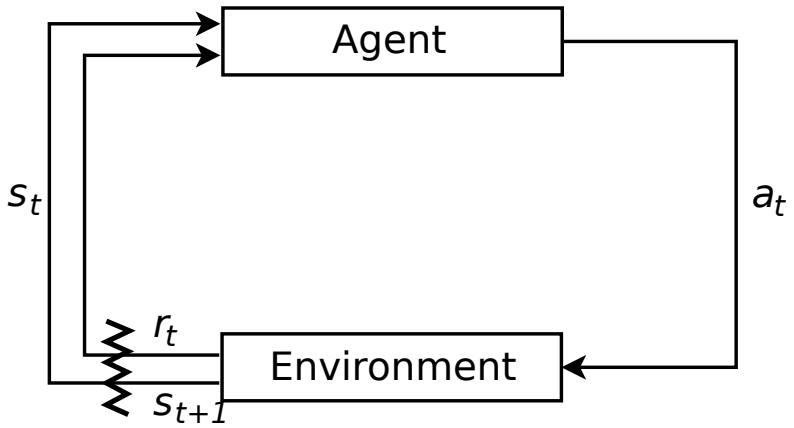


Figure 5: The diagram shows the architecture of the reinforcement learning framework. The agent interacts with the environment according to a policy by taking actions and evaluates how good or bad taking a particular action (a_t) in a particular state (s_t) is by observing the next state it transitions to (s_{t+1}) and the reward it receives along the way (r_t). [10]

The two most crucial components, the agent and the environment, were simple to implement without the use of any libraries, but the ability to test various agents without making changes to the environment, or vice versa, was crucial in order to have a clean and structured system. This was especially important given the number of experiments performed during the course of our project. The

ideal solution had to be a plug-and-play system where we could easily switch out different agents or environments without having to modify other aspects of the system.

OpenAI Gym is an open-source library that is built on the tenet of providing all the tools necessary to develop and compare RL algorithms. OpenAI Gym allowed us to define an environment, which the agent acts upon. The environment then returns the observation and reward. Essentially, this allowed us to use the principle of Separation of Concerns (SoC). The environment is just a 'black box' to the RL agent, and hence we could switch out agents easily without having to alter the environment. It is the responsibility of the Gym environment to calculate the state of the car and give an appropriate reward. It is also the responsibility of the Gym environment to accept any action that is supplied to it by the RL agent, and perform it on the car (real or simulated).

This separation of concern allowed us to experiment with the DQN and PILCO agents that use completely different learning styles on the same environment to deduce the most efficient strategy for learning an optimal drift controller, which is discussed further in Section 3 of the *Reinforcement Learning Report*.

Additionally, as a future extension, the OpenAI Gym drift environment can be made open source to the OpenAI community so that various other algorithms can be tested on the same environment to find more effective and efficient solutions.

2.2 Robot Operating System

The various components described in this project require some channel of communication between them to exchange information - state information from the car or the simulator needs to be communicated to the RL agent to compute rewards and costs, and deduce the next action. Actions need to be communicated to the car or Gazebo to control the car. We could have used simple forms of communication like sockets, HTTP requests/responses, but this would have coupled the two components too tightly, making it difficult to replace the simulator with the physical RC car, or switch between different agents and environments. Having tightly coupled components would have also made it difficult for different teammates to work on different aspects of the project. Hence, it was crucial to have a communication protocol that was as decoupled as possible.

Robot Operating System (ROS) is one such framework that allowed us to set up a decoupled, asynchronous communication system. At its core, ROS provides a messaging interface for different processes (referred to as ROS nodes) to communicate with each other, among other features. ROS provides a queuing mechanism for a ROS node to send a message to another ROS node by posting a message in a ROS queue that can be read by another ROS node, completely asynchronously. ROS nodes do not need to worry about other nodes and how they are implemented, and instead only need to know the address of the ROS queue. Additionally, ROS is language agnostic, which was important since the PILCO agent was implemented in MATLAB while the simulator environment was implemented in Python. By doing so, we created a completely decoupled communication network, which allowed us to work on modules independently, and interchangeably.

There are various open source ROS packages that were used in this project. The following sections highlight the packages incorporated.

2.2.1 Inertial Measurement Unit (IMU) data fusion

As discussed in Section 3.2.2 of the *Hardware Report*, after considering various methods for getting accurate estimates of the velocity of the car from the car's ref-

erence frame, we came to the decision of performing integration on the acceleration data from multiple IMUs. In order to do so, IMU data from the MPU9250 sensor and an Android device placed on the car were combined together to obtain the acceleration of the car. The noise in the IMU data is assumed to have a Gaussian distribution. Hence, using multiple sources of IMU data allowed us to eliminate the noise as much as possible and get a cleaner reading of acceleration.

The biggest issue with using two completely stand alone systems lies in synchronizing the system clocks to get time sensitive information - the IMU reading from the two sensors need to match up exactly so that they can be combined to obtain a better estimate for the velocity components.

In order to do so, the *message_filters* package was used. A time synchronization message filter was set up to collect messages from the two sources and output them only when the time stamps on the messages matched. A margin of 0.05 seconds was allowed to ensure that important IMU events were not lost in case the time stamps did not match exactly. Once the IMU messages were synchronized, they were passed into a *robot_localization* node, which made use of an Extended Kalman Filter (EKF) to compute the corresponding odometry data for the car. This included the velocity, position and orientation of the car. Figure 6 below shows the system architecture for computing velocity data.

Both the *message_filter* and EKF *robot_localization* nodes are implemented flexibly to allow us to add more sources of IMU information quickly if more accurate data is needed.

2.2.2 ROS on Android

In order to collect IMU data from an Android device, we leveraged the Android library of the ROS framework. Using the Android library, we created an Android application that was used to initiate a ROS node on the Android device and communicate IMU data wirelessly to the master node on our laptops. The added advantage of using an Android device for collecting IMU data is that numerous Android devices can be added easily in case the data is still noisy.

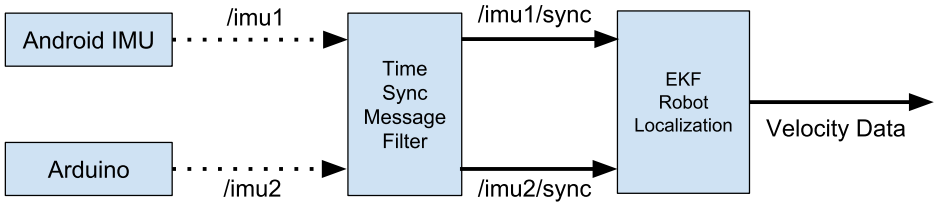


Figure 6: The diagram shows the architecture for using an Extended Kalman Filter (EKF) for estimating velocity from IMU data. The IMU data from different sources is synchronized using a *message_filter* before being passed to the *robot_localization* node, which performs the data fusion.

2.2.3 Demonstration

As outlined in Section 3.2.1 of the *Reinforcement Learning Report*, we explored three methods of initialization for our PILCO algorithm: random initialization, initialization with the DQN model and demonstration. Performing random initialization was straightforward - we simply had to take a series of random actions either on the physical RC car or the simulator and get the resultant state from each action. A similar method was used for the DQN model, where the model outputs the actions to be taken. However, in order to collect data from the demonstration, we needed teleoperation support for the car.

We developed two systems for performing teleoperation on the car:

- Keyboard: The arrow keys were mapped to send throttle and servo commands to the car. The *teleop_twist_keyboard* ROS package was used to capture keystrokes and convert them to the appropriate linear and angular velocities.

- Joystick: One popular method for drifting is to make use of opposite locking or counter steering (turning the steering in the opposite direction of the intended direction of motion). In order to do so, one needs the ability to

quickly change steering directions, which is not possible with a keyboard teleoperation node. Hence, a joystick controller was used. The *joy* ROS package was used to set up the joystick and publish joystick commands, which were processed to produce the appropriate actions.

2.3 Gazebo Simulator

Even with the Open AI Gym environment optimally set up, it usually takes RL agents thousands of iterations before they are able to converge to an optimal strategy for reward maximization. Running many iterations on the RC car is not feasible for a few reasons:

1. The hardware will get worn out quickly.
2. The battery cannot last for thousands of iterations.
3. The car may get damaged, since the agent will experiment with sub-optimal moves in early stages of the algorithm.

Hence, to circumvent these problems, we needed to train our algorithm in a simulated environment first.

Gazebo is an open-source simulation library which allows us to design and model the RC car and quickly test the RL agents in a realistic world. It uses the Open Dynamics Engine (ODE) as the physics engine, with properties like friction, gravity etc., in a simulated world. It also allows us to easily create a robot that will mimic the behavior of the real RC car, which is crucial, since we want the controller trained on the simulation to work equally well in the real world.

2.3.1 Modelling the RC car

Gazebo makes use of the Unified Robot Description Format (URDF), which is an XML format for defining accurate robotic models. Essentially, models are defined using links and joints, where a joint is used to connect two links. We started at the root link, known as the base link and built on top of that.

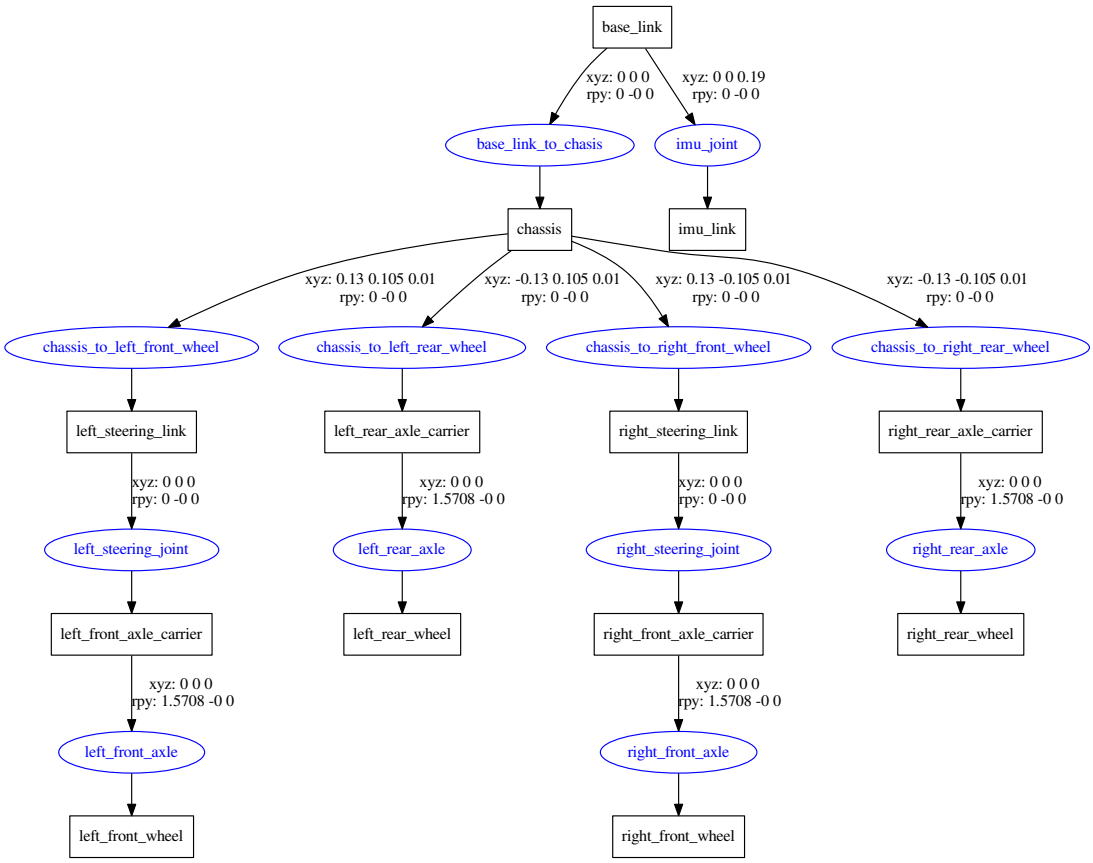


Figure 7: This figure shows the relationship between the joints and links of the simulated RC car. The rectangles represent the links, whereas the ovals represent joints. They combine together to form the simulated body of the car.

As seen in Figure 7, our robot consists of the following links:

1. Base Link: The root link on top of which the rest of the robot is defined.

2. Chassis: The base frame of the car, which houses the electronic components as defined in Section 3 of the *Hardware Report*. Practically speaking, the various components on top of the chassis will not affect physical properties of the car relevant to drifting other than the mass. The mass distribution on the actual car is evenly spread out, and hence the mass of all components on

top of the chassis were added directly into the mass of the chassis to reduce complexity.

3. IMU link - Physical mount for the IMU.

4. Left/Right Front/Rear Axle Carrier: All wheels have an axle joint, actuating on which causes the wheels to roll. The physical link connected to the axle is defined as the axle carrier.

5. Left/Right Steering Link: The front wheels of the RC car can be actuated on by the servo, causing the front wheels to turn. Since the servo is not directly defined, the steering link connects the chassis to a steering joint.

6. Left/Right Front/Rear Wheel: The wheels of the RC car.

To connect the various links described above, our robot consists of the following joints:

1. Base link to chassis: Simple fixed joint connecting the chassis to the root link.

2. IMU joint: Simple fixed joint connecting the physical IMU mount to the link.

3. Chassis to Left/Right Front/Rear wheel: Simple fixed joint connecting the chassis to the entire wheel housing (i.e. the axle joints, steering joints, and the wheels).

4. Left/Right Steering Joint: Revolute joint (used for single axis fixed rotations, such as wheels turning) connecting the steering link to the axle. This is the joint that is actuated on to turn the wheels, causing the direction of motion of the car to change.

5. Left/Right Front/Rear Axle: Continuous joint (used for single axis continuous rotations, such as wheels rotating) connecting the axle carrier to the physical wheel. This is the joint that is actuated on to rotate the wheels, causing the car to move forward. To simulate a four wheel drive (4WD), all

the axles are actuated on, whereas only the rear axles are actuated on for a two wheel drive (2WD).

2.3.2 Proportional–integral–derivative (PID) Controllers

In order to actuate on the joints defined above, we attached controllers, which are Gazebo's equivalent of servos and motors. The *ros_control* package was used to create these controllers. The package makes use of a typical closed (feedback) loop control system to accurately control the output signals to an actuator.

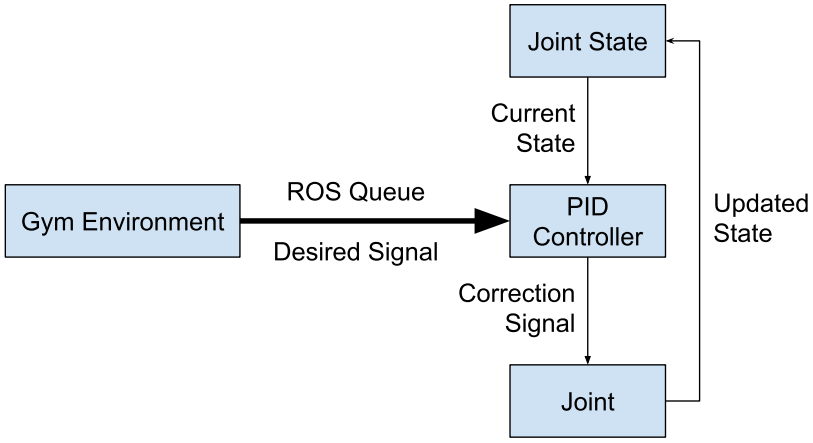


Figure 8: Each controller exposes a ROS queue where the desired joint velocity/effort can be published by the Gym environment. Upon receiving the desired signal, the PID controller uses the current state of the joint to calculate the correction signal needed to achieve the desired signal.

Figure 8 visualizes the controller system, which takes the joint state, acquired from encoders on the actuator, as input. Each controller exposes a ROS queue where the desired joint velocity or effort can be published. The PID controller uses the current state of the joint to calculate and send a correction signal to the actuators in order to control the output.

The simulated RC car has six such controllers - one on each of the four axle joints, and the two steering joints on the front wheels. Only the rear axle joints

are used on a 2WD version of the car, whereas all four axles are used on the 4WD car. Once the Gym environment receives an action to be executed, it calculates the appropriate desired signal to be sent to each PID controller.

2.3.3 Body Frame Velocities

As outlined in Section 3.1 of the *Reinforcement Learning Report*, we initially had a full Markovian state of $[x,\ y,\ \theta,\ \dot{x},\ \dot{y},\ \dot{\theta}]$, where each value is with reference to the world frame. It was easy to acquire the state directly from Gazebo, since it maintained a comprehensive state of all models in the simulation environment. However, once we arrived on the final state representation of $[\dot{x}_{car},\ \dot{y}_{car},\ \dot{\theta}]$, we required velocities from the car's reference frame i.e. body frame velocities. We considered two possible methods for solving this problem:

1. Integrating IMU data: Using the IMU on the simulated car, we could have used the acceleration data and integrated it to get the instantaneous body frame velocities. However, the biggest problem with this approach is the drift in velocity data caused by noise accumulation over time.

2. Vector Transformation: Since velocities are vector quantities, we could simply use a vector transformation from the world frame to the car's reference frame. Gazebo provides all the data required to generate a transform, and hence it is comparatively easier to use this method. This method would give us precise and accurate velocity data, resulting in an optimal drift controller.

In order to simulate the collection of body frame velocities from the real world, using the IMU would be the most logical option. However, the controller learned from noisy data would be inaccurate, and would simply fail to sustain a circular drift. Thus, we decided to go with the less realistic, but more accurate approach of vector transformation.

There are two ROS packages that helped us store and maintain the appropriate transformations needed to track the body frame with reference to the world frame:

- *tf*: Keeps track of multiple coordinate frames by maintaining the relationship between frames in a tree structure in real time. Using the tree structure, vectors and points can be transformed between any two coordinate frames.
- *robot_state_publisher*: Takes the joint states data from the controllers defined in the section above and publishes the positions of the joints to *tf*. It uses the URDF of the robot to get the exact kinematic model of the robot to publish precise information.

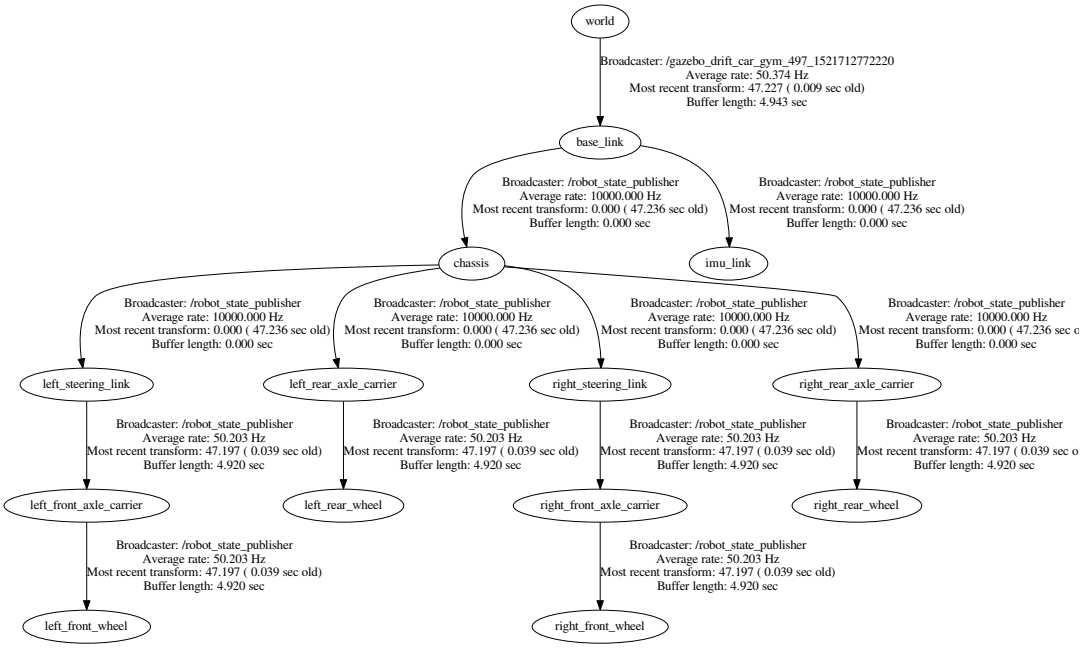


Figure 9: The tf tree structure maintains the relationships between any two links of the RC car. Using this tree, once we have the velocity of the car in the world frame, we can easily transform that vector into the car reference frame.

Figure 9 shows the *tf* tree structure. There are two things to note here. The transforms between the world frame and the *base_link* frame are published manually after the relevant model state is collected from Gazebo. The remaining transforms between the *base_link* and the other links defined above are handled by

the *robot_state_publisher*. Thus, using this tree, we can get the velocity of the car in the *base_link* coordinate frame world frame using the world frame velocities.

2.4 System Architecture

Combining the various concepts introduced above, we can get the entire system architecture needed to replicate this project. The ROS network that was used to connect the various components is shown in Figures 10 and 11. Blue rectangles represent processes running as ROS nodes, while the thicker solid arrows represent the ROS queues (the text beside each arrow denotes the queue name). Additionally, dashed arrows represent library function calls, while the red dotted lines represent wireless data transmission.

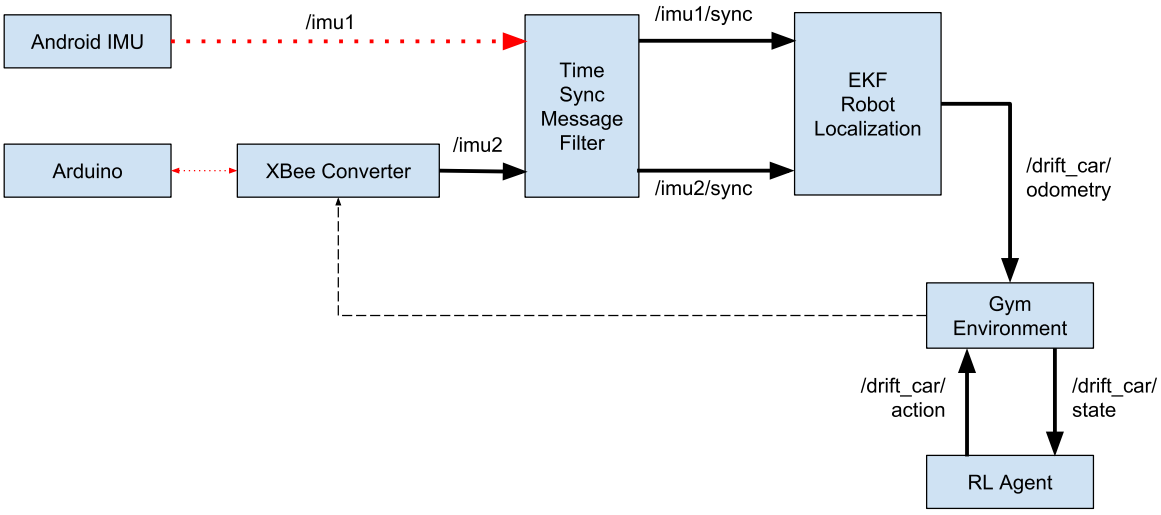


Figure 10: ROS network structure for the RC car which shows the RL agent interacting with the Gym environment using the abstract ROS queues for state and action. It also shows how the Gym environment acquires the state of the RC car using the EKF localization node. Finally, the diagram also shows how the actions from the RL agent are forwarded to the Arduino for teleoperation.

Figure 10 denotes the ROS network structure for the RC car. It shows the RL agent interacting with the Gym environment using the abstract ROS queues for state and action. It also shows how the Gym environment acquires the state of the RC car using the EKF localization node. Finally, we can also see how the actions

from the RL agent are forwarded to the Arduino for teleoperation.

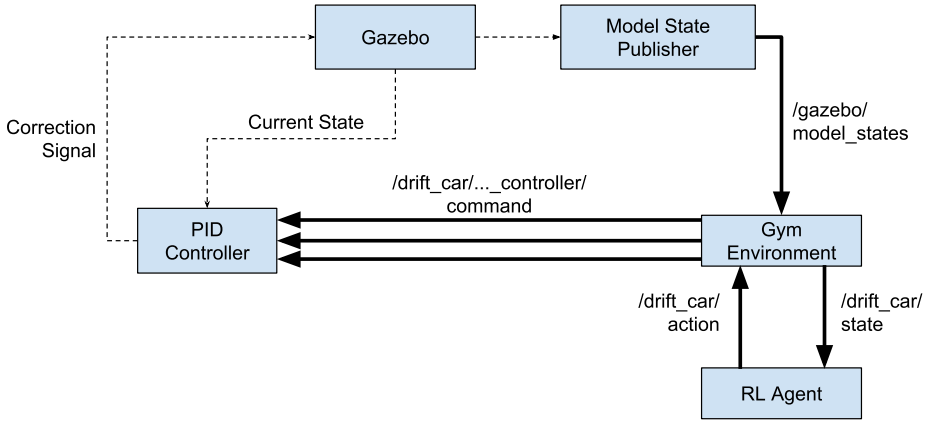


Figure 11: ROS network structure for the Simulated RC car. We can see how the Gym environment acquires the state of the RC car using the Gazebo model state publisher. Additionally, the diagram illustrates how the actions from the RL agent are forwarded to the PID controllers, which communicate with Gazebo to perform the desired actions on the car.

Figure 11 denotes the ROS network structure for the simulated RC car on Gazebo. The abstract ROS queues for state and action are the same, hence proving that the RL agent does not change behavior for the actual and the simulated RC car. We can also see how the Gym environment acquires the state of the RC car using the Gazebo model state publisher. Lastly, we can observe how the actions from the RL agent are forwarded to the PID controllers, which communicate with Gazebo to perform the desired actions on the car.

3 Conclusion

To summarize, we justified why autonomous drifting cars are important and how drifting can be useful in emergencies to avoid accidents. As we have already discussed, current self driving cars and stability control techniques try to avoid slipping tires and in doing so, restrict the capability of the car. However, we need to exploit the full capability of a car during emergencies. So clearly, having an autonomous drifting car that learns an optimal drift control policy using our methods can help reduce the number of accidents caused by hydroplaning and make roads safer.

Motivated with this intention, the report firstly outlines the need for separation of concerns in this project, and how that was facilitated by OpenAI Gym. Next, it introduces ROS, and how it was leveraged as the backbone of the communication stack. We also discussed how ROS was used to acquire IMU data from an Android device, fuse that data to get a velocity estimate, and capture demonstration data from the RC car (physical and simulated). Next, we examined the need for a simulated environment in this project, and how Gazebo was used for the same. We gave a detailed description of how the robot model was created using URDFs, and how we added actuators on the car in the form of PID controllers. Finally, an overview of the entire system is presented to combine the information of the various components introduced earlier in the report.

Although the initial aim of the project was to implement autonomous sustained circular drift in a physical RC car, we did not manage to achieve it completely, owing mostly to hardware challenges associated with indoor localization as discussed in Section 3.2.1 of *Hardware Report* report and cost constraints in acquiring a 4WD RC car. Nevertheless, much effort was put into closely modelling the physical properties of an RC car in the simulator as discussed in this report. Thus, given our success in finding a robust and stable sustained circular drift controller with the simulator, we firmly believe the results can be easily replicated on a physical RC car once the hardware is acquired.

References

[1] F. Zhang, J. Gonzales, K. Li, and F. Borrelli, "Autonomous drift cornering with mixed open-loop and closed-loop control," in *Proceedings IFAC World Congress*, 2017.

[2] S. Saha, P. Schramm, A. Nolan, and J. Hess, "Adverse weather conditions and fatal motor vehicle crashes in the united states, 1994-2012," *Environmental Health*, vol. 15, 2016.

[3] A. T. van Zanten, R. Erhardt, G. Landesfeind, and K. Pfaff, "Vehicle stabilization by the vehicle dynamics control system esp," *IFAC Mechatronic Systems, Darmstadt, Germany*, pp. 95–102, 2000.

[4] J. Ackermann, "Robust control prevents car skidding," *IEEE Control Systems Magazine*, vol. 17, pp. 23–31, 1997.

[5] K. Yoshimoto, H. Tanaka, and S. Kawakami, "Proposal of driver assistance system for recovering vehicle stability from unstable statesby automatic steering," in *Proceedings of the IEEE InternationalVehicle Electronics Conference*, 1999.

[6] A. Hac and M. Bodie, "Improvements in vehicle handling through integrated control of chassis systems," *International Journal of Vehicle Design*, vol. 29, no. 1, 2002.

[7] J. Wei, Y. Zhuoping, and Z. Lijun, "Integrated chassis control system for improving vehicle stability," in *Proceedings of the IEEE International Conference on Vehicular Electronics and Safety,*, 2006.

[8] A. Trachtler, "Integrated vehicle dynamics control using active brake steering and suspension systems," *International Journal of Vehicle Design*, vol. 36, no. 1, pp. 1–12, 2004.

[9] P. Frère, *Sports Car and Competition Driving*. Bentley, 1969.

[10] R. S. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.