COMPARISON OF MODERN CONTROLS AND REINFORCEMENT LEARNING FOR ROBUST CONTROL OF AUTONOMOUSLY BACKING UP TRACTOR-TRAILERS TO LOADING DOCKS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Mechanical Engineering

by

Journey McDowell

November 2019

COMMITTEE MEMBERSHIP

TITLE:	Comparison of Modern Controls and Reinforcement Learning for Robust Control of Autonomously Backing Up Tractor-Trailers to Loading Docks
AUTHOR:	Journey McDowell
DATE SUBMITTED:	November 2019
COMMITTEE CHAIR:	Charles Birdsong, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	John Fabijanic, M.S. Professor of Mechanical Engineering
COMMITTEE MEMBER:	Eric Mehiel, Ph.D Professor of Aerospace Engineering
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

Comparison of Modern Controls and Reinforcement Learning for Robust Control of Autonomously Backing Up Tractor-Trailers to Loading Docks

Journey McDowell

Two controller performances are assessed for generalization in the path following task of autonomously backing up a tractor-trailer. Starting from random locations and orientations, paths are generated to loading docks with arbitrary pose using Dubins Curves. The combination vehicles can be varied in wheelbase, hitch length, weight distributions, and tire cornering stiffness. The closed form calculation of the gains for the Linear Quadratic Regulator (LQR) rely heavily on having an accurate model of the plant. However, real-world applications cannot expect to have an updated model for each new trailer. Finding alternative robust controllers when the trailer model is changed was the motivation of this research.

Reinforcement learning, with neural networks as their function approximators, can allow for generalized control from its learned experience that is characterized by a scalar reward value. The Linear Quadratic Regulator and the Deep Deterministic Policy Gradient (DDPG) are compared for robust control when the trailer is changed. This investigation quantifies the capabilities and limitations of both controllers in simulation using a kinematic model. The controllers are evaluated for generalization by altering the kinematic model trailer wheelbase, hitch length, and velocity from the nominal case.

In order to close the gap from simulation and reality, the control methods are also assessed with sensor noise and various controller frequencies. The root mean squared and maximum errors from the path are used as metrics, including the number of times the controllers cause the vehicle to jackknife or reach the goal. Considering the runs where the LQR did not cause the trailer to jackknife, the LQR tended to have slightly better precision. DDPG, however, controlled the trailer successfully on the paths where the LQR jackknifed. Reinforcement learning was found to sacrifice a short term reward, such as precision, to maximize the future expected reward like reaching the loading dock. The reinforcement learning agent learned a policy that imposed nonlinear constraints such that it never jackknifed, even when it wasn't the trailer it trained on.

ACKNOWLEDGMENTS

There were seven people who were instrumental in making this thesis possible. I was studying abroad in Germany where I participated in the first Formula Student Driverless competition in 2017 (municHMotorsport). Students took their previous year Formula SAE (Student) cars and implemented autonomous drive kits for perception, path planning, and control. I was influenced by this competition because I saw the power of Convolutional Neural Networks for computer vision of detecting cones. This is how I caught the bug for machine learning.

One day at a coffee shop in Munich, I bumped into Dr. Charles Birdsong, who is the Committee Chair for this thesis. He was abroad because he was teaching a short Summer course on autonomous vehicles. I described to him what I was involved with during my time in Germany and also asked to keep in touch as a potential advisor for me. I told him I was interested in doing a project comparing conventional controls and machine learning. He wasn't too sure if he was interested because I did not have a clear project. I have to give credit to Dr. Charles Birdsong for putting me in touch with David Smith.

David Smith, a Cal Poly alumni and an engineering manager for the autonomous team at Daimler Trucks North America, provided the use case of backing up a tractor-trailer. Dr. Birdsong, David, and I had a meeting where we discussed a potential project. David described industry problems such a lack of sensors on trailers and the sheer number of different trailer types. Amongst my initial research, I stumbled across a few papers where machine learning was used with a tractor-trailer. I discussed some of these papers with David and he actually asked if he could take them as reading material on the plane ride home. He was interested.

Thank you Dr. Birdsong for your ability to scope a project and for the help with discovering my research project tied to an industry problem. Your guidance has served me well along the way. John Fabijanic was our faculty advisor for Formula SAE at Cal Poly. This was a club that really taught me to be a better engineer. It was an outlet for students to bring the things they learn in academia into practice and fruition. Your vehicle dynamics knowledge was extremely helpful. Dr. Mehiel taught me modern controls. I hope I can teach you a few things about optimal control from the reinforcement learning perspective. Dr. Kurfess helped me sift through the behemoth topic of artificial intelligence, from a computer science perspective. All of these people make me feel proud for choosing to study Mechanical Engineering with a Mechatronics concentration.

You never know who you might meet when reaching out to the research community. Chris Gatti, also known as the "Academic Acrobat," researched using Reinforcement Learning for backing up

a tractor-trailer with discrete actions. I stumbled upon a very interesting Ted Talk he gave about his career path. Two days after defending his PhD, he ended up joining the circus. He found his passion in teaching people how to perform a handstand. His website actually offered a service where he could instruct a group of people to just that. I reached out because I always wanted to learn how to do it, but also because I had a few questions regarding his work. We actually met up in San Luis Obispo because the timing worked out and I can honestly say we have become friends.

The last person who really deserves recognition is my father, Michael McDowell. You are my biggest supporter. You gave me the same thirst for knowledge that you have. Thank you for being a continual soundboard during my journey of this research project. You told me of the story that got you fascinated with artificial intelligence as a kid. You couldn't quite remember the name, but once described it to me as something you tried to recreate. It used beads in matchboxes for learning to play tic-tac-toe; the number of beads really made you understand the weights that are learned for decision making. My research actually stumbled upon it as Machine Educable Noughts and Crosses (MENACE) by Donald Michie [1]. Even though you claim you only have an Electrical Engineering background, the conversations and cross-pollination has helped me make this thesis what it is today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Goods can get transported hundreds of miles in trailers only to be damaged in the last few feet. Merchandise in trailers get towed across the country by a heavy duty vehicle called a tractor. To get the goods into facilities or warehouses, the tractors need to back up to loading docks so the items can get unloaded. The task is non-trivial because the trailer moves in the opposite direction of the steered vehicle as shown in Figure 1.1. Continuing to steer in the wrong direction, even at slow speeds, could cause a phenomenon called jackknifing. This is where the trailer angle becomes large enough that the combination vehicle gets into an un-driveable state. The driver must drive forward again to re-position the vehicle to a better starting point, only to reverse again. It takes skill to back up a trailer without crashing into the loading dock and there is a shortage of truck drivers. Autonomous vehicles can help fill this gap, however, current control methods for path following struggle performing on a different trailer than it was designed for. Machine learning may be able to help generalize control policies for placement of various trailer types.

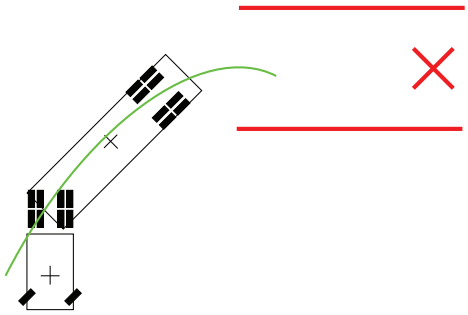


Figure 1.1: When traveling in reverse, the trailer moves in the opposite direction of steered vehicle.

The fundamental problem is the tractor manufacturers do not own the trailers in circulation throughout America. A variety of different trailers exist such as flatbeds, tankers, reefers, box trailers, and container trailers. The trailers are not equipped with a back-up camera or other sensors to assist the driver. Sometimes the trailers are 20 years old and the industry must work with the existing infrastructure.

Industry has developed Advanced Driver-Assistance Systems (ADAS) on the tractors to prevent common accidents, however, often times they do not even know a trailer is attached. The SAE J1939 [2] is a protocol for communication for various heavy-duty vehicles. The ADAS systems do not necessarily exist on the same CAN bus communication line to prevent bus overloads. According to

SAE J1939, information like trailer Anti-lock Braking System (ABS) status and amount of physical load applied to the tractor-trailer attachment by the trailer may be transmitted to the tractor. This is different than the ISO 11992-2 [3] in Europe, where basic information including number of axles, wheelbase, and other trailer parameters are required to be sent. In other words, SAE J1939 is not strict enough. This makes it more difficult to know what trailer is currently being used for an autonomous drive kit.

There are simply too many different brands and styles of trailers currently circulating in America with a lack of sensor information. One solution could be to create regulations and laws that mandate the installation of sensors on trailers. Currently ADAS sensors on the tractors include RADARS and cameras. Unfortunately, setting up sensors for new trailers would require installation and calibration; this requires time out of service that corporate may not be able to justify financially.

When automating a trailer back-up using modern controls, usually a system model of the combination vehicle parameters are needed to design the controller. In addition to there being different trailers, the same trailers can actually modify the wheelbase by locking the brakes of the trailer and moving forward or backward with the tractor. There is a need for managing the complexity in the abundance of trailer parameters; there is a desire to generalize the trailer parameters for control systems. In other words, autonomous and ADAS systems need to be able to handle trailers with different parameters as suggested in Figure 1.2 with the different lengths.

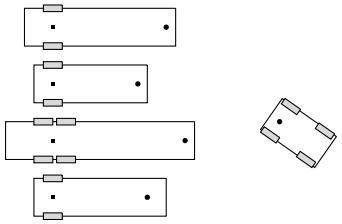


Figure 1.2: It is often difficult to obtain updated models for the various trailers.

Reversing a tractor-trailer to a loading dock has become a good measure of different control methods according to the following literature. Classical control methods have been found to use Proportional-Integral-Derivative (PID) [4]. Modern control methods include Linear Quadratic Regulators (LQR) [5, 6, 7, 8], Model Predictive Control (MPC) [9], Lyapunov Controllers [10, 11, 12], and Fuzzy Logic [13, 14, 15, 16]. Many modern control methods seek to solve the optimal control problem where gains are designed to minimize a cost function.

Most of the preceding applications designed the problem such that the trailer must minimize the error from a specified path, i.e. how one gets to the goal matters. The machine learning

control methods in literature appear to have created the truck backer-upper problem where machine learning finds the best way to get to the goal without jackknifing. Machine learning control methods include neuro-controllers [17, 18, 8, 19, 20] trained with labeled examples of trajectories. The neuro-controllers are created using supervised learning, where samples are labeled by a human so a machine can predict the correct control action. Machine learning control methods also include reinforcement learning where an agent learns through interaction with an environment [21, 22, 23, 24]. The agent is the learner and the decision maker who improves by trial-and-error to maximize a scalar reward.

Given the discovery in the last decade that Graphics Processing Units (GPUs) were able to accelerate learning through using parallel computations, there have been a substantial amount of recent activity in machine learning. Artificial Neural Networks (ANNs or NNs) can be thought of as universal non-linear function approximators. This means that it creates a mapping of input to output just as one could think about conventional controls. It would literally foster an approximation of the state space for the best action to take. The neural networks consist of weights and biases that can be thought similarly to neurons firing in one's brain when a certain smell triggers a memory. The question is whether or not this approximation can generalize across varying trailer parameters in this tractor-trailer back-up problem.

A policy is the controller, i.e. a learned mapping between actions and outcomes. Reinforcement learning (RL) has been used for solving the optimal control problem as well [25, 26, 27], except for an approximate optimal policy is learned. Instead of hand labeling data such as images of birds or planes in supervised learning, reinforcement learning self labels its data from sequential interactions with an environment through a reward function. The reward is a scalar value composed of how good or bad certain states are from a given action. This can be thought of as giving a dog a treat for sitting down when instructed with the appropriate command. Instead of a dog, one of the many algorithms within reinforcement learning would explore the space of possible solutions until a policy achieves the objective within reasonable bounds.

Recently reinforcement learning has learned to play Atari games from screen pixels only [28] and also beat human world champions at the game of GO [29] in discrete action spaces. For continuous action spaces, reinforcement learning has solved nonlinear physical problems such as path following of marine vehicles [30], autonomous car racing in the TORCS simulator [31], and robotic manipulation from camera images [32].

All of these applications use neural networks as a tool for representing the policy and the algorithms update the neural networks. Each of the applications use neural networks because the

state space is too large to represent through discretized tables. The task at hand in this work is to generalize to many different trailer parameters. Thus, the function approximator updated from one of the reinforcement learning algorithms may estimate a better policy than modern controls in the face of different model parameters.

1.1 Problem Statement

Combination vehicles vary in wheelbase, hitch length, weight distributions, and tire cornering stiffness. A modern controller may consists of a gain matrix that is multiplied against the error vector to generate an action. The calculation for the gains rely heavily on having an accurate model of the plant. Real-world applications cannot expect to have an updated model for each new trailer before use. Reinforcement learning, with neural networks as their function approximators, can allow for generalized control from its learned experience. The neural network gets encoded into fixed weights and biases that are multiplied against the states to determine the action. The thesis statement can be summarized as followed:

> *Reinforcement learning with neural networks is investigated for generalization and robustness in the path following task of backing up different tractor-trailers; this is done in comparison with the performance of a modern controller on the same path for various trailers, holding the gains fixed.*

1.2 Objectives and Scope

The purpose of this section is to solicit limited investigation of this thesis. The comparison is made with a constant velocity tractor-trailer system model, so the only form of actuation will be a continuous steering angle with continuous range. The docking task is completed with reverse motion only. Path planning is constrained to only occur from the initial starting position to the end position, in other words, re-planning the path continuously is not permitted. This is to assess the comparison of control methods, not planning methods. The paths are constrained to an $80 \times 80m^2$ area without obstacles. The task is considered failed if the combination vehicle jackknifes, or the angle between the trailer and tractor exceeds $90°$.

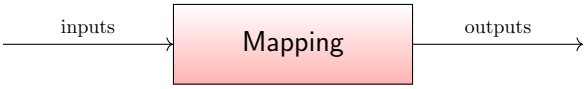
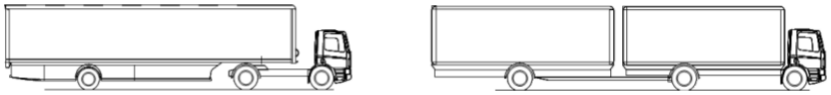


Figure 1.3: A function approximator takes inputs and maps it to an output.

The same information the modern controller gets as an input will be the same as what the reinforcement learning sees as suggested in Figure 1.3. This is a major contribution from this thesis because it bridges a gap between previous literature of modern controls and machine learning controls. First, the modern controller is designed for the purpose of path following. Machine learning is then used as a controller for the same path. This makes for an apples-to-apples comparison and should, in theory, make it easier for the neural networks to learn. The error from the path is calculated outside of the modern controller and reinforcement learning policy because this is a simple calculation. Although reinforcement learning could handle learning complex transformations for determining error from the path, it would make learning potentially intractable.

This study focuses on yaw control, not roll and pitch, which are considered beyond the scope of this thesis project. The trailers will be limited to one articulation such as semitrailers and truck-center axle trailers shown in Figure 1.4 from [33]. Full-trailers or vehicles with two articulations will not be evaluated.



(a) Tractor-semitrailer (b) Truck-centre axle trailer

Figure 1.4: Conventional vehicles with one articulation include (a) and (b). Tractor-semitrailers are more popular due to the coupling point being closer to the center of gravity on the tractor.

To indicate whether or not the controller places the trailer at the loading dock within an acceptable range, a criteria is created. The goal criteria is if the rear-most point of the trailer is within $15cm$ and the heading angle of the trailer is within $5.730°$ (0.1 radians) of the dock. This criteria suggests there is a region for this to be true, so the goal is actually not met unless the rear-most point of the trailer faces the correct side of the loading dock.

Table 1.1: Metrics used for comparing the performance of controllers include root mean squared (rms) and maximum errors from the path.

rms tractor angle [rad]	rms trailer angle [rad]	rms trailer lateral position [m]
max tractor angle [rad]	max trailer angle [rad]	max trailer lateral position [m]
min tractor angle @ goal [rad]	min trailer angle @ goal [rad]	boolean(goal)

The two methods of control will be evaluated over 100 random tracks for the following metrics found in Table 1.1. The root mean square provides for a quantifiable sense of how the controller is

doing along the path towards the goal, but hides instances along the path where the controller does poorly in certain scenarios. This is why the maximums will also be a metric.

Table 1.2: Tractor-Trailer Model Parameters

L2 [m]	h [m]	v [$\frac{m}{s}$]
8.192	0.228	-2.906
9.192	0.114	-2.459
10.192	0.0	-2.012
11.192	-0.114	-1.564
12.192	-0.228	-1.118

The modern controller gains are designed for a nominal case (yellow background in Table 7.2) for trailer wheelbase and hitch length. The weights are trained using reinforcement learning with the same nominal case. The fixed gains and weights are evaluated over the same 100 tracks for the metrics for each of the parameter changes in the table above. Testing both control methods over 100 new tracks they haven't seen before will demonstrate the generality and provide some statistical confidence. Varying the parameters one at a time will demonstrate each controllers' ability to perform when the tractor-trailer system changes from the designed.

In order to close the gap from simulation to reality, both control methods are assessed with changed control loop situations as described in Table 1.3. Sensor noise is increased using a Gaussian noise, where the standard deviation, σ, is altered. Then the controller frequency is varied from nominal with units of ms.

Table 1.3: Changed Control Loop Situations

sensor noise [σ]	controller frequency [ms]
0.0	1
0.03	10
0.04	80
0.05	400
0.06	600

In summary, this thesis is scoped to evaluate modern controls and a reinforcement learning policy for robust control. The use case is on a tractor-trailer autonomously being backed up to a loading dock with a path predetermined from a planner. As the tractor-trailer model changes or

the control loop situation gets more complicated, the performance of both controllers expected to degrade. The metrics listed above and the number of times the vehicle jackknifes will be used for characterizing this degradation.

1.3 Outline of the Thesis

Chapter 2: Background– gives a literature review and brief theory review on tractor-trailer models, path planners, modern controls and reinforcement learning.

Chapter 3: Tractor-Trailer–presents a derivation of the kinematic model of a tractor-trailer and discusses implementations in MATLAB and Python.

Chapter 4: Path Planning using Dubins Curves–describes the implementation of path planning with Dubins Curves and various tricks needed to represent the reference path in such a way the controllers can use it for error.

Chapter 5: Modern Controls–discusses the design, implementation and evaluation of the Linear Quadratic Regulator in MATLAB/Simulink. It comments on tuning of LQR gains and performance.

Chapter 6: Reinforcement learning–discusses the design, implementation and evaluation of the Deep Deterministic Policy Gradient (DDPG) in Python and Tensorflow. It publishes that several suggestions from the original paper did not actually result in better results. Tensorflow has some basic functionality problems that made getting repeatable results difficult, but a solution was found.

Chapter 7: Results–includes statistical claims and general trends of changing tractor-trailer parameters from the designed gains or trained weights. It also reveals controller loop degradation when adding sensor noise and varying control frequency.

Chapter 8: Summary and Conclusions–discusses and draws conclusions from the results and suggests future work.

Chapter 2

BACKGROUND

This chapter will introduce the reader to concepts that are considered as base knowledge for autonomously backing up a tractor-trailer to a loading dock. It is structured in such a way that the autonomous vehicle topics are consistent up until the point of the control method. An extensive survey was performed to identify the strengths and weaknesses of different machine learning tools and algorithms. It was ultimately determined that reinforcement learning offered the best chance of providing robust control. This chapter explains the relevant research to be able to compare control methods from the two different fields of modern controls and reinforcement learning.

Autonomous vehicles require three distinct tasks: 1) perception 2) path planning and 3) control. Perception involves using sensors to gauge where the vehicle is in its environment. Path planning includes determining which route to take to reach the desired location. It provides the desired values along the path for the controllers to target. The controller takes the desired values and feedback from sensors in order to determine how much to move an actuator to minimize the error from the path. In order to evaluate the controllers' performance in steering a tractor-trailer to follow a reverse path to a loading dock, it is necessary to use path planning. It may not, however, be necessary to model the sensors in perception to assess the controller performances because the information will already be available in simulation.

Usually the sensor information comes from some combination of cameras, RADARs, LiDARs, DGPS, or HD maps. RADAR is an acronym for Radio Detection and Ranging. A transmitter sends out high frequency radio pulses that are reflected by a physical obstacle directly back to the receiver. The time delay helps to calculate position. RADARs are able to operate in rain and fog, but current sensors typically scan horizontally. LiDAR is an acronym for Light Detection and Ranging. The idea is similar to RADARs, except shorter wavelengths are used including ultraviolet, visible or infrared. LiDARs can create 3D point clouds of information like in Figure 2.1 from [34], but the current sensors are expensive and are challenged by the rain.

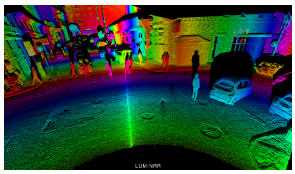


Figure 2.1: This is an example of an ultra high-resolution mapping image of an urban environment obtained using LiDAR.

Differential Global Positioning Systems (DGPS) provide better precision (2cm) than normal GPS (5-10m). A second GPS is actually used to subtract off the measurement error. The High Definition (HD) maps can also be thought of as a sensor because a digital representation of the environments are updated from many sources that are pushed to the cloud. The idea is that this information is used for high level planning and in the event that the other sensors are reading nonsensical data.

It has been found that not all of the sensors are perfect in every real-world scenario, so a combination of sensors can be used to eliminate false positives. Sensor fusion is used to update an internal map in the autonomous drive kit stack, i.e. the software. The map could be a topological map or simply a 2D Cartesian plane with x-y coordinates. The map is a simplified view of the environment and is used for the controller to reference. In other words, the controller does not operate directly from the sensors. Thus, perception is left for future investigations because the problem statement can still be answered without modeling sensors.

A great deal of similarities exist between the fields of modern controls and reinforcement learning principles trace back to Lev Pontryagin and Richard Bellman in the 1950s. Bellman created dynamic programming, which is used for solving exact solutions to optimal control. Approximate dynamic programming, or reinforcement learning, is used for large and challenging multistage decision problems where the exact solution may be computationally intractable [27]. An approximate optimal policy is learned because an Artificial Neural Network is used as a nonlinear function approximator. Keeping this in mind can help with understanding the concepts that lie ahead.

Pontryagin and Bellman used similar terminology in their fields such as states and actions which can be found in Figures 2.2 and 2.3 from [35]. They even wrote about some form of objective function, which could be a cost or a reward. In reality, the reward function is simply the negative of the cost function: $r(s, a) = -c(x, u)$. The reward, r, is a function of the states, s, and actions, a. Similarly, the cost, c, is a function of the states, x, and the actions, u. The two fields discuss the idea of a Markov Decision Process (MDP), so its relevance will be shown in a later section.

Figure 2.2: Richard Bellman used s_t for state, a_t for action, and $r(s, a)$ for reward function.

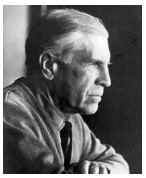


Figure 2.3: Lev Pontryagin used x_t for state, u_t for action, and $c(x, u)$ for cost function.

The following sections are meant to describe a snapshot of the relevant theory and reinforce them with literature. This chapter does not consist of rigorous mathematical proofs, but rather reveals the key insights from important equations which are relevant to the body of this thesis.

2.1 Tractor Trailer Models

A combination vehicle can be thought of as a single powered vehicle (tractor) with an un-steered vehicle (trailer) attached to the rear of it. The trailer gets pulled when moving forward, but pushed when moving backwards. With semitrailers, the "fifth wheel" connects the trailer to the tractor with the kingpin as depicted in Figure 2.4. Analogously, truck-center axle trailers use a drawbar hitch instead as shown in Figure 2.5.

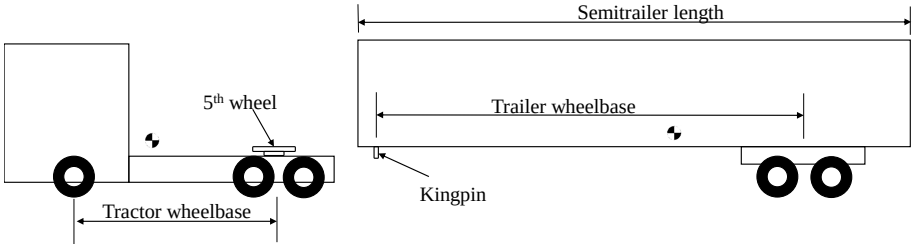


Figure 2.4: The semi-trailer connects to the tractor with the fifth wheel and kingpin.

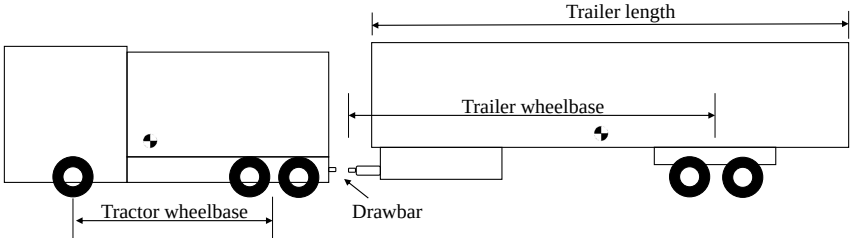


Figure 2.5: The truck-center axle trailers attach the trailer to the towing vehicle using a drawbar.

In reality, the fifth wheel on the towing vehicle are adjustable. This defines a variable hitch length. In addition to this, trailer wheelbases are also adjustable because the rear axles can be locked and the tractor can push or pull the trailer along rails. This is helpful for the drivers to maneuver in new scenarios. In summary, in addition to there being many trailers out there, the geometry can still be changed on the vehicles themselves.

Pneumatic tires are what connect the vehicles to the road. The rubber provides traction to the surface at the molecular level in the contact patch. Tires generate lateral forces, F_y, when the vehicle is negotiating a curve. This results in tire slip angles, α, which is the deviation of the angle from

the steered direction. The combination of lateral forces and slip angles allow for the yaw motion of a vehicle. The lateral tire forces are inherently non-linear. The tires can be thought of as springs until the tire reaches their limit. They are a function of normal load, F_z, and slip angle, α. More advanced models include lateral load transfer for changing normal loads, but this study is restricted to lateral motion only [36]. Figure 2.6 shows the lateral tire slip characteristics with the cornering stiffness, C_α, as the slope of the lateral force with units of Newtons and slip angle in degrees.

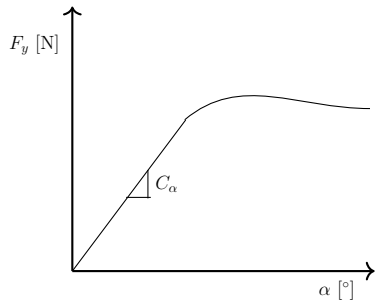


Figure 2.6: Tires are linear when the slip angles are small. C_α is the cornering stiffness.

Figure 2.7 depicts a tire and its forces where N is the normal load and ma_y is the mass times the lateral acceleration. The deflection of the tire generates the lateral force when induced with the lateral vehicle acceleration.

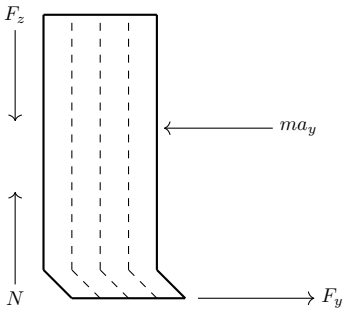


Figure 2.7: The depiction of a tire and its forces show the mechanics.

The steering characteristics of a vehicle can be expressed in terms of understeer, neutral steer or oversteer. The definitions depend on the relation of the front slip angles, α_1, and the rear slip angles, α_2. Tire stiffness and suspension springs affect the slip angles. Understeer and oversteer are important because can induce jackknifing at faster speeds.

$$\begin{aligned} &\alpha_1 > \alpha_2 \rightarrow \text{understeer} \\ &\alpha_1 = \alpha_2 \rightarrow \text{neutral steer} \\ &\alpha_1 < \alpha_2 \rightarrow \text{oversteer} \end{aligned} \tag{2.1}$$

Jackknifing is a phenomenon with articulated vehicles where the angle between the trailer and the tractor (hitch angle) becomes too great. This results in the trailer either hitting the tractor or leaves the combination vehicle in an undriveable state. This can happen at fast speeds going forward when the rear wheels of the tractor lose grip. The trailer could also swing due to the trailer axle wheels being locked during braking. This is an example of the trailer oversteering. At low speeds, this can occur very slowly due to an unskilled driver backing up.

Luijten [33] provides an excellent overview of tractor-trailer parameters and how they pertain to rearward amplification. The main focus in Luijten's paper is to evaluate stability criteria and parameter effects. Rearward amplification is a performance measure which quantifies the dynamic lateral acceleration amplification. Rearward amplification provides some insight to parameters that affect the dynamics of a combination vehicle. It can be calculated using the time histories of lateral accelerations from a lane change used as the input.

$$RA_t = \frac{max(|a_{y_{trailer}}|)}{max(|a_{y_{towing}}|)} \tag{2.2}$$

Trailer wheelbases are defined as the distance between the center of the trailer axles to the kingpin. A short wheelbase results in lower trailer yaw damping, evidently causing higher reward amplification values. Different types of connections between the trailer and tractor affect the dynamics. A fifth wheel connection is favorable because the mass of the trailer is closer to the center of gravity of the tractor. A towing hitch with a drawbar connection pushes the mass further away from the tractor. A longer drawbar reduces rearward amplification.

More articulations, or more trailers connected to each other, increases the rearward amplification. The weight distribution or effectively where the center of gravity is on the trailer also affects the rearward amplification. Having more mass located in the back of the trailer is an example that increases rearward amplification. Increasing the cornering stiffness of the rear axle of the towing vehicle or decreasing it in the front axle will increase the likelihood of understeer. Increasing the cornering stiffness of the trailer tyres by having more axles decreases the rearward amplification.

Kinetic and kinematic models of tractor-trailers occur in literature. At faster speeds, the kinetic model should better represent the system due to considering velocity, masses, forces, and the tires. While a vehicle is in motion, it has momentum from the mass and velocity. It takes more energy to stop a fast moving, heavy object. It is important to note that the kinetic model rotates about the center of gravity, whereas the kinematic model rotates about the rear axle.

Luijten [33] investigates stability with a kinetic model for driving forward. It is a bicycle model where the tires are lumped into one for each axle. The model is derived using Lagrangian mechanics

and only considers lateral motion, i.e. no roll or pitch. The model assumes constant normal load, in other words, the cornering stiffness of the tires remained constant. A similar model is also found in Pacejka [12]. Further experimental investigations of the model determined that this kinetic model broke down when driving backwards. Refer to Appendix A for a discussion on this topic.

A kinematic model, however, only considers velocity and geometry. This accounts for the primary motion, but may produce unrealistic results at higher speeds because inertial dynamic effects cause slip when a heavy vehicle drives through a curve. In addition to this, the kinematic model assumes the tires roll without slip. According to Karkee [37], the kinematic models are not valid for faster than $4.5\frac{m}{s}$ or approximately $10mph$. The kinematic model will seem to drive paths effortlessly, but it is not realistic due to more inherent dynamics occurring during higher speed motions. According to Elhassan [38], the average speed of drivers reversing a tractor-trailer is $1.012\frac{m}{s}$ or $2.25mph$. Thus, it should be reasonable to use a kinematic model for this evaluation.

The kinematic model utilizes the tractor wheelbase, trailer wheelbase and hitch length. Thus, its use reduces the number of parameters to vary for assessing the control methods. Wu in [5, 39] uses the kinematic bicycle model for lateral position control with a Linear Quadratic Regulator, but it is actually derived by Karkee [37]. Wu shows the model works for both forward and backwards, but only on a straight and a circle. Karkee evaluates a kinematic model and a kinetic model for system identification, which is a field for determining mathematical models of systems from measured data. The paper furthers their work by analyzing closed loop performance for a straight line and starting off path. The work in this thesis investigates actual path following from an initial starting point to the loading dock using their kinematic model.

2.2 Path Planning

Path planning can be thought of as the piano mover's problem: imagine trying orient a piano so that it will fit into the door of a house and through the halls. This is an example of moving a geometrically complex object through a complex environment without hitting obstacles. This section will describe why Dubins Curves were chosen by introducing some classical and modern path planning methods.

The motion planning problem is described in the real world, but lives in what is called the configuration space or map. Let us denote $q = (x, y, \psi)$ as the configuration space where x, y are Cartesian coordinates and ψ is a global angle defined by the unit circle. Holonomic motion planning simply means the object is allowed to move and rotate in any direction. So this would

be like a human walking and turning with ease or a mobile robot with omni-directional wheels. Non-holonomic motion planning is when the object cannot move in all directions, so this is like a car. Usually a car's front wheels are the only ones that steer, which explains why it is difficult to do parallel parking [40]. Non-holonomic motion planning must be used for a tractor-trailer.

There are three main approaches in motion planning according to Evestedt in [41]: 1) Combinatorial methods 2) Artificial Potential Field methods and 3) Sample-based methods. Combinatorial methods construct a graph of the configuration space called a roadmap by making polygonal representations. Artificial Potential Field methods are when the object moves under the action of a force generated from the potential field. The field is a combination of attractive potential towards the goal and a repulsive potential from obstacles.

Sample-based methods are probably the most applied methods used for real world planning. These methods have no prior information about the configuration space and use a collision detection module instead. The collision detection module is independent of the search algorithm, i.e. it is necessary to define the collision detection module unique to the problem. There are two approaches regarding how to build the graph using sample-based methods: 1) Discrete Deterministic and 2) Probabilistic Sampling.

Discrete methods involve many classic planning algorithms such as Dijkstras and the very popular derivative of it, A*. The task of the planning algorithm is to find a finite sequence of inputs that when applied transforms the initial state to the goal state. A graph is a set containing nodes where some pairs of nodes have a relationship. Each pair of related nodes is called an edge. Both Dijkstras and A* use a grid for the configuration space, so they can only find a solution with a certain resolution.

Probabilistic methods instead draw random samples from the configuration space and tries to connect them using a local steering method. Due to the randomness, it only offers a weak probabilistic completeness guarantee. Two main algorithms are the Probabilistic Road Maps (PRM) and Rapidly Exploring Random Trees (RRT). PRM builds a graph structure in the configuration space, but instead of fully characterizing the free configuration space, they generate random samples that are tested for collision and connected to produce a roadmap.

RRT is a special type of algorithm where each node on the graph only has one parent node. While not at the goal, the algorithm samples a random state in the configuration space. It determines the nearest neighbor using Euclidean distance and tries to extend towards it if there is no collision– adding it to the tree [42]. RRT is generally computationally fast, however, if no solution exists, the

algorithm will run forever. RRT solves many practical problems, but struggles with many narrow passages (slow convergence).

The aforementioned planning algorithms can be considered classical and have difficulty handling non-holonomic constraints without being modified. Motion planning for autonomous vehicles must consider some form of differential constraint according to Evestedt. Second order constraints would be kinetic model planning as in a two-point boundary value problem, which is similar to the planning in Model Predictive Control (MPC). If the problem formulation is convex, a global optimum can be obtained.

A first order constraint would be the kinematic model, only taking into account the geometry and velocity. An example of this is the Dubins car, which is constrained to only go forward. The Dubins Path is typically referred to as the shortest curve that connects two points in a two-dimensional Euclidean plane with a constraint on the radius. Dubins Curves consider the minimum turning radius through setting $u = \{-tan\delta_{max}, 0, tan\delta_{max}\}$.

$$\begin{aligned} \dot{x} &= cos\psi \\ \dot{y} &= sin\psi \\ \dot{\psi} &= u \end{aligned} \tag{2.3}$$

Between any two configurations, the shortest path for the Dubins car can always be expressed as a combination of no more than three motion primitives: S, L, R. [42]. This corresponds to Straight, Left, or Right. Essentially, this refers to 0, u, or $-u$. In the textbook Planning Algorithms, LaValle defines a word to be a sequence of SLR. There are ten possible path words of length three, but he proved only six are optimal:

$$LRL, RLR, LSL, LSR, RSL, RSR \tag{2.4}$$

This form of path planning considers both position and orientation, also known as pose. Dubins determines the shortest path by computing all six paths and determining which path has the minimum distance. Figure 2.8 demonstrates four of the six possible paths to get from q_0 to q_1 using only straights, lefts, and rights. The starting orientation is 90° (vertical) and the ending orientation is 45° (to the right of vertical). In this scenario, using just lefts and rights would not result in a valid path.

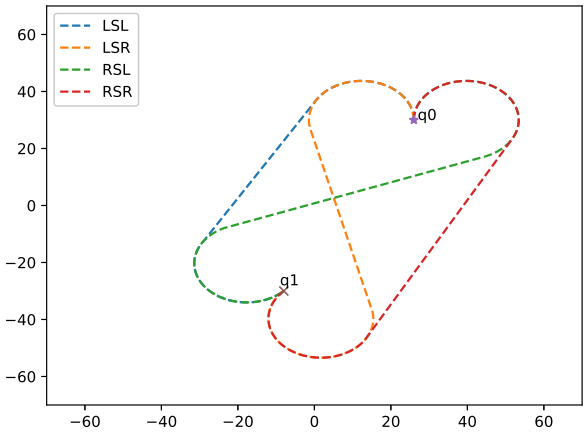


Figure 2.8: These Dubins curves combine three motion primitives, connecting straight lines to tangents of circles of the minimum turning radius.

Figure 2.9 demonstrates two of the six possible paths to get from q_0 to q_1 using a combination of lefts and rights. The starting pose is 90° and the ending pose is 270°. This is a situation where using a straight is not possible due to the minimum turning radius.

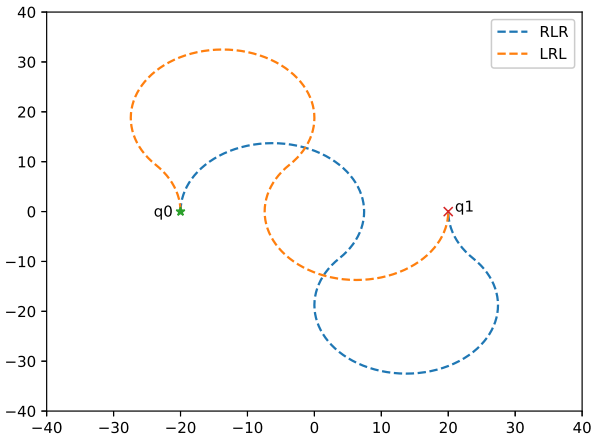


Figure 2.9: The Dubins curves here instead combine the tangents of circles to get to qg.

Dubins Curves are useful for environments where there are no obstructions. Interesting work by Elhassan [38] and Evestedt [41] use Dubins curves to further constrain Rapidly Exploring Random Trees (RRT) where obstacles are present. Given that the problem defined in this thesis is not

dealing with obstacles, it is much more computationally efficient to simply use Dubins Curves for path planning.

2.3 Reference Values

A substantial amount of research effort has been put into bridging the gap between the controller and the path. The main objective is to find a sequence of reference values so the controller with feedback brings the system states closer to the pre-defined reference path or trajectory. According to Boyali [43], retrieving the reference values from the path so the controller can follow it can be summarized into three different control objectives:

Point-to-Point Control: is setting a starting and ending configuration, but not caring how it gets to the goal.

Path Following: is where the vehicle has to follow a desired geometric path from a given initial configuration.

Trajectory Following: is when the vehicle follows a series of positions with time associated with it, i.e. setting a desired position and velocity.

From the three control objectives, the reference values will either be framed as a path following or a trajectory following task because the path to the goal matters in this thesis. Existing research for autonomous vehicles using the Linear Quadratic Regulator (LQR) sometimes fail to show the step in between designing the controller and feeding the proper desired states to produce the error. Designing the controller can be done with a reference of zero and starting with an offset. This must be the reason why literature often fails to show the gap between the controller and the reference path.

Boyali, Kapania, Murray, and Baur in [43, 44, 45] all mention a feedforward term is used with the LQR. The feedforward term is curvature, which is the inverse of radius. Boyoli shows a controller schematic where the input is zero and the feedforward term is added, so it is misleading that one can follow a path with only that information. This suggests that the feedforward term is solely used to introduce the reference path. Kapania shows a control diagram where the feedforward term is determined from the curvature and velocity as a function of time, but also does not show a target value.

Murray shows a control diagram where the reference is put into the trajectory generator and a desired and feed forward term are output from that. The trajectory generation is done by the differential flatness of the system. Baur competed in the 2017 Formula Student Driverless competition in

Germany [46] where he used the LQR. Bauer's paper, however, clearly shows what appears to be a path planner which gives a desired term and an optional feedforward term. Thus, the path planner should generate the reference values.

Nagy in [47] states that with manipulation like in 6 DOF robotic arms, kinematic equations are solved for a point in configuration space. In mobile robots, one must solve an underdetermined differential equation for a vector trajectory which achieves some unknown state trajectory that ends at the goal posture. Nagy uses inverse kinematics and third order polynomials. Rajamani [48] also mentions the usage of clothoids for the road model, which are curves proportional to its length. Research suggests there are many ways to represent the reference path, but Dubins Curves actually return reference values suitable for path following. It is not considered a trajectory because there is no time associated with the reference values. Dubins Curves are also beneficial because the kinematic model for the tractor-trailer does not have states that include velocities.

Figure 2.10 displays a bicycle car model and the lateral error of the rear axle to the path to help understand the reference values. The reference path, s, is denoted with the dashed line. The vehicle heading, ψ, should match the path orientation ψ_d which are provided by Dubins Curves. At the same time, the lateral error is measured from the reference point, y_d, and the lateral coordinate of the rear axle on the vehicle.

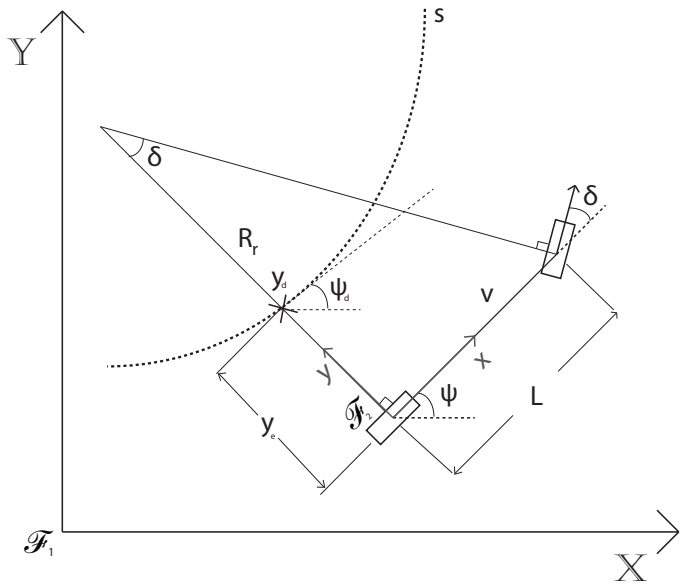


Figure 2.10: The vehicle is in local coordinates whereas the path is in global coordinates.

Literature in Rajamani and Snider [48, 49] claims the vehicle system equations are rewritten in terms of path coordinates or error dynamics. Writing the system in path coordinates, however,

may hide the underlying mechanisms between the plant and the reference path. Additionally, this can cause the problem to become a Linear Time Varying (LTV) one and as such, new gains would need be calculated for every iteration with modern controls. Instead of writing the equations in path coordinates, it was recognized that the error space is in the global coordinates. This means the error can be transformed back to the vehicle coordinate system.

Lin and Martin [50, 51] use a rotation matrix to describe the error distance from the path. A direction cosine matrix is used as a frame rotation, or a passive rotation. This simply means the error is described in the vehicle or local coordinate system such that the controller regulates the error to zero. The lateral position and orientation errors are frame rotated by the current heading angle, not the heading angle error.

$$\begin{bmatrix} x_e \\ y_e \\ \psi_e \end{bmatrix} = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r - x \\ y_r - y \\ \psi_r - \psi \end{bmatrix} \tag{2.5}$$

Even if the lateral position is primarily controlled as in Wu, one would still need to measure the x-coordinate in order to get the correct y_e term. Focusing on lateral position and orientation is reasonable with constant velocity. The two describe vehicle placement on the path and using constant speed reduces the complexity when Dubins Curves does not return a time requirement. The reference values are used to calculate the errors and the Direction Cosine Matrix is applied to those errors as input to the controllers for the path following task in this thesis.

2.4 Markov Decision Processes

Both modern controls and reinforcement learning can be related when talking about states and actions. Both control methods take the current states are given as input and the controller produces an action. They do not make a decision using memory, or a history of the states and actions. The Markov property suggests if the current action, a_t, and state, s_t is available, then one can ignore the history of of the states and actions.

$$\begin{matrix} a_{t-1}, a_{t-2}, ..., a_0 \\ s_{t-1}, s_{t-2}, ..., s_0 \end{matrix} \tag{2.6}$$

This primarily means that the behavior depends only on the current observation. In other words, if the same thing is observed twice, the same thing is done twice. This happens regardless of what happened before. A process that has the Markov property is a Markov Decision Process (MDP). This is demonstrated with Figure 2.11

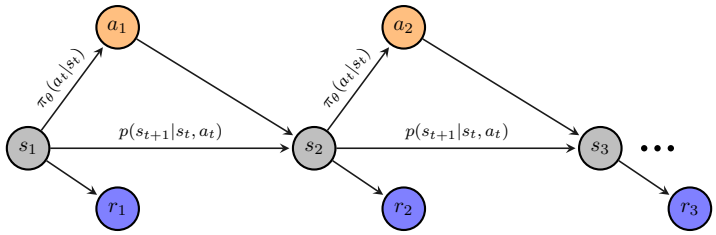


Figure 2.11: Markov Decision Process State Transition can be represented as such.

Given state, s_1, a policy π will make an action, a_1, which should result in s_2 with some probability, p. This state transition will have a reward or a cost associated with it. The process should continue sequentially without needing the history of state transitions because the dynamics are understood, i.e. with differential equations or statistical probability.

A Non-Markovian property requires the history of states and actions to be able to make a decision on what to do next. In terms of modern controls, perhaps the position of the trailer's rear axle is not measured. This would make the problem Non-Markovian, which is also known as a Partially Observed Markov Decision Process (POMDP). Modern controls can use what is called an Observer to alleviate problems with this, which will be covered in the next section. With reinforcement learning, one can transform a POMDP into an MDP by using a history of state transitions. This may involve using a Recurrent Neural Network with a sequence of observations, which will also be explained later.

Solving the problem of optimal control has been studied by both fields of reinforcement learning and modern controls. In fact, the continuous version of the Markov Decision Process is actually the Hamilton-Jacobi-Equation [26] – a key equation used for solving the gains for the Linear Quadratic Regulator.

2.5 Modern Controls

Modern controls is different than classical controls in the sense that systems are evaluated in the time domain instead of the frequency domain. In addition to this, modern controls are useful for Multiple Input, Multiple Output (MIMO) problems whereas Classical Controls is restricted to Single Input, Single Output (SISO). Systems are represented by state space, which uses first order differential equations with linear algebra. There are many ways to derive state space equations such

as Newtonian Mechanics, Linear Graphs, and Lagrangian Mechanics.

$$\begin{aligned} \dot{\underline{x}} &= A\underline{x} + B\underline{u} \\ \underline{y} &= C\underline{x} + D\underline{u} \end{aligned} \tag{2.7}$$

A represents the state matrix and is $n \times n$ where n is the number of states. B is the input matrix and is shape $n \times r$. C is the output matrix ($m \times n$). D is the feedthrough matrix and is shape $m \times r$. In most physical systems, the feedthrough matrix is filled with zeros. The state space representation shown here is for Linear Time Invariant (LTI) systems.

One can then develop a Full State Feedback Control Law where $\underline{u} = K(\underline{r} - \underline{x})$. The gain matrix, K, is fixed for LTI systems and heavily depend on the system modeling done beforehand. Figure 2.12 shows the control loop of the full state feedback controller. The plant is another name for the system.

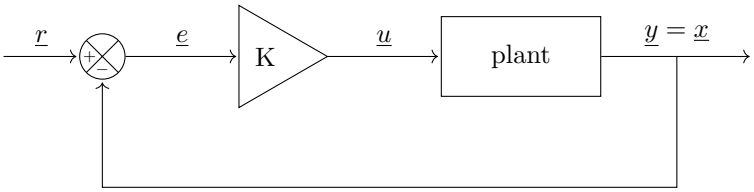


Figure 2.12: The Full State Feedback architecture requires a desired vector, r, and for all the states, x to be measured.

The poles of the system transfer function are the eigenvalues of the system matrix, A. When closing the loop with feedback, the gains in K shift the eigenvalues. Adjusting the pole locations affect the system response for stability, rise time, and settling time.

$$\begin{aligned} \dot{\underline{x}} &= A\underline{x} + BK(\underline{r} - \underline{x}) \\ \dot{\underline{x}} &= (A - BK)\underline{x} + B\underline{r} \end{aligned} \tag{2.8}$$

There is a difference between asymptotically unstable, marginally stable and asymptotically stable. When all the eigenvalues are on the left half side of the graph in Figure 2.13, the system is asymptotically stable. In other words, the response will eventually be driven to zero. If any single eigenvalue is on the right half side of the graph, the system is thus asymptotically unstable. The response can grow unbounded. However, if a single eigenvalue resides on the crossover line, the system is marginally stable. This means the response will harmonically oscillate which can be seen in Figure 2.14.

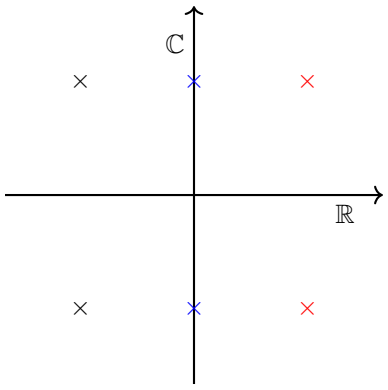


Figure 2.13: Asymptotically unstable, marginally stable and asymptotically stable.

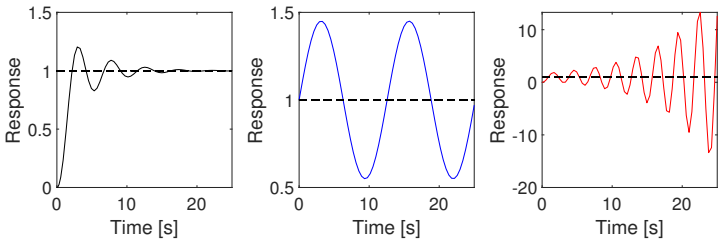


Figure 2.14: The colors correspond to asymptotically stable, marginally stable, and unstable.

The tractor-trailer models are open loop asymptotically stable going forward if under a certain speed, but when travelling in reverse, the system has positive poles [33, 37] indicating instability. Assume a tractor and trailer are positioned in a loading dock with a non-zero hitch angle. Without driver intervention, the trailer traveling in reverse will continue to increase the angle between the tractor and trailer. The vehicle does not stabilize itself where the tractor and trailer headings are perfectly in line. The undesirable physics get worse at higher speeds. By closing the loop of the controller, it is possible to stabilize the system or shift the eigenvalues from the right hand side to the left hand side of the real axis.

When one has the full state or can measure all the states, systems are usually checked for controllability. Having controllability means the system states can be manipulated by actuators. The actuators are the system input. The controllability matrix can be found as:

$$P = [B\ AB\ A^2B...] \tag{2.9}$$

If the rank of P is the same as the number of states n, then the system is controllable. If the system is uncontrollable, then there is no sense in designing a controller for the system. If one has the

ability to measure full state and the system is controllable, one can arbitrarily place poles using gains. One might place poles on second order systems to have certain damping ratios or peak time step responses.

Observability is a check for when the system can utilize a state estimator to help when states cannot be measured. State estimators or observers are used to ensure full state feedback using the L matrix. In other words, the state is reconstructed by looking at the outputs. Figure 2.15 shows the layout of a full state feedback architecture with an observer.

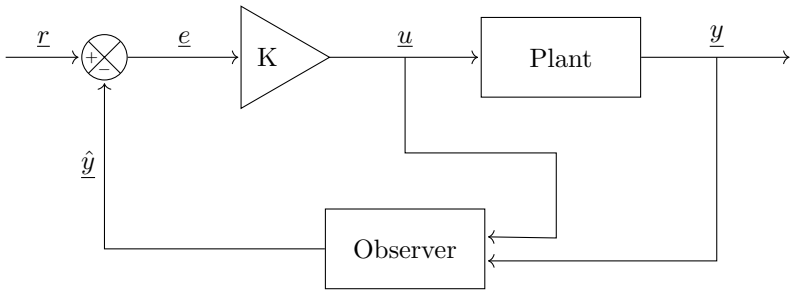


Figure 2.15: The Full State Feedback with Observer architecture uses a model of the system and the action to fill in missing measured states.

The output error is defined such that the estimated output, $\hat{y}$, was subtracted from the output. If some arbitrary gain matrix L is multiplied by the output error and added to the system, the result is a linear homogeneous differential equation.

$$\begin{aligned} \underline{e}_y &= \underline{y} - \underline{\hat{y}} \\ \underline{\dot{x}} &= A\underline{x} + B\underline{u} + L\underline{e_y} \\ \underline{\hat{y}} &= C\underline{\hat{x}} \quad \& \quad \underline{y} = C\underline{x} \\ \underline{\dot{x}} &= (A - LC)\underline{x} + (A - LC)\underline{\hat{x}} \end{aligned} \tag{2.10}$$

Whether or not an observer can be successfully used can be determined by using the observability matrix:

$$Q = \begin{bmatrix} C \\ CA \\ CA^2 \end{bmatrix} \tag{2.11}$$

If the rank of Q is the same as the number of states, then an observer can be used. One can then place poles on second order systems to have certain damping ratios or peak time step responses just as the full state feedback architecture gains. Usually the observers are designed such the system runs faster than the actual system.

The Separation Principle suggests that one can put the full state feedback gains K and state estimator gains L together to close the loop.

$$\begin{bmatrix} \dot{\underline{x}} \\ \dot{\underline{\hat{x}}} \end{bmatrix} = \begin{bmatrix} A & -BK \\ LC & A - BK - LC \end{bmatrix} \begin{bmatrix} \underline{x} \\ \underline{\hat{x}} \end{bmatrix} + \begin{bmatrix} B \\ B \end{bmatrix} \underline{r} \tag{2.12}$$

The problem of Optimal Control in terms of Modern Controls is one that finds the control law $\underline{u} = K(\underline{r} - \underline{x})$ such that it minimizes the cost function, J. Minimizing J is the same as maximizing $-J$. There are many different types of objective functions one can make, but the one relating to the Linear Quadratic Regulator (LQR) can be sometimes thought of as optimizing for minimum effort.

$$J = \int_0^{t_f} (\underline{x}^T Q \underline{x} + \underline{u}^T R \underline{u}) dt \tag{2.13}$$

Richard Bellman's Principle of Optimality states [52]:

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

The Hamilton-Jacobi-Bellman (HJB) equations takes a discrete problem and converts it into a continuous one. The solution to the HJB is the Algebraic Riccati Equation. For a Full State Feedback Problem, the Algebraic Riccati Equation to solve is:

$$A^T S + SA + G^T Q G - (SB + G^T Q H)(H^T Q H + \rho R)^{-1}(B^T S + H^T Q G) = 0 \tag{2.14}$$

S is the solution to the Algebraic Riccati Equation, which is then used to derive the gain matrix, K. The reader is referred to Brogan [53] for a foundation on Modern Controls and Hespanha [54] for applications of implementation in MATLAB.

When one uses a State Estimator called a Kalman Filter that adapts to noise, it is considered the Linear Quadratic Gaussian (LQG). Model Predictive Control (MPC) is very similar to LQR, but is much more computationally expensive. Instead of the gains being fixed, the differential equations are used as a model to sample trajectories as a convex optimization problem. It is done for a finite horizon and then only the first action is used per timestep. MPC continuously solves the Algebraic Ricatti Equation to determine the best gains at each timestep. These methods can improve robustness.

Wu from [5] uses LQR to control the lateral position and orientation of a tractor-trailer for both going forward and backward along both a circular and straight reference path. In Wu's thesis [39], investigations are done with LQG and MPC on both kinematic and kinetic models. Barbosa in [55]

uses LQR to control a kinetic model for going forward. Rimmer in [6] uses full state feedback to control a kinetic model in reverse along a path within $400mm$. Interestingly Ljungvist in [56] uses the LQR for stabilizing the hitch angles of two trailers, but uses a Lyapunov controller in conjunction for the overall path control. Evestedt [57] uses cascaded controllers to reverse two trailers in line with one another. A pure pursuit controller is used on the outer loop, but then input into an LQR to stabilize the trailers.

Model Predictive Control would be an excellent controller for backing up a tractor-trailer because of its ability to impose constraints such as preventing jackknifing. It does so by sampling different trajectories to the goal at every timestep, filtering out the decisions from the bundle that would cause a jackknife. The more often replanning occurs, the less perfect the plan or system model needs to be. The problem with using MPC that in this thesis, however, is that it makes it difficult to compare modern controls to reinforcement learning. MPC would provide updated trajectories every iteration, which may would not require Dubins Curves. The trajectories may result in different reference values depending on the action taken from the calculated gains or the neural network. Thus, this thesis uses the Linear Quadratic Regulator with fixed gains. This allows for a direct comparison of controllers on the same paths.

2.6 Artificial Neural Networks

Machine Learning (ML) has three main domains: 1) Supervised learning 2) Unsupervised learning and 3) Reinforcement learning. Supervised learning requires labeled test samples to train on. Unsupervised learning can be thought of as allowing the algorithm to cluster unlabeled data into potentially meaningful categories. Reinforcement learning is still considered to be an active area of research according to Chollet in [58]. Artificial Intelligence (AI) can be considered as the big umbrella of the field, see Figure 2.16. Inside it contains Machine Learning (ML), which holds the aforementioned domains.

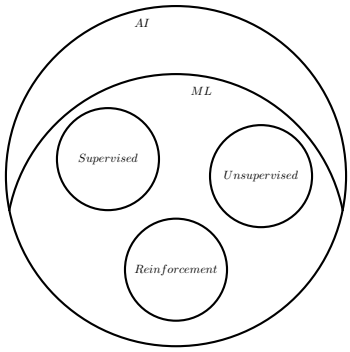


Figure 2.16: AI is the big umbrella that houses the three domains of machine learning.

The reader might be wondering where Artificial Neural Networks (ANNs or NNs) fit in Figure 2.16. Neural Networks are a tool for applying different forms of learning. They are universal nonlinear function approximators. The truth is, however, they are not the only tool for this! There also exists Decision Tree Learning, Gradient Boosting Machines, and Markov Chains according to Norvig in[59]. Neural Networks are beneficial for large datasets and nonlinear pattern recognition. There are a few main structures of NNs such as fully connected networks, Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) outlined by Goodfellow in [60].

As mentioned before, NNs are nonlinear function approximators. The math behind it is a linear transformation plus a translation, effectively creating an affine transformation. Weights are the connection strengths between neurons. The applied weights stretch and rotate the input, but a bias translates it. After the calculation is done, the result is put through an activation function, which provides a nonlinear expansion. An example of a fully connected neural network is shown in Figure 2.17 with the inputs as x_1, x_2 as layer i and the output as y_1. The circles are the neurons where the weights are the connection strengths to them. The dashed circles are the biases.

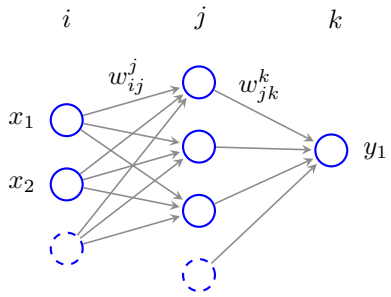


Figure 2.17: An example of a fully connected neural network demonstrate the weights and biases between the inputs and outputs.

Equation 2.15 shows the weighted sum vector output, $\underline{z}^L$, of a single layer. The weights, W^L, are a matrix that is multiplied by the output vector after the activation function, a^{L-1}, of the previous layer. The bias vector, $\underline{b}^L$, consists of biases for each of the neurons.

$$\underline{z}^L = W^L \underline{a}^{L-1} + \underline{b}^L \tag{2.15}$$

This can also be written as an affine transformation as seen quite often in computer graphics:

$$\underline{z}^L = [W \mid \underline{b}]^L \, \underline{a}^{L-1} \tag{2.16}$$

An example weight and bias matrix, $[W \mid \underline{b}]^k$, would look like the following with indices starting at zero. The result of the matrix multiplication below results in the weighted sum vector from layer k, before the activation function is used.

$$\begin{bmatrix} z_0^k \\ 1 \end{bmatrix} = \begin{bmatrix} w_{00}^k & w_{01}^k & w_{02}^k & b_0^k \\ w_{10}^k & w_{11}^k & w_{12}^k & b_1^k \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_0^j \\ a_1^j \\ a_2^j \\ 1 \end{bmatrix} \tag{2.17}$$

Activation functions, σ, can be different within each of the layers. Various activation functions may include a Rectified Linear Unit (RELU), sigmoid, or a softmax. The important property of an activation function is that it is differentiable at every value. This is important so that the weights can be updated through backpropagation, the method of updating weights and biases. The sigmoid function looks like the following in Figure 2.18.

$$\sigma = \frac{1}{1 + e^{-z}} \tag{2.18}$$

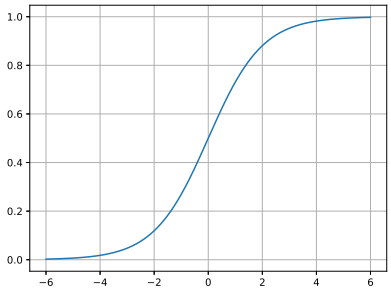


Figure 2.18: The sigmoid function is useful for when the output should be between 0 and 1.

The output of the layer is finally determined by applying the activation function to the weighted sum vector, creating $\underline{a}_L$. This can then be used for the calculation of the next layer, however in the

example shown, this result is actually the output of the neural network, y_1.

$$\underline{a}^L = \sigma(\underline{z}^L) \tag{2.19}$$

The objective is to adjust the weights and biases to provide a mathematical mapping of input to output. Training consists of learning the best set of weights and biases in all of the layers to minimize a loss using backpropagation, which is also known as Stochastic Gradient Descent [60]. The neural network gets initialized with random weight and bias values. It then makes a forward prediction of what the output should be given the inputs. An error is calculated between the labeled value and the predicted value. This is then used in conjunction with partial derivatives across each layer going back all the way to the first layer. The chained partials are then used to update the weights and biases.

Training is usually done in batches, or a subset of the entire data to promote randomness. An epoch is completed after the neural network has looked at all of the training data. If the batch size was chosen to be 256 and the training data included 1024 samples, an epoch would complete after four batches. The stochasticity in Stochastic Gradient Descent comes from a number of things: 1) randomly selecting what training data to look at and 2) what the initial weight and bias values are. Continuing the process would look like the loss value drunkenly traveling down a valley until it is a minimum. The process is repeated until the results are of acceptable performance. The gradient of the loss, E, is characterized by the following:

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial W^0} \\ \frac{\partial E}{\partial b^0} \\ \vdots \\ \frac{\partial E}{\partial W^n} \\ \frac{\partial E}{\partial b^n} \end{bmatrix} \tag{2.20}$$

$$\frac{\partial E}{\partial W^L} = \frac{\partial z^L}{\partial W^L}\frac{\partial a^L}{\partial z^L}\frac{\partial E}{\partial a^L} \qquad \frac{\partial E}{\partial b^L} = \frac{\partial z^L}{\partial b^L}\frac{\partial a^L}{\partial z^L}\frac{\partial E}{\partial a^L} \tag{2.21}$$

The weights are then updated using a learning rate, α, multiplied against the gradient of the error. The learning rate is a tuning factor used for speeding up or slowing down the updates. As the partial derivative is negative, the weights and biases increase. This can be seen in the following equation:

$$W_{new} \leftarrow W_{old} - \alpha\frac{\partial E}{\partial W} \tag{2.22}$$

This form up an update must occur for the weights in each layer, but also for the biases. Thus, the combination of all the parameters being updated are usually denoted with θ.

$$\theta_{new} \leftarrow \theta_{old} - \alpha\nabla E \tag{2.23}$$

An example that is easily understood is if training data is available for the classifications of foods. Broccoli could be an example of the vegetable class, which has low values of calories and sweetness. The measure of the calories and sweetness are known as the feature values. A pretzel belongs to the starch class because it has a high value of calories, but lower on the sweetness. Cheesecake is included into the desert class due to its high levels of calories and sweetness. Diet Coca cola belongs to the diet soda class due to their low level of calories and high sweetness. Gathering enough of these creates a vector of feature values, called patterns. If the patterns are labeled, then it is considered training data. If the patterns are not labeled, then this can be considered testing data. The objective is to train on a given set of patterns and be able to generalize to patterns the NN has not necessarily seen before.

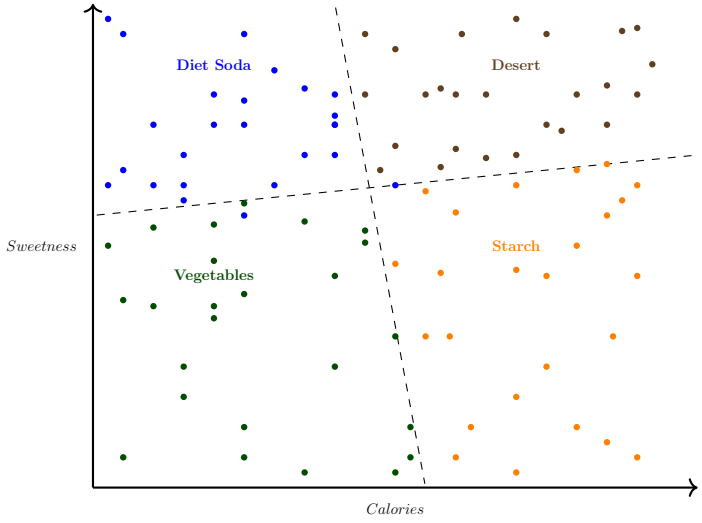


Figure 2.19: A neural network would implicitly divide pattern spaces into classes based on training data.

As can be seen by Figure 2.19, the data could be spatially separated by what are called hyperplanes (dashed lines). A hyperplane is the line effectively created by the affine transformation from a single neuron. Each neuron is basically a decision point of whether or not the input data is on one side of the hyperplane or the other because of the biases. The decision here is the result of a dot product projection. This is easily visualized in 2D, but much more incomprehensible with n-dimensional data. Using linear algebra and a translation, a computer can be taught to perform on isolated tasks.

Without using backpropagation, a thought experiment may be to think of how one could select the weights and biases such that the output gives the correct class of foods. Two neurons can be used in the first layer to create two hyperplanes. The next layer can have four neurons for the four

convex shapes bounded by the hyperplanes. The final layer could have four neurons which could be thought of as logic gates for determining which class the given feature value likely belongs to.

Three layers are usually sufficient enough to theoretically identify objects with convex spaces [60]. The input is considered layer zero, or in the case of Figure 2.20, layer i. With input values for calories and sweetness, patterns in the data can be approximately classified for the different food groups.

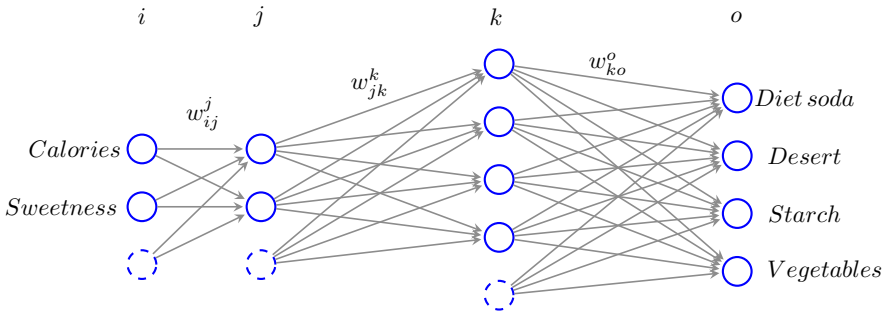


Figure 2.20: A manual creation of a three layer neural network for classifying foods is shown for using neurons like logic gates. The dashed neurons are the bias neurons.

The thought experiment provides insight for a how a rudimentary NN's weights and biases could classify data. More difficult problems could require a higher capacity NN architecture with more layers. Furthermore, supervised learning could wish to solve regression problems as well, which is attempting to approximate continuous values such as the housing market prices. A NN for a regression problem would simply use different loss functions for backpropagation.

Deep learning (DL) is considered an extension of machine learning where large networks with many layers are used to generate internal representations of complex problems. Deep neural networks can often be difficult to train due to the many layers. Some solutions to these problems include weight regularization, batch normalization, dropout, early stopping, data augmentation and transfer learning [60]. All of these try to improve the performance of deep learning models.

Convolutional Neural Networks are useful for image processing. It can be used to classify objects like cones, bikes, or giraffes, or hand written digits for peoples' addresses on envelopes. A CNN is a structure of a NN that is well suited for computer vision tasks. Images usually have three layers for the different colors in RGB. This causes the data to become 4-dimensional, i.e. the vectors to store the information are arrays of arrays.

A 3×3 filter striding across the pixels of an image are used to detect features. CNNs are extremely useful because they are invariant, meaning they have the ability to recognize features that are rotated and skewed.

A convolution layer stores an array of features the filters want from their weights, which becomes process intensive. To speed things up, a max pooling layer is applied to take the max value in a 2×2 window. It is effectively taking a step back and looking at the major features. Figure 2.21 depicts the convolution and max pooling layers over an image to eventually determine which of the numbers $0 - 9$ a handwritten digit is.

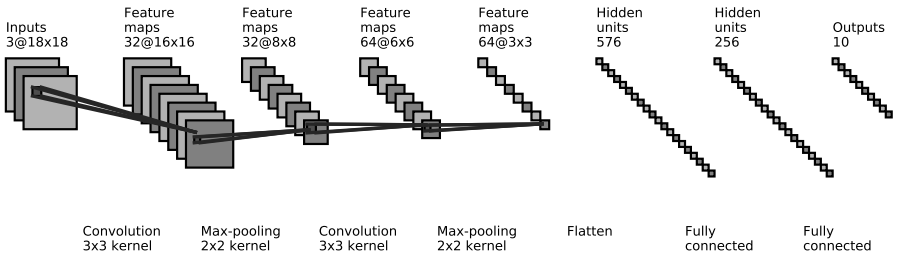


Figure 2.21: A Convolutional Neural Network has dominated the computer vision field in recent years.

Recurrent Neural Networks (RNN) are useful for problems when sequences are needed such as Natural Language Processing. RNN's are interesting because they introduce feedback into a NN. This means they are able to retain short term memory of a sentence as it goes by. They have a state machine behavior. As shown in Figure 2.22, an RNN can conceptually be unrolled to visualize training it. In other words, it is a temporal snapshot of the same network one after another.

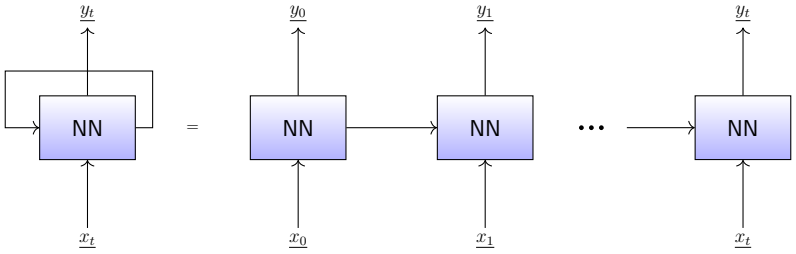


Figure 2.22: Recurrrent Neural Networks essentially have some memory in the form of feedback

They have a single problem: vanishing gradients. It is an effect where the model becomes untrainable with many layers deep. The RNN with Long Term Short Memory (LSTM) is capable of learning long-term dependencies. The RNN with LSTM is designed to effectively forget and store information while things are still unrolled. It can sort of be thought of as using neurons as logic gates and using a conveyor belt line of information passed through all of the unrolled networks. For more information, refer to Goodfellow [60].

All of the neural network models are trained over multiple epochs, or number of times the model looks at the complete dataset to update the weights. One must be careful to not over train the model. Training longer does not necessarily mean better results are obtained. This phenomenon occurs when when the accuracy of the training data increases, but the validation data hovers around a constant value. Validation data is simply withheld data the model does not train on to double check the reality of its fitting. The neural network could be creating hyperplanes that will only perform well on the data it has trained on. Figure 2.23's example suggests that it is likely best to stop training after two epochs on this specific problem. Overfitting is the term used when a function approximator predicts well on the trained data, but does not generalize to new ones. Having more data or use of regularizers may help alleviate this problem.

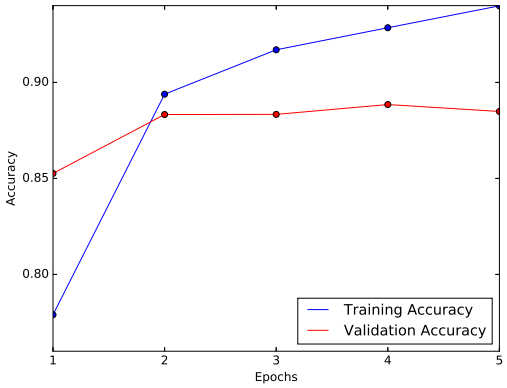


Figure 2.23: Plotting the training and validation accuracies during training is useful to know when to stop. From this plot, it looks like it is reasonable to stop training after two epochs and settle for approximately 87% accuracy.

The Truck Backer-Upper problem was originally studied with Neural Networks in the 1990 with Nguyen and Widrow in [17, 61, 62]. The Truck Backer-Upper problem statement is to reverse a trailer to a loading dock at position (0, 0) without jackknifing. Reinforcement Learning was investigated, however, a neural controller was trained by a complex method of adjusting its weights to minimize an objective function using gradient descent through an emulator.

$$J = (\frac{x_{dock} - x_{trailer}}{1000})^2 + (\frac{y_{dock} - y_{trailer}}{1000})^2 + \frac{1}{50}(\frac{\theta_{dock} - \theta_{trailer}}{\pi})^2$$

Another neural network was first trained to essentially act as an emulator or a surrogate model of the mathematical simulator. During training, the truck backs up randomly going through many cycles with randomly selected steering signals. The emulator learns to generate the next positional

state vector when given the present state vector and the steering signal. The emulator was trained using supervised learning.

A 2 layer neural controller containing 25 neurons was used for both the emulator and the neural controller. Apparently a neural network emulator was needed instead of differential equations because only the final state of the vehicle was available. This meant that training the neural controller was done in a supervised learning fashion by backpropagating the emulator NN for the equivalent errors of the controller in the intermediate steps.

The training of the controller was divided into several lessons with increasing level of difficulty. It took altogether $20,000$ backups to train the controller, which Nguyen considered to be substantial. Many later works following Nguyen did not use an emulator, rather hard coded equations were used as a simulator and the loss for backpropagation could be achieved by other means.

Jenkins [18] decomposed the Truck Backer-Upper problem into subtasks. In doing this, only two hidden neurons were needed. This method helped solve a problem of computation in the early 90s, but requires substantial modifications for each new problem. Jenkins points out the fact that Nguyen's network learned to make a decision based on the absolute location and orientation of the truck. In that scenario, a similar truck orientation may not be able to exploit what has been learned because it is in a different location. Ho [20] uses Jenkins network, but updates the weights with a genetic algorithm.

Kinjo [8] uses a neuro-controller for the Truck Backer-Upper and also uses a genetic algorithm to update the network instead of backpropagation. Kinjo also uses an LQR to evaluate the neuro-controller's performance. Both controllers were evaluated in simulation and an experiment. It was stated that the neuro-controller exhibited better performance in both accuracy and response speed. Kinjo mentions the neuro-controllers ability to achieve intended control even when the real environment is changed. It is worth reiterating that most Truck Backer-Upper problems disregard following a specific path, which does not show extrapolation to new docks and forms of autonomy.

Much of the aforementioned work requires significant manual tuning of the algorithms for the specific problem. It often requires collecting trajectory rollouts from other controllers. The rollouts are used as labeled data, which can have a probability distribution mismatch when used in a supervised learning method. Reinforcement Learning, however, provides self labeled training data by providing a framework for learning with interaction in an environment.

2.7 Reinforcement Learning

Reinforcement learning (RL) is still very much an active area of research for sequential decision making, but more people are looking to it for solving engineering problems. Resources for RL are Sutton and Barto's textbook [25], the Berkeley CS294 videos [35] and David Silver's videos [63]. The framework consists of an agent that interacts with the environment to receive rewards continuously until a reasonable policy is learned which can be seen in Figure 2.24. The agent consists of a function approximator, like a neural network. The environment may consist of simulation or real world robotics.

This form of machine learning is different than others because there is no supervisor hand labeling data, only a reward signal. The reward function is unique to problems, which consists of a scalar value representing a measure of how good certain state transitions are in the environment. The neural network predicts what the next action should be, takes the action in the environment, then gets a reward value for the next state. The agent's job is to take actions sequentially to maximize the cumulative future reward.

The process is usually repeated in episodes, instead of epochs. A simulator provides state transitions given a certain action, which are the number of iterations in an episode. Many episodes will be used in training and may terminate early due to reaching a state that results in going beyond the point of return. The neural network learns the policy, usually denoted π, because it is a mapping of states to actions. This is the controller; this is the agent's behavior. The estimate for the optimal policy function is π^*. Reinforcement learning approximates the solution to the optimal control problem.

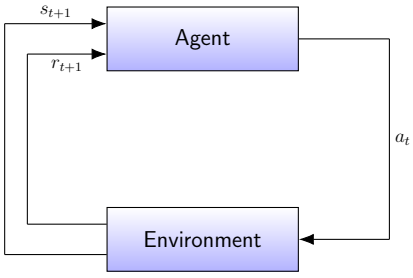


Figure 2.24: Reinforcement Learning Agent Environment

The agent must explore different actions to find more information above the environment. It must also exploit known information in order to maximize the reward. Continuously interacting with the environment must consist of exploring as well as exploiting. For example, a dog may hear a command to roll over from the owner, but does not know what needs to be done. The dog may

try to sit because he or she knew that has worked in the past. This is an example of exploiting the current policy or choosing the greedy policy. Since the dog did not perform the correct action, the owner does not give a treat. Then the dog tries to lay down and receives half of a treat. This is an example of exploring. The owner commands again, "roll over." The dog finally rolls to the side and receives the full treat.

It may be better to sacrifice immediate reward to gain more of the long-term reward. Maximizing the future reward can be formally defined by maximizing the expected discounted return (reward), G_t. Discounting is done with the discount rate, γ, which is defined to be $0 \leq \gamma \leq 1$. It determines the present value of future rewards, in other words, a reward received k time steps in the future is only worth γ^{k-1} times what it would be worth if it were received immediately.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T \tag{2.24}$$

A smaller γ promotes maximizing immediate rewards, whereas a large gamma encourages maximizing rewards down the road. An example of this is if an 85 year old person won the lottery. If he could obtain a \$90 million today versus \$160 million in fifteen years, taking the money sooner may be in his best interest.

Different reinforcement algorithms change the way the reward updates the function approximator. The following flowchart in Figure 2.25 does not list all the algorithms, however, it summarizes the primary and modern algorithms used. Most of the other algorithms are variants of these.

The first decision in the reinforcement learning algorithm flowchart is if one needs to use a Model-Based or Model-Free approach. The question one needs to ask is whether or not the system dynamics are known? If not, perhaps the goal is to implement an agent on real life hardware to learn by interacting with the real world. This is usually the case when there are way too many unknowns and simulation is difficult. One might actually wish to start in a simulated environment, then maybe proceed to the physical world. Training in simulation is faster, cheaper, and more safe than unleashing a robot to interact with the world. The Model-Free methods learn a policy without needing to use a model of the system dynamics a priori.

On the other hand, a general idea of the dynamics might be known because a simulated environment is already built for the agent to interact with. Therefore, one could use something like model predictive control (MPC) to use trajectory optimization to direct policy learning and avoid local optima [64]. This makes learning more efficient because of the ability to choose regions to explore. This idea is very similar to Guided Policy Search (GPS). Dyna is another Model-Based RL, but instead, a surrogate model is learned through the history of state transitions using supervised

learning. The policy is then trained with the planning from the model as well as the experience from interacting with the environment. The problem here is that if the learned model is no good, there are two sources of approximation errors. Considering that planning is involved with the learning, Model-Based methods are not used in this thesis.

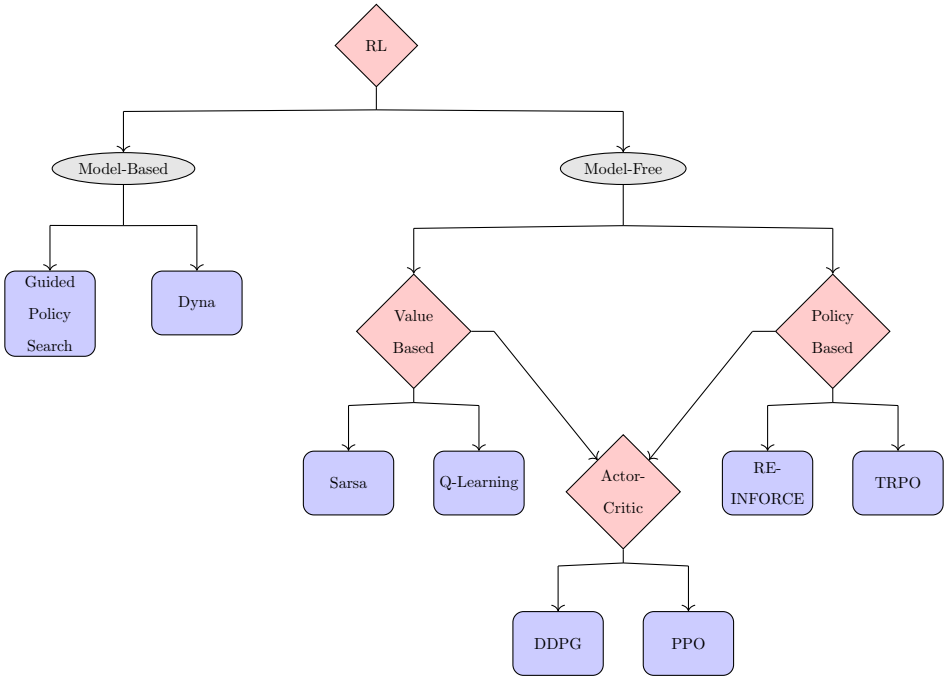


Figure 2.25: This flowchart can help determine which algorithm is best to use. The algorithm names are in the blue square boxes.

When looking at Model-Free algorithms, the strongest argument for which type to use is whether continuous or discrete actions are needed. For robotics and autonomous vehicles, policy based methods are preferred because they can handle continuous actions. Policy based methods directly differentiate the policy using stochastic gradient ascent (instead of descent). It is ascent because these methods try to maximize good actions so that it is more likely and the bad is made less likely.

2.7.1 Policy Based Methods

REINFORCE is the most basic policy gradient algorithm. It is a Monte Carlo Method, meaning the learning occurs once an episode is over and the summation of the discounted rewards for states and actions is used for the update, using the action as the label. Policy based methods are on-policy, which basically means the agent learns on the "job." REINFORCE and most policy gradient methods suffer from high variance in their gradients, thus it typically takes substantially more samples to train. The benefit of these algorithms is that they are theoretically guaranteed to converge [25].

Algorithm 1 REINFORCE: Monte-Carlo Policy-Gradient Control

Randomly initialize policy network $\pi(s|a, \theta)$ with weights θ

for n episodes **do**
  Generate samples $s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t$ following $\pi(s_t|a_t, \theta)$
  for each step of the samples, $t = 0, 1, ...T - 1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
    $\theta \leftarrow \theta + \alpha\gamma^t G \nabla \ln \pi(a_t|s_t, \theta)$

A way to help with the high variance is to subtract the discounted reward by a baseline. The most natural baseline is what is called a state value function. A value function, V, is an estimate of the return for being in a certain state. The optimal estimated state value function is V^*. So this means there are two neural networks. One for the policy and the other for estimating the state value.

Algorithm 2 REINFORCE with Baseline

Randomly initialize policy network $\pi(s|a, \theta)$ with weights θ
Randomly initialize state value network $V(s, \omega)$ with weights ω

for n episodes **do**
  Generate samples $s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t$ following $\pi(s_t|a_t, \theta)$
  for each step of the samples, $t = 0, 1, ...T - 1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
    $\delta \leftarrow G - V(s_t, \omega)$
    $\omega \leftarrow \omega + \alpha^{\omega} \delta \nabla V(s_t, \omega)$
    $\theta \leftarrow \theta + \alpha^{\theta} \gamma^t \delta \nabla \ln \pi(a_t|s_t, \theta)$

The Trust Region Policy Optimization (TRPO) algorithm is another policy based method that produces near monotonic improvements for continuous stochastic actions. TRPO essentially constrains the updates to the network to prevent the new policy from diverging too far from the existing policy by using the natural gradient. Deterministic actions are useful in robotics and autonomous vehicles because picking the exact same action given the exact same state is usually how modern controls work. Stochastic policies will select an action according to a learned probability distribution. If one of the sensors were not reliable or noisy, a stochastic policy may do better. For example, a stochastic policy would do better in rock paper scissors than a deterministic one. Thus, TRPO unfortunately is not the best algorithm for this application.

2.7.2 Value Based Methods

One of the most popular RL algorithms people hear about is a Value Based Method: Q-Learning. These methods are not Monte Carlo updates, but rather temporal difference updates. Temporal

difference refers to making estimates off of estimates, which is also known as bootstrapping. This allows for updating the weights at every timestep, so learning can improve while in the episode. Think about driving a car to work. If traffic suddenly occurred, with Monte Carlo updates, one would be forced to continue driving the same path to work stubbornly. With temporal difference updates, one could decide to take surface streets.

Instead of a neural network changing the policy directly, it is implicitly represented by a value function. This value function, usually an action-state value function, represents how good an action is to take given a state. Continuous actions are difficult here because there would be too many actions to evaluate to figure out the maximum value function. The optimal policy is still estimated since the agent simply needs to select the discrete action the Q^* function said was best.

Exploration is thus achieved through an ϵ-greedy approach where some probability, ϵ, of the time in the episode, the action is selected at random from the discrete options. The remaining actions are greedily selected, or based on what the function approximator currently thinks is best.

The Bellman optimality equation expresses the fact that the action-state value under an optimal policy must equal the expected return for the best action from that action-state. This is important because it allows for the value based algorithms to express values of states as values of other states. The Bellman optimality equation enables the use of iterative approaches for calculating estimated optimal policies. Bellman's equation for deterministic actions is shown below:

$$Q^*(s,a) = r + \gamma max_{a'} Q(s',a') \tag{2.25}$$

Value based methods use an iterative approach for creating a loss to update the neural networks using the temporal difference error. This is the self-labeling of reinforcement learning. In other words, this is maximizing the future expected reward. The temporal difference error, δ_t, is a measure of the difference between the estimated action-state value and the better estimate, $r_{t+1} + \gamma Q(s',a')$.

$$\delta_t = r_{t+1} + \gamma Q(s',a') - Q(s,a) \tag{2.26}$$

Sarsa is also an on-policy algorithm like REINFORCE, which remember, means learning on the "job." Sarsa is very similar to Q-learning, but distinguishes itself by being on-policy. This means that Sarsa will perform better while training, however, it is more likely to find only a locally optimal policy. Chris Gatti implemented Sarsa(λ) in [65], but this meant he only had three discrete actions for steering the tractor. The actions were full left, full right, and straight. The λ in the Sarsa algorithm is the eligibility trace, which allows for updating with n-steps of discounted rewards as

opposed to just one. Using the eligibility traces can be thought of as being somewhere in between Monte Carlo (full episode update) and temporal difference update (one step).

Algorithm 3 Sarsa (on-policy) Temporal Difference Control

Randomly Initialize state-action value network $Q(s|a, \omega)$ with weights ω

for n episodes **do**
  Observe initial s
  $a \leftarrow$ action from ϵ-greedy policy
  for each step of episode **do**
    Take action a, observe reward r and next state s'
    if s' is terminal **then**
      $\omega \leftarrow \omega + \alpha[r - Q(s, a, \omega)]\nabla Q(s, a, \omega)$ Go to next episode
    $a' \leftarrow$ action from ϵ-greedy policy for s
    $\omega \leftarrow \omega + \alpha[r + \gamma Q(s', a', \omega) - Q(s, a, \omega)]\nabla Q(s, a, \omega)$
    $s \leftarrow s'$
    $a \leftarrow a'$

One does not have to use a neural network for Sarsa, a look up table can be implemented. If the states are just horizontal position and velocity and there are just three actions, the table is three dimensional. The issue is that the states are continuous. So a method of discretizing or binning the states can be used.

One of the problems with Value Based methods is that the label for updating the weights in the neural network keeps changing, so it is like a dog chasing its tail. This is not like supervised learning where the labels are static trying to classify a bird or a plane. The tabular methods don't have problems learning, but they are limited to smaller toy problems when the states get huge.

Q-learning is also a temporal difference algorithm, however, it is considered off-policy. This means the network is learning by "looking over someone's shoulder." Benefits for off-policy is the ability to learn from observing policies from other agents to maybe learn a better policy. This can be thought of as being more risky in the episode; this is why Sarsa is likely to perform better in the episode initially. Sarsa will likely just learn a worse policy than Q-learning [25]. Off-policy also allows for experiences to be re-used in batch updates as opposed to just one sample at a time. Learning from a batch is similar to how training is done in supervised learning. This also eliminates the potential of un-learning good policies due to noise of just one sample.

Algorithm 4 Tabular Q-Learning (off-policy) Temporal Difference Control

Initialize Q(s, a) with zeros

for n episodes **do**
  Observe initial s
  for each step of episode **do**
    $a \leftarrow$ from ϵ-greedy policy
    Take action a , observe reward r and next state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
  until s is terminal

Q-learning is technically not performing gradient descent because there is no gradient through the max operator in $r + \gamma max_a Q(s', a')$. Training is also difficult because Q-learning tends to have a bias due making maximum estimates of the maximums. According to Sutton and Barto [25], there are certain conditions when Value Based methods are not guaranteed to converge. In fact, sometimes they can diverge and grow out without bound. The deadly triad includes 1) Non-linear Function approximation 2) Bootstrapping and 3) Off-Policy.

Even though Q-learning has shown the best performance in the early days with tables as the function approximators, this divergence phenomenon has been an on-going research question. However, in 2015, Mnih from David Silver's team at Google's Deepmind [28] found a way to stabilize the algorithm. This was a major breakthrough for reinforcement learning because the same network parameters were able to achieve performance comparable to humans across 49 Atari video games. The algorithm was called the Deep Q-Network (DQN).

Algorithm 5 DQN

Randomly initialize state-action value network $Q(s|a, \theta)$ with weights θ
Initialize target state-action-value network $Q'(s|a, \theta')$ with weights $\theta' \leftarrow \theta$
Initialize replay buffer B

for n episodes **do**
  Observe initial s
  for each step of episode **do**
    $a \leftarrow$ from ϵ-greedy policy using target network
    Take action a, observe reward r and next state s'
    Store transition (s_t, a_t, r_t, s_{t+1}) in B
    Sample a random minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from B
    if s' is terminal **then**
      $y_i = r_i$
    Set $y_i = r_i + \gamma max_{a'} Q'(s_{i+1}, a', \theta')$
    Update $Q(s|a, \theta)$ by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i, \theta))^2$
    Every C steps, copy $\theta' \leftarrow \theta$
    $s \leftarrow s'$

David Silver said it wasn't the cleanest approach, but what they did was to have a second network called the target network. They froze the weights in the target network and did the bootstrapping estimates from it because the predictions were consistent. Every now and again they would copy the weights from the primary network over to the target network and freeze them again. Copying the weights over may be done every $10,000$ steps. The target network helped with the correlation problem, i.e. the dog chasing its tail. In addition to this, they utilized a replay buffer to batch update the network to avoid the problem with Sarsa potentially training on noise from one sample. So despite not having theoretical convergence guarantees, the field of reinforcement learning has found ways to make the algorithms work in practice.

2.7.3 Actor-Critic

Policy Based methods and Value Based methods both have their problems. Policy Based methods have high variance and require many samples to learn since it trains by Monte Carlo (only after the episode has finished). Value Based methods have convergence problems, bias, and can only do discrete actions. There is, however, a solution. Actor-Critic methods blend the benefits of both together. The actor can be thought of as the policy network, μ. The critic can be thought of as the action-state value estimator, Q. This means there are two neural networks.

Proximal Policy Optimization (PPO) is an algorithm that allows for continuous actions. It has the benefits of constraining the size of the policy updates like TRPO, but it is much simpler to implement according Schulman [66]. A critic is used to reduce the variance with the policy that is learned. PPO trains with large batch sizes collected from multiple episodes of the same policy (on-policy), then it throws away the samples. The frequency of the weight updates occur once the batch size has been fulfilled. The algorithm explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in the action selection depends on the initial conditions and the update rule, which may include clipping. Over the course of training, exploring becomes less random as the update rule encourages it to exploit what it has learned. A stochastic policy was not as attractive for autonomously steering a tractor-trailer, so this algorithm was not pursued.

The Deep Deterministic Policy Gradient (DDPG) is also an Actor-Critic method used for continuous actions. The algorithm utilizes a replay buffer and the trick with using the target network as found in DQN. Hence, DDPG is considered an off-policy algorithm. The frequency of the weight updates happen at every iteration since it is a temporal difference method.

Algorithm 6 Deep Deterministic Policy Gradient

Randomly initialize critic network $Q(s|a, \theta^Q)$ with weights θ^Q
Randomly initialize actor network $\mu(s, \theta^\mu)$ with weights θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer B

for n episodes **do**
  Initialize a random process N for action exploration
  Observe initial s
  for each step of episode **do**
    $a \leftarrow \mu(s_t, \theta^\mu) + N_t$
    Take action a, observe reward r and next state s'
    Store transition (s_t, a_t, r_t, s_{t+1}) in B
    Sample a random minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from B
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}, \theta^{\mu'}), \theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (Q(s_i, a_i), \theta^Q) - y_i)^2$
    Update the actor policy using the sampled policy gradient:
$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a, \theta^Q)|_{s=s_i, a=\mu(s_i)}) \nabla_{\theta^\mu} \mu(s, \theta^\mu)|_{s_i}$$
    Update target critic network, $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    Update target actor network, $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$

DDPG was popularized by the paper from Lillicrap [67] describing where the same network parameters were applied to over 20 different toy continuous environments. Instead of having an epsilon greedy approach to explore, noise is injected to the action using Ornstein & Uhlenbeck [68]. The method essentially adds some momentum to the noise so the agent could learn following some similar action for awhile. Now that there is no discrete maximum Q to determine the next action, the policy network conveniently determines the action. The critic network then can evaluate a Q value for the given action and states. Figure 2.26 reorganizes the agent-environment diagram such that the DDPG agent can be better understood.

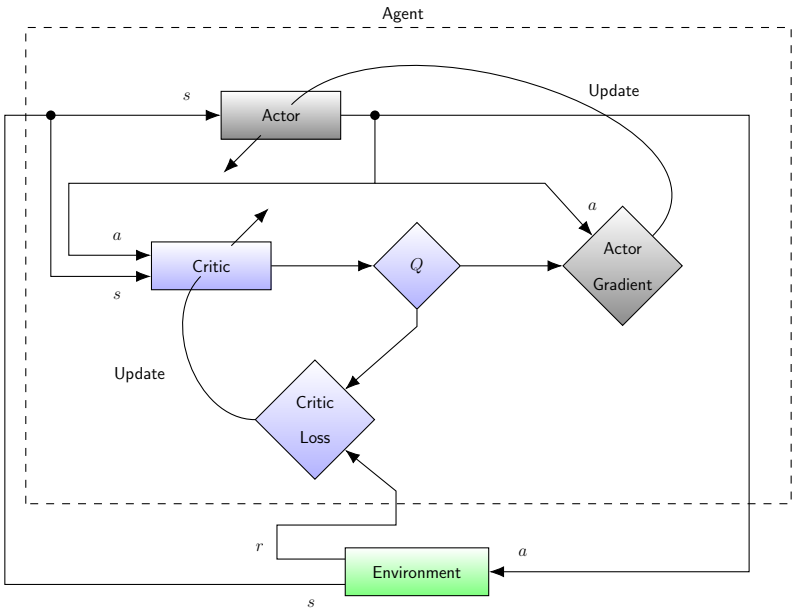


Figure 2.26: The DDPG algorithm conveniently uses continuous actions from the actor and the a critic provides a measure of how well it did; hence the interplay between the actor and critic

Since target networks are used, this means that essentially four networks are used in DDPG–but only two of them are trained. Instead of abruptly copying the weights over, the target networks are updated slowly using polyak averaging.

$$\theta' = \tau\theta + (1 - \tau)\theta' \tag{2.27}$$

The reward is used to create the temporal difference error for the critic loss to update the critic. Usually the critic needs to get trained first to be able to improve the actor. The gradient of the Q value with respect to the chosen action is used to update the actor through gradient ascent. This process is done at every time step, with a random batch from the replay buffer. Figure 2.27 shows the neural network architecture for the actor. Table 2.1 highlights the hyperparameters for each of the layers. Hyperparameters are network architecture choices such as number neurons, activation functions, and weight initialization schemes.

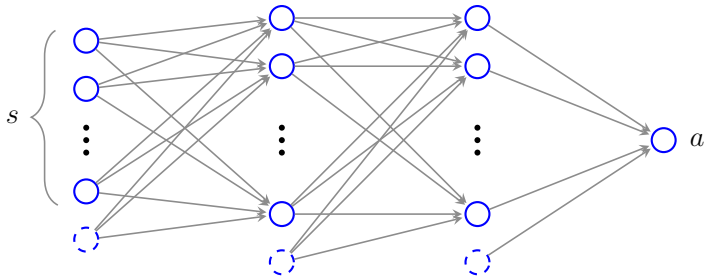


Figure 2.27: The actor is the policy network. Once trained, the actor is simply used as the controller.

Table 2.1: Actor Hyperparameters

Hyperparameter	Inputs	Layer 1	Layer 2	Outputs
# Neurons	# states	400	300	# actions
Activation	None	RELU	RELU	tanh
Initializer	None	$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$	$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$	[-.003, .003]
Batch Normalization	Yes	Yes	Yes	Yes
L_2 Regularizer	None	None	None	None

The initialization of the two hidden layer weights and biases are using a a uniform random metric bounded where f is the number of inputs into the network. The output weights and biases are initialized as such in order to make the initial output of the network small. Batch normalization is useful for scenarios where the states have wildly different units. It modifies the inputs such that there is a mean of zero and a variance of one. This helps to minimize covariance shift during training, by ensuring that each layer receives a whitened input. The mean and variance are stored at each layer such that when it comes time for testing or exploring, the output is rescaled and shifted back to represent the population [69].

The critic network architecture is slightly different than normal because the action is not added until the second hidden layer as can be seen in Figure 2.28. This is likely to allow the critic to estimate the Q value based on the actual action because the previous layers are batch normalized. Table 2.2 shows the various hyperparameters for the critic network, which primarily are different from the actor with the inclusion a regularizer.

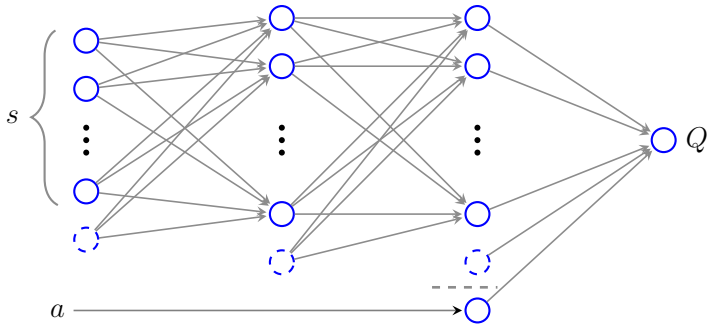


Figure 2.28: The action of the actor is fed into the critic, but at the second hidden layer.

Table 2.2: Critic Hyperparameters

Hyperparameter	Inputs	Layer 1	Layer 2	Outputs
# Neurons	# states	400	300	# actions
Activation	None	RELU	RELU	linear
Initializer	None	$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$	$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$	[-.003, .003]
Batch Normalization	Yes	Yes	None	None
L_2 Regularizer	None	.01	.01	.01

The critic's number of neurons and initialization of weights and biases are similar to the actor. The difference is that only the layer preceding the augmented action input is batch normalized. The states are not normalized, or at least not explicitly said by Lillicrap. The weights are regularized by an L_2 Norm weight decay of 0.01. This is to penalize large weights. The weights are multiplied by the decay rate and added to the loss term during backpropagation.

For its ability to handle deterministic, continuous action spaces and its usage of modern tricks in reinforcement learning, DDPG was selected as the algorithm in this thesis. Using Dubins Curves as a path planner from randomized initial and ending poses, the DDPG policy will be benchmarked against the LQR controller for autonomously backing up a kinematic model of a tractor-trailer to a loading dock in simulation.

Chapter 3

TRACTOR-TRAILER

A mathematical model of a tractor-trailer system is needed for simulation, but also for designing the gains for the linear quadratic regulator. The mathematical model will also be used for training the weights for the reinforcement learning algorithm.

A kinetic model for the combination vehicle was investigated, however, it was discovered that the equations broke down when traveling in reverse. The Luijten kinetic model equations [33] diverge when any non-zero steering input is given to the model when driving backwards. It was originally believed that the model worked going backwards since it could drive in a straight in all four directions backwards, with zero input. This just proved the initial conditions with the equations were valid. Some time was spent re-deriving the equations such that the force on the tires were in the opposite direction, but this ended up requiring more time than allotted. Refer to appendix A for the derivation of the kinetic model that is usable for going forwards.

This chapter investigates the derivation of the kinematic tractor-trailer equations, explores the system eigenvalues for going forward and backward, discusses implementations in MATLAB and Python, and validates the equations against IPG Carmaker.

3.1 Tractor Trailer Kinematic Model

Position and angle orientation of the trailer will be needed to describe the deviation from a defined path in Cartesian coordinates, (x, y), with an angle orientation represented using the unit circle. The position and angle orientation of the tractor will be needed too because the system input is steering angle, δ. The trailer's motion is dependent on what the tractor's motion. The positions are represented by the rear axle of the tractor and the rear axle of the trailer.

The heading angle of the tractor is defined to be ψ_1 whereas the heading of the trailer is set to be ψ_2. The hitch angle is defined to be the angle between the tractor and trailer, θ. A hitch angle of larger than 90° is considered jackknifing for this study. The position of the rear axle of the tractor is described as (x_1, y_1). The location of the rear axle of the tractor is set as (x_2, y_2). These are all described in global coordinates.

The kinematic equations can be modeled as the bicycle model with another vehicle attached to it. The bicycle model lumps the two wheels of an axle into a single tire. The tractor wheelbase is denoted as L_1 and the wheelbase of the trailer is set as L_2. The hitch length, or the distance from

the the rear axle to the coupling point in the kinematic model, is described as h_1. These parameters can be found in Figure 3.1.

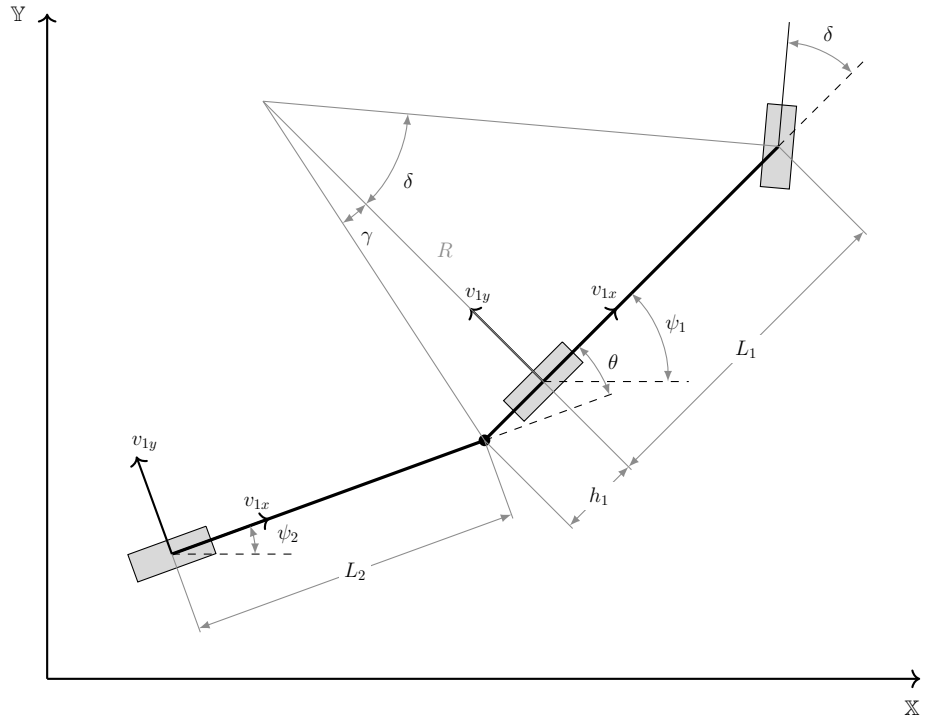


Figure 3.1: The kinematic equations can be modeled using a bicycle with an additional vehicle attached to it. The combination vehicle is described in global coordinates $(\mathbb{X}, \mathbb{Y})$ and orientation ψ with the unit circle.

The kinematic model assumes there is no lateral tire slip, i.e. the vehicle tires will all move in the direction it is facing. The velocity of the tractor, v_{1x}, is set to be constant. Roll and pitch motions of the combination vehicle are ignored. The velocity of the tractor in the $\mathbb{X}$ and $\mathbb{Y}$ directions are given by:

$$\dot{x}_1 = v_{1x} \cos \psi_1 \tag{3.1}$$
$$\dot{y}_1 = v_{1x} \sin \psi_1 \tag{3.2}$$

The velocity equations are described using the global Cartesian coordinate system. The derivation is explained for the combination vehicle going forward, however, the model is still valid when driving in reverse.

For high speed turning, in other words high lateral acceleration, there must be lateral forces generated by the tires to negotiate the corner. The tires track about the instantaneous center where the radius of the turn, R, is measured relative to the center of gravity. However for low speeds, the vehicle tracks about the geometric center and R is measured relative to the rear axle of the

bicycle model. Considering low speeds, the kinematic model can thus define an approximation for the geometric steer angle otherwise known as Ackermann Steering angle [36]. The length of the tractor wheelbase is L_1.

$$\tan\delta = \frac{L_1}{R} \tag{3.3}$$

The linear velocity of the tractor can be expressed as an angular velocity, or the kinematic constraint.

$$v_{1x} = R\dot{\psi}_1 \tag{3.4}$$

This can then be rewritten for R such that equation 3.4 is substituted into equation 3.3.

$$\tan\delta = \frac{L_1}{v_{1x}}\dot{\psi}_1$$

Rearranging the result for $\dot{\psi}_1$ results in a differential equation for the tractor yaw rate, i.e. the derivative of the heading of the tractor.

$$\dot{\psi}_1 = \frac{v_{1x}}{L_1}\tan\delta \tag{3.5}$$

The lateral velocity of the trailer at the hitch point is kinematically constrained by the length of the trailer, L_2, and its yaw rate. The equation can be rearranged such that the trailer yaw rate is on the left hand side.

$$\begin{aligned} v_{h2y} &= L_2\dot{\psi}_2 \\ \dot{\psi}_2 &= \frac{v_{h2y}}{L_2} \end{aligned} \tag{3.6}$$

The lateral velocity of the trailer, v_{h2y}, is not yet known. It can be determined through what Karkee [37] describes as resolving the velocities at the hitch point on both the tractor and the trailer. This can be seen with Figure 3.2.

The velocity of the hitch point on the tractor is at some angle, γ, away from the longitudinal axis. The relationship between the angle and the components of $\vec{v_h}$ is as follows. The equation can be rewritten with v_{h1y} on the left hand side.

$$\begin{aligned} \tan\gamma &= \frac{v_{h1y}}{v_{h1x}} \\ v_{h1y} &= v_{h1x}\tan\gamma \end{aligned} \tag{3.7}$$

Another relationship for γ can be determined geometrically, which can then be substituted into equation 3.7.

$$\tan\gamma = \frac{h}{R} \tag{3.8}$$

$$v_{h1y} = v_{h1x}\frac{h}{R} \tag{3.9}$$

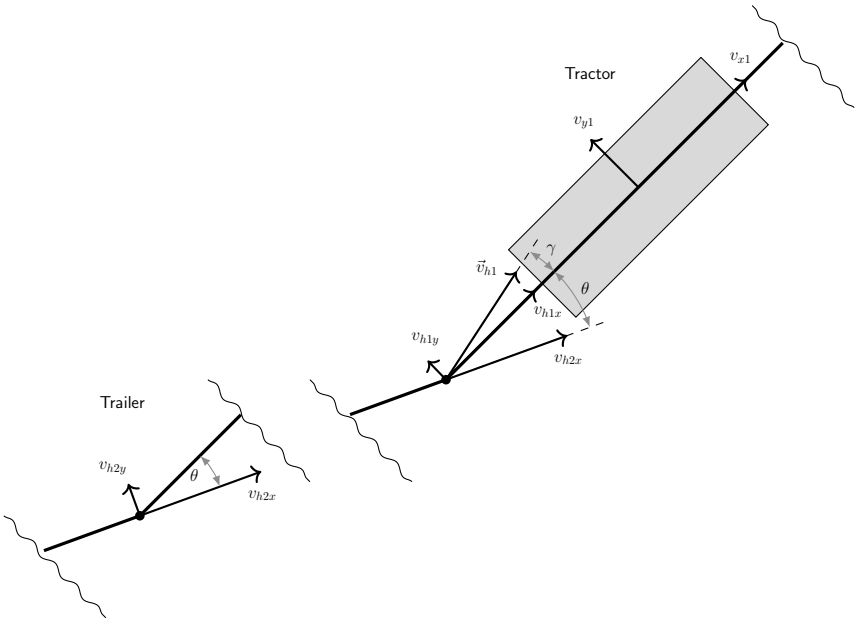


Figure 3.2: This zoom-in of the tractor trailer model highlights the separation of the hitch point between the tractor and trailer. The y-component of $\vec{v_h}$ is used to determine v_{h2y}, which is needed to determine the differential equation for $\dot{\psi}_2$.

The radius, R, is also known because of the Ackermann steering angle relationship found in equation 3.3. This can be then combined with the previous equation to create the following:

$$v_{h1y} = \frac{v_{h1x}h}{L_1}\tan\delta \tag{3.10}$$

Finally, v_{h1x} is actually known because it is the same as velocity v_{1x}. This is due to the hitch point being in line with the longitudinal axis of the tractor, whereas the lateral velocity of the hitch point is induced to some rotation.

$$v_{h1y} = \frac{v_{1x}h}{L_1}\tan\delta \tag{3.11}$$

The lateral velocity of the hitch point is needed to determine the lateral velocity of the trailer hitch point because v_{h2y} is a result of what the tractor does. Determining the longitudinal and lateral velocities at the trailer hitch point is determined through an active rotation by the hitch angle. In other words, it is the velocity of the tractor plus the relative motion of the hitch point. The hitch angle is defined as $\theta = \psi_1 - \psi_2$.

$$v_{h2x} = v_{1x}\cos\theta - v_{h1x}\sin\theta \tag{3.12}$$
$$v_{h2y} = v_{1x}\sin\theta + v_{h1x}\cos\theta \tag{3.13}$$

Due to the tractor yaw motion centering about the rear axle, the hitch point will have a lateral velocity in the opposite direction than that of v_{y1}.

$$v_{h1y} \rightarrow (-)\frac{v_{x1}h}{L_1}\tan\delta \tag{3.14}$$

Substituting equation 3.14 into equations 3.12 and 3.13 results in the following:

$$v_{h2x} = v_{1x}\cos\theta + \frac{v_{1x}h\tan\delta}{L_1}\sin\theta \tag{3.15}$$

$$v_{h2y} = v_{1x}\sin\theta - \frac{v_{1x}h\tan\delta}{L_1}\cos\theta \tag{3.16}$$

Recall the differential equation for trailer yaw rate was missing v_{h2y}. It is now substituted into equation 3.6, which was $\dot{\psi_2} = \frac{v_{h2y}}{L_2}$.

$$\dot{\psi_2} = \frac{v_{1x}}{L_2}\sin\theta - \frac{v_{1x}h}{L_1L_2}\tan\delta\cos\theta \tag{3.17}$$

The longitudinal velocity of the trailer is the same as v_{h2x} because the tractor pulls the trailer. Similarly as before, the velocity of the trailers can now be described as:

$$\dot{x_2} = v_{2x}\cos\psi_2 \tag{3.18}$$

$$\dot{y_2} = v_{2x}\sin\psi_2 \tag{3.19}$$

For clarity, the nonlinear differential equations that were derived are stated below:

$$\begin{aligned}
\dot{x_1} &= v_{1x}\cos\psi_1 \\
\dot{y_1} &= v_{1x}\sin\psi_1 \\
\dot{\psi_1} &= \frac{v_{1x}}{L_1}\tan\delta \\
\dot{x_2} &= v_{2x}\cos\psi_2 \\
\dot{y_2} &= v_{2x}\sin\psi_2 \\
\dot{\psi_2} &= \frac{v_{1x}}{L_2}\sin\theta - \frac{v_{1x}h}{L_1L_2}\tan\delta\cos\theta
\end{aligned} \tag{3.20}$$

The following relationships are useful: the velocity of the trailer and the hitch angle.

$$\begin{aligned}
v_{2x} &= v_{1x}\cos\theta + \frac{v_{1x}h\tan\delta}{L_1}\sin\theta \\
\theta &= \psi_1 - \psi_2
\end{aligned} \tag{3.21}$$

The nonlinear equations should be used for simulation, however, the equations need to be linearized in order to be put into standard state space representation for the LQR. Linearization can be accomplished using the Jacobian about an equilibrium point, also known as an operating point. This can be thought of as taking a Taylor expansion series and only using the first term. Small angle approximations are effectively Jacobian linearizations around the equilibrium points of $\underline{x} = \underline{0}$.

The nonlinear differential equations are generally expressed as $\dot{\underline{x}}(t) = f(\underline{x}(t), \underline{u}(t))$. The approximated differential equations work as long as the deviation from the operating point remain small. The system matrix, A and B, can be determined by taking the Jacobian and substituting the equilibrium points, $\bar{\underline{x}}$ and $\bar{\underline{u}}$, in.

$$\dot{\underline{x}}(t) \approx \left.\frac{\partial f}{\partial \underline{x}}\right|_{\substack{\underline{x}=\bar{\underline{x}} \\ \underline{u}=\bar{\underline{u}}}} (\underline{x}(t) - \bar{\underline{x}}) \quad + \quad \left.\frac{\partial f}{\partial \underline{u}}\right|_{\substack{\underline{x}=\bar{\underline{x}} \\ \underline{u}=\bar{\underline{u}}}} (\underline{u}(t) - \bar{\underline{u}}) \tag{3.22}$$

$$A \approx \left.\frac{\partial f}{\partial \underline{x}}\right|_{\substack{\underline{x}=\bar{\underline{x}} \\ \underline{u}=\bar{\underline{u}}}} \quad , \quad B \approx \left.\frac{\partial f}{\partial \underline{u}}\right|_{\substack{\underline{x}=\bar{\underline{x}} \\ \underline{u}=\bar{\underline{u}}}} \tag{3.23}$$

Considering the objective is to get the trailer to the loading dock without jackknifing, the following equations are investigated for linearization. The longitudinal velocity of the trailer, v_{2x}, is substituted in.

$$\begin{aligned}
\dot{\psi_1} &= \frac{v_{1x}}{L_1} \tan\delta \\
\dot{\psi_2} &= \frac{v_{1x}}{L_2} \sin\theta - \frac{v_{1x} h}{L_1 L_2} \tan\delta \cos\theta \\
\dot{x_2} &= (v_{1x} \cos\theta + \frac{v_{1x} h \tan\delta}{L_1} \sin\theta) \cos\psi_2 \\
\dot{y_2} &= (v_{1x} \cos\theta + \frac{v_{1x} h \tan\delta}{L_1} \sin\theta) \sin\psi_2
\end{aligned} \tag{3.24}$$

The equilibrium points are determined by setting $f(\underline{x}(t), \underline{u}(t)) = 0$ and solving for the states and input. Keeping the hitch angle in exploits information about ψ_1 and ψ_2. This determines that the equilibrium points require that they are equal to each other.

$$\begin{aligned}
\dot{\psi_1} = 0 &= \frac{v_{1x}}{L_1} \tan\delta \quad \rightarrow \quad \bar{\delta} = 0,\, n\pi \quad \text{for} \quad n = 0, 1, 2... \\
\dot{\psi_2} = 0 &= \frac{v_{1x}}{L_2} \sin\theta - \frac{v_{1x} h}{L_1 L_2} \tan\delta \cos\theta \quad \rightarrow \quad \bar{\theta} = 0,\, n\pi \\
\theta &= \psi_1 - \psi_2 \\
\psi_1 - \psi_2 = 0 \quad &\rightarrow \quad \psi_1 = \psi_2 \\
\dot{y_2} = 0 &= (v_{1x} \cos\theta + \frac{v_{1x} h \tan\delta}{L_1} \sin\theta) \sin\psi_2 \quad \rightarrow \quad \bar{\psi_2} = 0,\, n\pi \\
\therefore \quad \bar{\psi_1} &= 0,\, n\pi \\
\bar{y_2} &= 0
\end{aligned} \tag{3.25}$$

The differential equation for longitudinal velocity, however, results in a different equilibrium point as the former equations. The $\cos\psi_2$ term thus requires $\bar{\psi_2} = \bar{\psi_1} = \frac{(2n-1)\pi}{2}$ for $n = 0, 1, 2...$etc. The objective, therefore, is then to create a state space representation which includes lateral motion, but

not longitudinal. The chosen states are thus:

$$\underline{x} = \begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} \tag{3.26}$$

The Jacobian for ψ_1 can be found by taking the partials with respect to each of the different states and actions. The following equations are found after plugging in the hitch angle.

$$\begin{aligned} \frac{\partial \psi_1}{\partial \underline{x}} &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \\ \frac{\partial \psi_1}{\partial \underline{u}} &= \begin{bmatrix} \frac{v_{1x}}{L_1} \frac{1}{\cos^2 \delta} \end{bmatrix} \end{aligned} \tag{3.27}$$

Similarly, the Jacobian with respect to the states can be found for ψ_2:

$$\begin{aligned} \frac{\partial \psi_2}{\partial \psi_1} &= \frac{v_{1x}}{L_2} \cos(\psi_1 - \psi_2) + \frac{v_{1x} h}{L_1 L_2} \tan\delta \sin(\psi_1 - \psi_2) \\ \frac{\partial \psi_2}{\partial \psi_2} &= -\frac{v_{1x}}{L_2} \cos(\psi_1 - \psi_2) + \frac{v_{1x} h}{L_1 L_2} \tan\delta \sin(\psi_1 - \psi_2) \\ \frac{\partial \psi_2}{\partial y_2} &= 0 \end{aligned} \tag{3.28}$$

The Jacobian of ψ_2 with respect to the action is as follows:

$$\frac{\partial \psi_2}{\partial \delta} = -\frac{v_{1x} h}{L_1 L_2} \cos(\psi_1 - \psi_2) \frac{1}{\cos^2 \delta} \tag{3.29}$$

The Jacobian of y_2 with respect to the states are:

$$\begin{aligned} \frac{\partial y_2}{\partial \psi_1} &= -v_{1x} \sin(\psi_1 - \psi_2) \sin\psi_2 + \frac{v_{1x} h}{L_1} \tan\delta \cos(\psi_1 - \psi_2) \sin\psi_2 \\ \frac{\partial y_2}{\partial \psi_2} &= -v_{1x} \cos(\psi_1 - \psi_2) \cos\psi_2 - v_{1x} \sin\psi_2 \sin(\psi_1 - \psi_2) \\ &\quad + \frac{v_{1x} h}{L_1} \tan\delta \sin(\psi_1 - \psi_2) \cos\psi_2 \\ &\quad + \frac{v_{1x} h}{L_1} \tan\delta \cos(\psi_1 - \psi_2) \sin\psi_2 \\ \frac{\partial y_2}{\partial y_2} &= 0 \end{aligned} \tag{3.30}$$

Lastly, the Jacobian of y_2 with respect to the input results in:

$$\frac{\partial y_2}{\partial \delta} = \frac{v_{1x} h}{L_1} \sin(\psi_1 - \psi_2) \sin\psi_2 \tag{3.31}$$

The Jacobian can now have the operating point plugged into the matrix to result in the system matrix, A, and input matrix, B.

$$A \approx \left.\frac{\partial f}{\partial \underline{x}}\right|_{\substack{\psi_1=0 \\ \psi_2=0 \\ y_2=0 \\ \delta=0}} = \begin{bmatrix} 0 & 0 & 0 \\ \frac{v_{1x}}{L_2} & -\frac{v_{1x}}{L_2} & 0 \\ 0 & v_{1x} & 0 \end{bmatrix}$$
$$B \approx \left.\frac{\partial f}{\partial \underline{u}}\right|_{\substack{\psi_1=0 \\ \psi_2=0 \\ y_2=0 \\ \delta=0}} = \begin{bmatrix} \frac{v_{1x}}{L_1} \\ -\frac{v_{1x}h}{L_1 L_2} \\ 0 \end{bmatrix} \tag{3.32}$$

To summarize, the relevant linearized equations are gathered below:

$$\begin{aligned} \dot{\psi_1} &= \frac{v_{1x}}{L_1}\delta \\ \dot{\psi_2} &= \frac{v_{1x}}{L_2}\psi_1 - \frac{v_{1x}}{L_2}\psi_2 - \frac{v_{x1}h}{L_1 L_2}\delta \\ \dot{y_2} &= v_{1x}\psi_2 \end{aligned} \tag{3.33}$$

The state space representation is therefore:

$$\begin{aligned} \dot{\underline{x}} &= A\underline{x} + B\underline{u} \\ \underline{y} &= C\underline{x} + D\underline{u} \end{aligned}$$

$$\begin{bmatrix} \dot{\psi_1} \\ \dot{\psi_2} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ \frac{v_{1x}}{L_2} & -\frac{v_{1x}}{L_2} & 0 \\ 0 & v_{1x} & 0 \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} + \begin{bmatrix} \frac{v_{1x}}{L_1} \\ -\frac{v_{1x}h}{L_1 L_2} \\ 0 \end{bmatrix} \delta \tag{3.34}$$

$$\begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \delta \tag{3.35}$$

This section derived the nonlinear differential equations for a kinematic model of the tractor-trailer. The equations were derived with constant velocity going forward, so driving reverse is such that the velocity is negative. The model assumes an off-axis hitch, which is a numerical number that can be adjusted. Setting the hitch length to zero produces an on-axle hitch. Using a negative hitch length creates a vehicle that is more like a tractor with a semi-trailer. The model was derived as an off-axis hitch behind the rear axle due to its visibility of the parameters in the diagram. The equations were then linearized using the Jacobian evaluated at an operating point. The linearized model is a good approximation as long as the states do not stray far from the equilibrium point.

3.2 Open Loop System Eigenvalues

The nominal parameters of the tractor-trailer system are chosen in the following Table 3.1. The majority of the States in America have regulated the maximum length from the kingpin to the rear axles of semi-trailers to be 40 feet or 12.192 m [70]. The nominal parameters are chosen such that they reside in the median of realistic values.

Table 3.1: The nominal system parameters are chosen such that the trailer parameters reside in the middle of the varied parameters.

Parameter	L_1	L_2	h	v_{1x}
-	5.74m	10.192m	0.0m	-2.012 $\frac{m}{s}$

First considering the combination vehicle going forwards, a positive velocity of $2.012\frac{m}{s}$ creates an marginally stable system because the eigenvalues produced have a single negative value and the other two are at zero. The system will remain stable until the velocity exceeds some critical velocity. In other words, the trailer will track the tractor until the vehicle is moving too fast that speed wobble may occur with any slight disturbance. The eigenvalues are calculated and plotted in the pole zero map in Figure 3.3.

$$\lambda_1 = 0 \quad , \quad \lambda_2 = -0.1974 \quad , \quad \lambda_3 = 0 \tag{3.36}$$

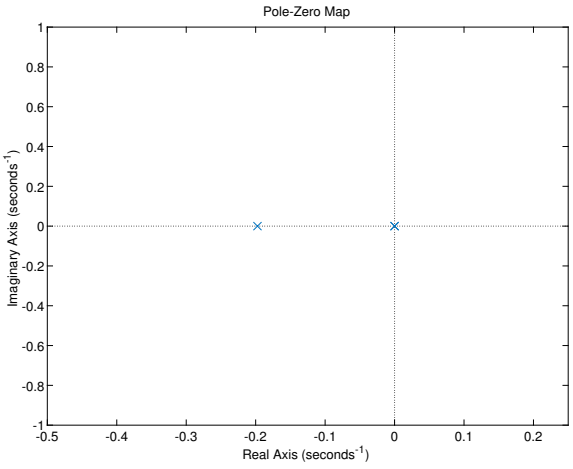


Figure 3.3: Due to the pole for ψ_2 being negative and the other two poles being at zero, the system can be considered asymptotically stable.

When the vehicle reverses, however, the system becomes unstable due to the positive eigenvalue of ψ_2. Fortunately this can be mitigated by closing the loop of the system with a controller. The

eigenvalues are calculated and plotted again in Figure 3.4.

$$\lambda_1 = 0 \quad , \quad \lambda_2 = 0.1974 \quad , \quad \lambda_3 = 0 \tag{3.37}$$

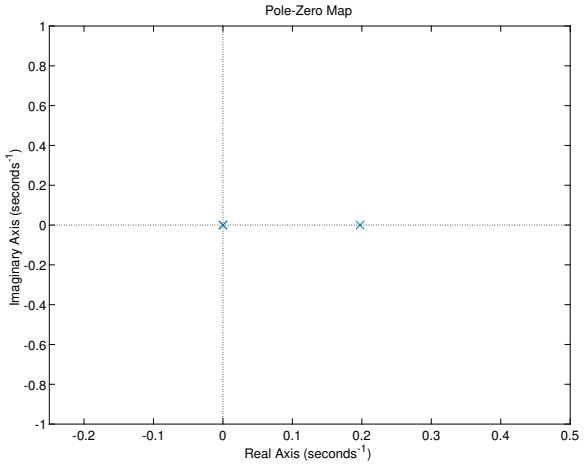


Figure 3.4: Due to the pole for ψ_2 being positive, the system is considered unstable.

This section discussed the stability of the system when driving forward and in reverse using the linearized system equations. The non-linear equations should, however, be used in simulation.

3.3 Kinematic Model in MATLAB/Simulink

Modeling the mathematical equations for the kinematic tractor-trailer began in MATLAB/Simulink 2018a. The simulation consisted of a single '*.m*' file and a single '*.slx*' file. The files were named *trailerKinematicsLQR.m* and *LQRTrailerKinematics.slx*. The '*.m*' file is where the nominal parameters were declared along with the computation of the matrices A, B, C, D for the state space representation of the linearized equations. In addition, this file declares variables in the workspace such that the '*.slx*' file can read them. Some examples of this are the initial conditions and maximum steering angle.

The maximum steering angle is declared as $\pm 45°$ in the workspace and then a block in Simulink acts as a saturator for the input. Simulink provides a powerful graphical programming language that looks and operates similar to block diagrams. It includes many signals and logic gates. Behind the scenes, Simulink uses numerical solvers along with the defined loops in order simulate the tractor-trailer model. The '*.m*' file starts the Simulink simulation. After every timestep, information about the simulation is logged and sent back to the workspace. Once the simulation is over, the '*.m*' file

continues to run further commands. It generates plots and produces an animation of the tractor-trailer through the global Cartesian coordinate system.

The nonlinear equations for the combination vehicle are programmed in Simulink. It is placed into a subsystem to clarify that the inputs are steering angle and the outputs are the states and other variables needed for the animation. The reason Simulink blocks are used for the nonlinear equations are because of the ability to easily extract the integrals of terms with the integration block. The subsystem containing the nonlinear equations is shown by Figure 3.5. The implemented nonlinear equations are shown in Simulink in Figure 3.6.

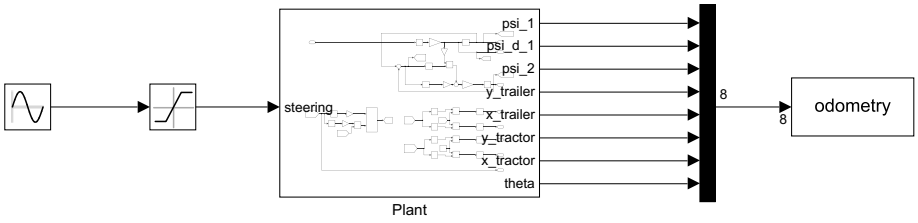


Figure 3.5: The subsystem shows the inputs are the steering angle and the outputs are whatever is necessary to output to the workspace.

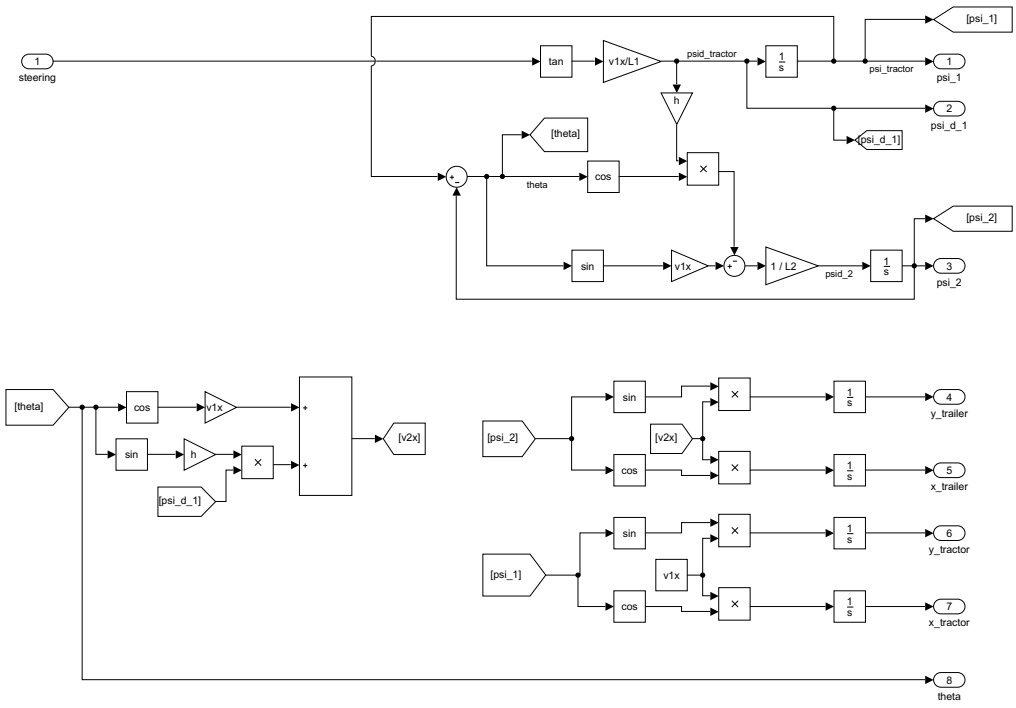


Figure 3.6: The nonlinear equations are implemented in Simulink. The goto blocks are used to connect variables together without a line.

This section talked about implementing the nonlinear equations into MATLAB/Simulink and some useful tips for programming. There are a multitude of numerical equation solvers at the disposal of the user. Originally, a variable time step solver was used, but was found later that a fixed time step is needed.

3.4 Fixed Time Step

It was discovered that fixed timesteps were needed because the distance to goal checking did not have enough resolution in steps with ode45, a variable timestep solver with a relative tolerance of $1e-9$. An example array of distance in meters at each time step near the goal is as follows:

$$\text{distance} = [39.13796, \quad 12.1376, \quad 14.8624, \quad 41.8624, \quad 68.86424]$$

Between 12.1376 and 14.8624m should have been a number approximately zero. This simulation was done with a constant velocity going forward and no steering input, meaning the combination vehicle should have definitely crossed the goal. Figure 3.7 shows the tractor-trailer continuing to drive and the simulation failing to report the goal criteria was met.

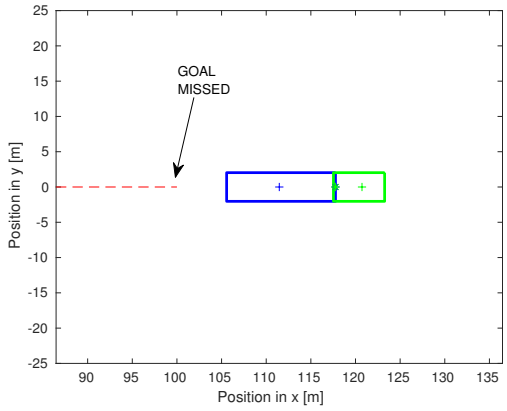


Figure 3.7: The resolution of the sensor (solver) was not great enough to achieve the distance requirement. This simulation was done using MATLAB's ode45 variable step solver.

There are benefits to using variable step solvers such as faster computation and decreasing the step size when the model states are changing rapidly. A smaller step size will increase the accuracy. The problem, however, is that variable step solvers also increase the step size when the states are changing slowly. Fixed step solvers, on the other hand, compute in a manner that is useful for determining the goal criteria. The fixed step solver chosen was ode4, the Fourth Order Runge Kutta. The downside is that the step size needs to be wisely chosen, otherwise computation will

take too long. Determining the step size will be discussed after the implementation of the kinematic tractor-trailer model in Python.

3.5 Kinematic Model in Python

MATLAB was discussed in the previous sections because it was originally conceived that the comparison of controller performances could be accomplished keeping the LQR in MATLAB and the DDPG in Python. The kinematic model was implemented in Python3.5 because it is the native language to machine learning. Python allows people to import open source modules or packages that help with certain tasks. Some examples of these are *Tensorflow*, which is used for machine learning, and *matplotlib*, which allows for plots and animations similar to MATLAB. Another useful module is *numpy*, which makes Python feel similar to MATLAB.

The kinematic model was implemented in a fashion such that fellow researchers could try different reinforcement learning methods in the same environment. OpenAI gym is a format of environments that is found in many toy reinforcement learning examples such as MountainCar, CartPole, Atari games, and MuJoCo humanoid [71]. The basic functionality of the environments consist of methods for reset(), step(), and render() in an episodic manner. The physics solvers, however, are not unique to OpenAI gym. For example, MuJoCo, is a physics solver which requires an additional license. OpenAI gym offers the python module, *gym*, to quickly import these environments. Custom environments are simply programmed to fit the framework.

The environment primarily consisted of two files, which were defined as classes. The first file, *truck_backerupper_env.py*, contained the methods for reset(), step(), and render(). The kinematic model is also found in this file. The numerical solver was thus selected to also be the fixed step solver, Runge Kutta method of order 4(5). The *scipy.integrate* module was imported so the *ode*() method could be used.

Scipy requires the ordinary differential equations to be placed into a function so the equations can be integrated iteratively. Scipy's Application Program Interface (API) is similar to MATLAB's numerical solvers, but the older API requires manual stepping through time. Newer versions of *scipy.integrate*, such as versions $> 1.1.0$, have *solve_ivp*() which automatically steps through time. It, however, lacks the ability to easily change the input to the differential equations. In other words, there is no convenient way to manually change the variable values in the differential equation. Thus, the older API, *ode*() was used. The input, u, is changed using the $set\_f\_params(u)$ method. The code snippet below demonstrates the gist of what methods are needed for running the simulation.

```
    import numpy as np
    import matplotlib.pyplot as plt
    import scipy.integrate as spi
    import gym

    class TruckBackerUpperEnv(gym.Env):
        def __init__(self):
            ''' '''
        def reset (self):
            self.solver = spi.ode(self.kinematic_model).set_integrator('dopri5')
            self.solver.set_initial_value(self.ICs, self.t0)
            self.solver.set_f_params(self.u)
            self.s = self.get_error(0)
            return self.s

        def step(self, u):
            self.solver.set_f_params(a)
            self.solver.integrate(self.solver.t + self.dt)
            self.s = self.get_error(self.sim_i)
            self.sim_i += 1
            return self.s, r, done, info

        def render(self):
            plt.pause(eps)
```

The nonlinear kinematic model equations are defined within the class of $TruckBackerUpperEnv()$. The input, u, is changed at every timestep which is generated by the chosen control method such as the LQR or DDPG. If one had the gains designed already, then multiplying the error vector by the gain matrix will produce the input action. If the neural network was trained already, the action is simply the output of the policy network. This section, however, focuses on open loop responses.

```
        def kinematic_model(self, t, x, u):
            self.u = u
            n = len(x)
            xd = np.zeros((n))
            xd[0] = (self.v1x / self.L1) * np.tan(self.u)
            self.theta = x[0] - x[1]

            xd[1] = (self.v1x / self.L2) * np.sin(self.theta) -\
                    (self.h / self.L2) * xd[0] * np.cos(self.theta)

            self.v2x = self.v1x * np.cos(self.theta) + self.h * xd[0] *\
                       np.sin(self.theta)

            xd[2] = self.v1x * np.cos(x[0])
            xd[3] = self.v1x * np.sin(x[0])
            xd[4] = self.v2x * np.cos(x[1])
            xd[5] = self.v2x * np.sin(x[1])
            return x
```

The second file, called *PathPlanner.py*, is a class which contains various helpful methods for the kinematic model to interact with the path. The UML class diagram for the simulation environment in Python is provided in Figure 3.8. This will be discussed in more detail in the next chapter. Now that the equations have been implemented in both Python and MATLAB, the fixed time step for the numerical solvers need to be determined.

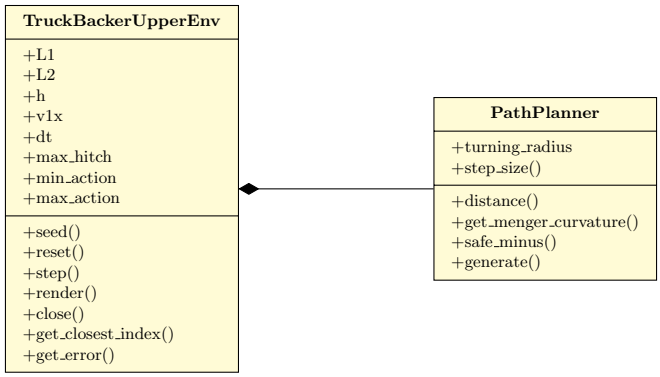


Figure 3.8: The class diagram for the simulation environment in Python shows how the TruckBackerUpperEnv class has a relationship called composition with the PathPlanner class, denoted by the black diamond.

3.6 Determining the Fixed Timestep

Having too large of a fixed timestep speeds up computation, but results in a less accurate simulation. For the purpose of the trailer reversal simulation, it was noticed the trailer would miss the goal due to a lack of resolution in the distance check array. Having too small a fixed timestep produces accurate results, but immensely slows down the computations. This results in training taking substantially longer.

Another point of concern is having too large of a fixed timestep such that the controller performance degrades. The timestep depends on the speed of the dynamics of the system, i.e. the eigenvalues. Initial training on DDPG began with $10ms$ as the timestep and a substantial amount of time was wasted. It is thus suggested to perform a convergence test of the timesteps in simulation.

As it turns out, the simulation could be sped up to a fixed timestep of $80ms$. Once the controller gains were determined, the LQR was used to control the system along 100 paths. A Python script was written to loop through various timesteps on the same 100 tracks, decreasing the timestep until the root mean squared errors were within five percent of the results from a timestep of $1ms$. Eight different timesteps were evaluated using the LQR, each over the same 100 paths. This was run over the course of two days, but it was worth the investigation. Figure 3.9 demonstrates the convergence test assuming $dt = .001$ gives the correct answer.

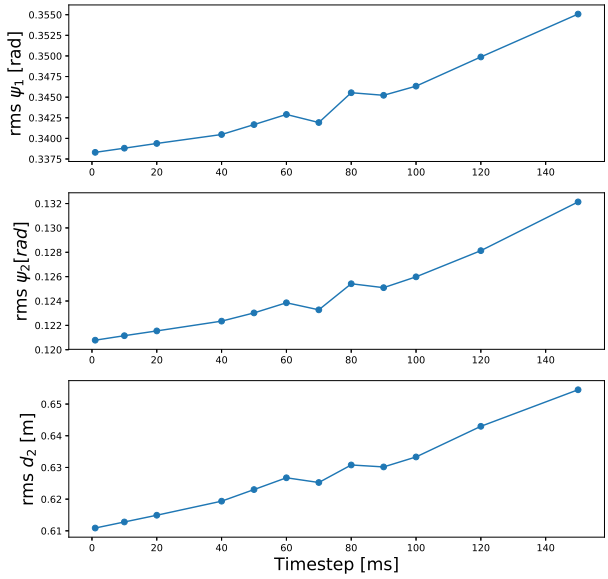


Figure 3.9: The timesteps were decreased until all three root mean squared errors from the path were within 5% of $dt = .001$. A larger timestep does degrade performance.

Another quality that was monitored was whether or not the timestep produced a result that made it to the goal as many times as the timestep of $1ms$. The goal criteria is for the rear of the trailer to be within $15cm$ of the loading dock and the angle of the trailer to be within $0.1\,rad$. Too large of a timestep, would mean that the change from one timestep to the next could not measure the resolution that is necessary. The number of goals reached from the convergence test are shown in Table 3.2.

Table 3.2: The fixed timesteps were evaluated over the same 100 tracks, whereas convergence also required the same number of goals were achieved as $1ms$.

dt [s]	0.15	0.12	0.1	0.09	0.08	0.07	0.06	0.05	0.04	0.02	0.01	0.001
goals	74	77	77	78	80	80	81	81	80	80	80	80

It is interesting to note that at a timestep of 60 and $50ms$ resulted in goal increase of one more than the timestep of $1ms$. The calculation of the distance checker depends on the resolution of the value provided at that instant, so it is not necessarily better to choose these timesteps. In other words, it may be better to actually interpolate the values regardless. A similar convergence test is performed in MATLAB to confirm that $80ms$ was an appropriate fixed timestep, but the

results varied slightly. The difference in numerical solvers will be discussed after the validation of the kinematic models using commercial software called IPG CarMaker.

3.7 IPG CarMaker Model

This section reviews a comparison of the single track kinematic model, single track kinetic model, and the multi-body dynamic IPG model. For more information regarding the kinetic model, see appendix A.

Initially realistic tractor-trailer parameters were used to create a model in IPG CarMaker, i.e. masses and tire cornering stiffnesses. It was quickly discovered that the default tire models provided by CarMaker were not comparable to R22.5 tires found on tractor-trailers. To get a sense of the normal loads on heavy duty vehicles, the additional mass of the trailer on the rear axle of the tractor can be found as $m_2 \frac{b_2}{l_2}$ [33]. The normal loads on the tractor can be found on Figure 3.10. After summing the forces using the free body diagram, the normal loads on each of the axles can be determined.

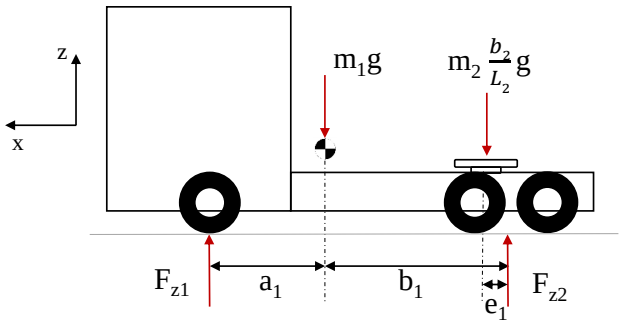


Figure 3.10: The mass of the trailer increases the normal loads on the rear axle of the tractor.

$$
\begin{aligned}
F_{z1} &= m_1 g \frac{b_1}{l_1} - m_2 g \frac{b_2}{l_2} \frac{e_1}{l_1} \\
F_{z2} &= m_1 g \frac{a_1}{l_1} + m_2 g \frac{b_2}{l_2} \frac{a_1 + h_1}{l_1} \\
F_{z3} &= m_2 g \frac{a_2}{l_2}
\end{aligned}
\tag{3.38}
$$

Using Barbosa's [55] parameters, the normal loads per tire assuming the correct number of tires per axle become:

$$\frac{F_{z1}}{2} = 30011.832\,N$$
$$\frac{F_{z2}}{8} = 19789.688\,N$$
$$\frac{F_{z3}}{8} = 24551.978\,N$$

Then looking at the linear region of some Cal Span Tire data [72], the tire cornering stiffnesses can be determined. The data only unfortunately has 4 data points as shown in Figure 3.11. The data consists of an excel file containing information for 24 different tire manufacturers. It reports the resultant cornering stiffness of the tire at various loads. If one were to interpolate between these four data points for the normal loads calculated above, the cornering stiffness can be estimated. Choosing Yokohama 11R22.5 tires as an example, the tire cornering stiffnesses are as follows:

$$C_1 = 156154\frac{N}{rad}$$
$$C_2 = 112580\frac{N}{rad}$$
$$C_3 = 134370\frac{N}{rad}$$

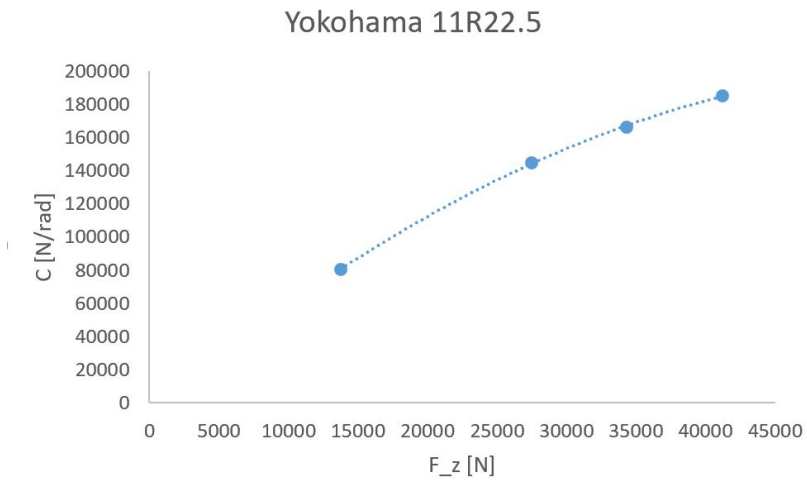


Figure 3.11: Given the four data points from CalSpan, one can interpolate what the cornering stiffness of the tire would be at certain normal loads.

When looking at the '*.tir*' files available from IPG CarMaker, it was apparent that these tires could not handle the normal load of a tractor-trailer. The only tires remotely possible were the F1 tires, but those are racing slicks. A paper by Zhang [73] used IPG Carmaker as parameter verification of the cornering stiffness of tires of a car-trailer combination. Most the parameters are

used for this validation, but were modified such that the rear axle of the trailer is lumped into one tire. Refer to Appendix A for an understanding of the additional variable names for a kinetic model.

$$m_1 = 1955\,kg \qquad m_2 = 1880\,kg$$
$$J_1 = 2690\,kg \cdot m^2 \qquad J_2 = 10350\,kg \cdot m^2$$
$$a_1 = 1.341\,m \qquad a_2 = 3.983\,m$$
$$b_1 = 1.344\,m \qquad L_2 = 5.274\,m$$
$$L_1 = 2.685\,m \qquad b_2 = 1.291\,m$$
$$h_1 = 2.127\,m$$
$$C_1 = 2 \times 110000\,\frac{N}{rad} \qquad C_3 = 4 \times 125000\,\frac{N}{rad}$$
$$C_2 = 2 \times 185000\,\frac{N}{rad}$$

There are a massive amount of parameters that can be changed in IPG CarMaker, arguably too many. In addition to the above parameters, the steering rack ratio had to either be modified in IPG or in the single track models. The default steering rack ratio is $100\frac{rad}{m}$, which must be based off the suspension kinematics–i.e. the steering arm and rack. The steering rack ratio of IPG was modified to $6.75\frac{rad}{m}$. Despite not knowing what the steering rack length is, the 1:1 ratio can be determined by understanding steering with slip angles. For a steady state turn of radius $60m$, the steering angle can be found by:

$$\delta = \tan^{-1}(\frac{L}{R}) + (\alpha_f - \alpha_r) \tag{3.39}$$

IPG CarMaker was set up to drive a steady state radius and told to follow the path. The slip angles and the actual steering was logged in simulation. The steering ratio was modified until the recorded steering output matched that of the theoretical using the logged steering angles. Thus, with a steering ratio of $6.75\,\frac{rad}{m}$, a $1:1$ steering was implemented like the single track equations.

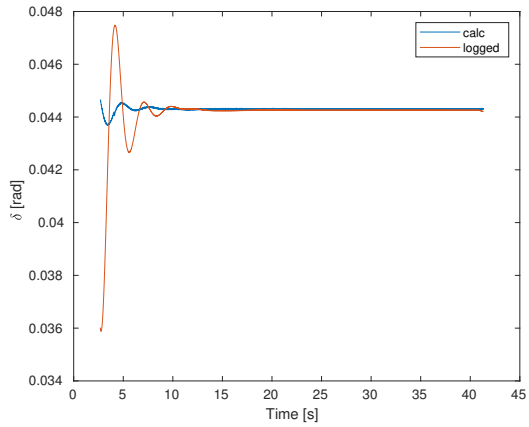


Figure 3.12: The steering ratio was modified to make the theoretical match the logged.

The kinematic and kinetic models assume constant longitudinal velocity, however, it should be noted that IPG CarMaker doesn't exactly do this. Even though a constant velocity can be set beforehand and during, the longitudinal velocity varies due to acceleration.

The starting point of the vehicle in the IPG environment is a source of confusion. If the desired track starts at position $(x, y) = (0, 0)$, it will start at the origin point of how one builds the vehicle. One can place sensors in various locations, so the obvious location is about the rear axle. This is due to the desire to control the rear axle of the trailer. The kinetic and the IPG CarMaker models rotate with respect to the center of gravity, but the kinematic model rotates with respect to the rear axle. Care must be given to ensure the IPG CarMaker simulation starts at the rear axle.

There are options to have an IPG Driver and Follow Course. These, however, are essentially controllers. The IPG Driver will cut corners for the racing line. In addition to this, even the IPG Driver would only follow the course with the car, thus jackknifing the trailer. The objective is to see how well the equations operate under open loop, not the controllers. Thus, a sinusoid input was used.

The sinusoid was generated from IPG CarMaker, then also fed into the kinematic and kinetic models via a '*.csv*' file. It is also possible to do this with Simulink, however, the license was not provided for the IPG/Simulink communication. The comparison is performed with MATLAB/Simulink because this work started in MATLAB.

Since the kinetic model rotates about the rear axle, this is the location that is logged. There is a bias between models because of the offset from the reference points between the kinetic and kinematic models, i.e. the center of gravity and rear axles. The location of the rear axles can be found by rotating the bias by the direction cosine matrix and adding it to the global coordinates of the center of gravity.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} + \begin{bmatrix} \cos\psi_2 & -sin\psi_2 & 0 \\ \sin\psi_2 & cos\psi_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -b_2 \\ 0 \\ 1 \end{bmatrix} \tag{3.40}$$

The kinematic and kinetic single track models are validated with the IPG Carmaker model using the same steering input as shown in Figure 3.13. The input resembles a quick lane change maneuver.

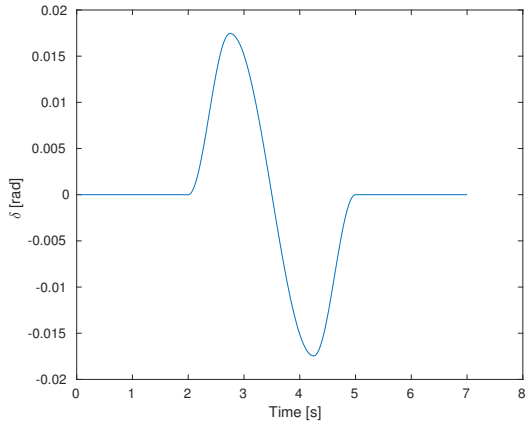


Figure 3.13: The sinewave has an amplitude of 1 degree with a period of 3s. The sinewave starts at 2s.

At a velocity of $4.5\frac{m}{s}$ driving forward, the three models are compared. The position of the trailers match identically for the kinetic and kinematic model, but vary $10.8mm$ from the IPG model in Figure 3.14. The following two figures are zoomed in on the response at the very end of the simulation. The plots represent the positions traveled in meters in global Cartesian coordinates.

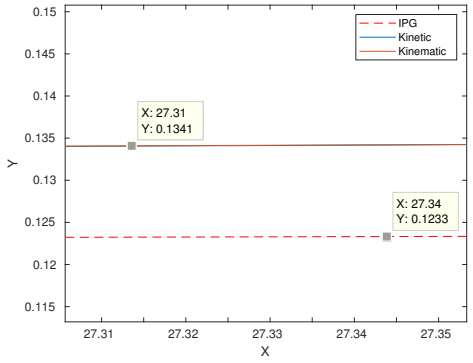


Figure 3.14: The lane change maneuver for the trailer varies by 10.8mm to the single track models.

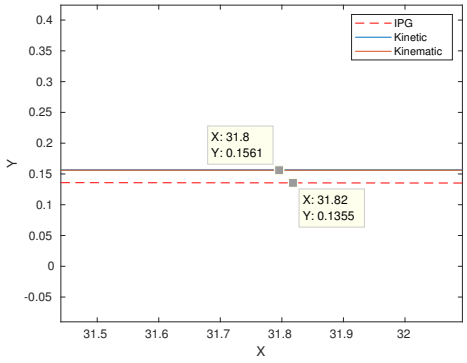


Figure 3.15: The lane change maneuver for the tractor varies by 20.6mm to the single track models.

For the same forward lane change maneuver, the positions of the tractor and trailer are nearly identical once again in Figure 3.15. They both deviate from the IPG model again, but this time by $20.6mm$. The headings of the tractor and trailer seem to be overestimated by both bicycle models compared to the IPG model, which can be seen in Figures 3.16 and 3.17.

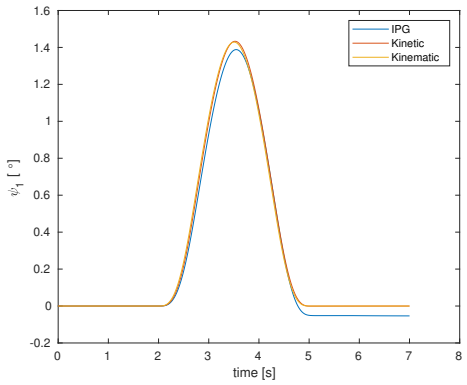


Figure 3.16: The heading of the tractor seem to line up fairly well, but both the kinetic and kinematic model overestimate compared to the IPG model.

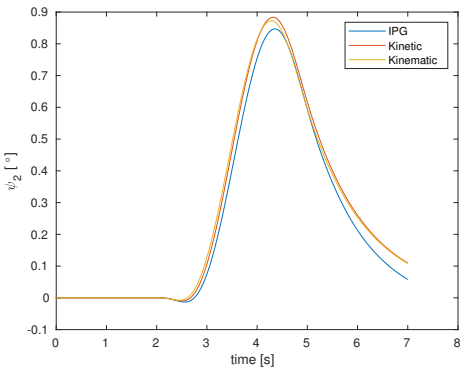


Figure 3.17: The kinetic and kinematic models also seem to overestimate ψ_2.

When evaluating the hitch angle for all three models, it was discovered the kinetic model's hitch angle appeared to be out of phase. It turns out the reason is simply because the hitch angle, θ, is defined differently as $\psi_2 - \psi_1$ instead of $\psi_1 - \psi_2$. Correcting this definition as a post processing method shows the trends are in fact similar as shown in Figure 3.18.

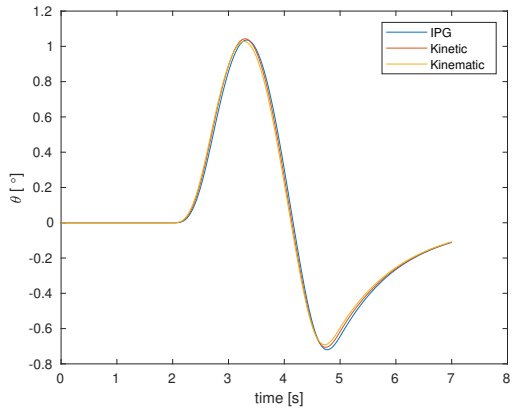


Figure 3.18: The kinematic and kinetic single track models overestimate the hitch angle due to overestimating the heading angles.

.

As the vehicle speed of the simulation is increased, the bicycle model of the kinetic tractor-trailer (Luijten model) [33] better tracks what IPG CarMaker does, but not significantly enough to show. At faster speeds, the kinematic model appears to drive flawlessly whereas both the Luijten model and the IPG Carmaker model produce varying results due to forces from the tires.

Some reasons for the difference in responses of the Luijten and the IPG model mainly have to do with how the tires are modeled. The Luijten model is a bicycle model where the tires on the axles are lumped into one tire, where a constant cornering stiffness is assumed. IPG CarMaker, however, simulates the effects of lateral load transfer and uses a nonlinear tire model for varying cornering stiffnesses.

3.8 Kinematic and IPG Reverse Validation

Setting the speed to $-4.5\frac{m}{s}$, the jackknifing of the kinematic and IPG model are compared. The same sinusoid input as before was put in to look at the jackknifing. Figure 3.19 shows that both positions of the rear axle respond similarly. Starting at $100m$, the trailer is reversed until approximately $70m$ where the simulation ends due to the simulation ending after $7s$.

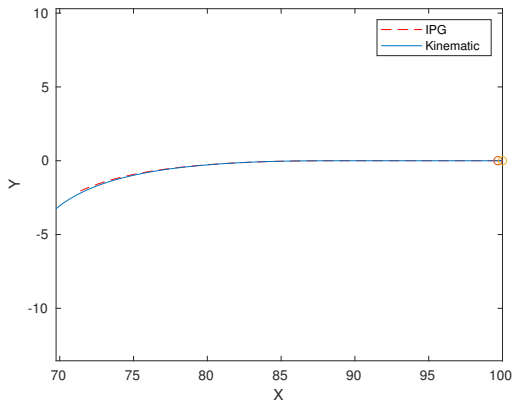


Figure 3.19: The zoomed in position of the tractor is straight, but with a sinusoid input, it jackknifes.

Figure 3.20 shows that the kinematic model overestimates the heading of the tractor by about $0.1°$, but nonetheless tracks pretty well until the end. The IPG model appears to have additional dynamics which cause the heading of the trailer to move from the straight line. It is suspected that this is due to bump steer. Bump steer is when the wheels unintentionally steer themselves when the suspension is induced to heave motion. This is likely due to the trailer hitch load increasing, pushing the rear of the car up. In other words, this raises the suspension up in the rear.

Figure 3.21 shows the heading of the trailer is overestimated by the kinematic model. The hitch angle appears to deviate by $16°$ in Figure 3.22. Both of the simulations do not necessarily end at a jackknife because the simulation ended, but the hitch angle would continue to increase until $90°$ is reached. The trailers do not straighten out because the tractor no longer adjusts the orientation to

correct the trailer heading. In fact, since the kinematic tractor straightens out and the IPG model does not, it makes sense the simulated hitch angle results deviate.

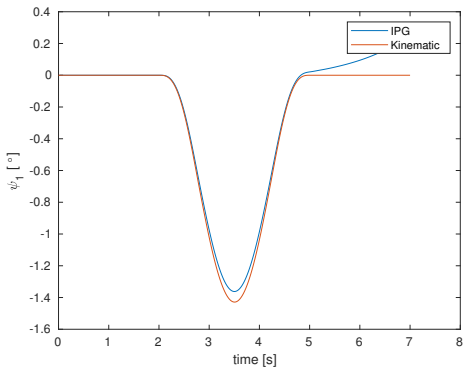


Figure 3.20: The car heading appears to track well until about $4.5s$ where the IPG model appears to have increased the heading. This is suspected to be due to bump steer; plus the car probably has too soft of springs.

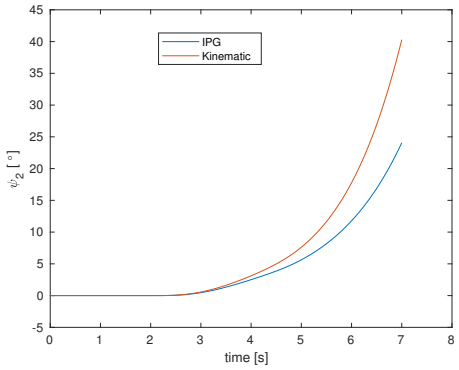


Figure 3.21: The kinematic models overestimate the heading of the trailer here because the heading of the tractor straightens out.

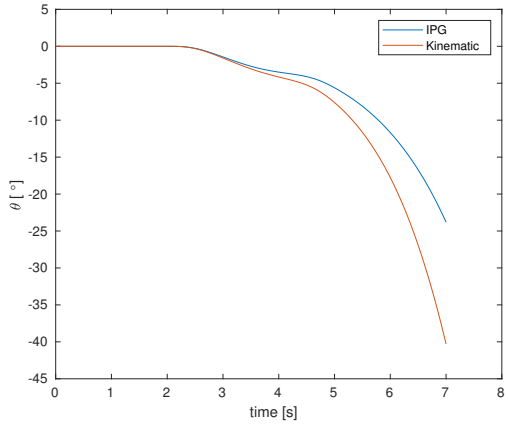


Figure 3.22: The hitch angle appears to deviate by about $16°$

There are a number of reasons for the difference in response between the kinematic and the IPG models. First, as mentioned before, IPG CarMaker models lateral load transfer and varying cornering stiffnesses. The kinematic model does not consider tires at all. This is likely why the kinematic model overestimates the heading angles. Second, as explained before, additional dynamics of the suspension can affect the steering through bump steer. Given that the motion between the two simulations track similarly and the evaluations will be conducted at slower vehicle speeds, the kinematic model is considered as an adequate representation of the primary motion. This section

validated the primary motion of the kinematic model, deeming it sufficient for the comparison of controllers.

The work in this section was evaluated in MATLAB/Simulink. Originally it was thought that the LQR simulations could be done in MATLAB/Simulink and could be compared to the machine learning work in Python. This was found to not be the best idea.

3.9 Comparison between MATLAB and Python Solvers

Both MATLAB and Python versions of the environment use a fixed time step solver equivalent of Runge Kutta. Despite having the same equations, the numerical solvers unfortunately produced different results. Figure 3.23 on the left shows the heading of the tractor for the same sinusoidal input and driving in reverse, both in MATLAB and Python. Figure 3.24 on the right zooms in on Figure 3.23 to display how the results are not the same. The results appear to deviate by a mere 50ms, just slightly delayed.

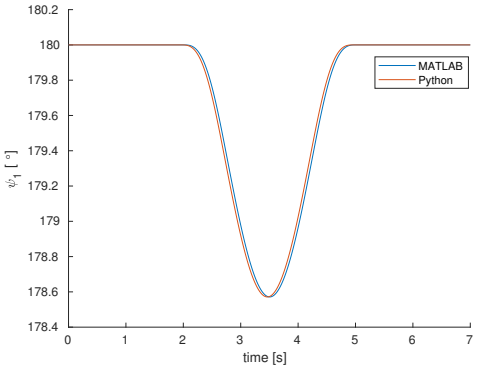


Figure 3.23: The heading angle of the tractor going in reverse.

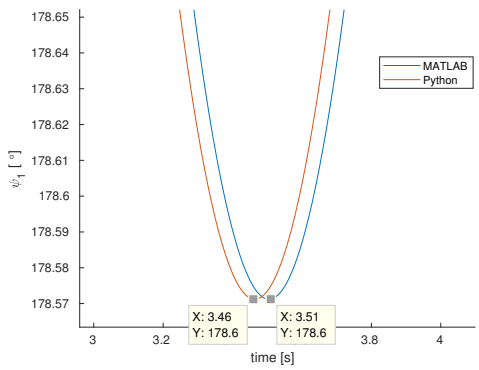


Figure 3.24: The heading angle of the tractor appears to be off by 50ms.

Unfortunately a similar trend is shown with the heading of the trailer with Figures 3.25 and 3.26. This time it appears the heading is actually different by 1.3°. Figures 3.27 and 3.28 also show a similar problem.

When trying to hunt down the source of the error, it appears it may be due to the difference of solvers. The python ode solver is described as a Runge-Kutta method of order (4)5 due to Dormand & Prince. The MATLAB ode4 is a 4th order Runge-Kutta Formula. The simulation was run again using MATLAB's ode5 solver which is a 5th order Dormand-Prince Formula. The results were still different. *Scipy.integrate* also has an 8th order Dormand and Prince Formula which is comparable to ode8 Dormand-Prince in MATLAB, but this also did not appear fruitful.

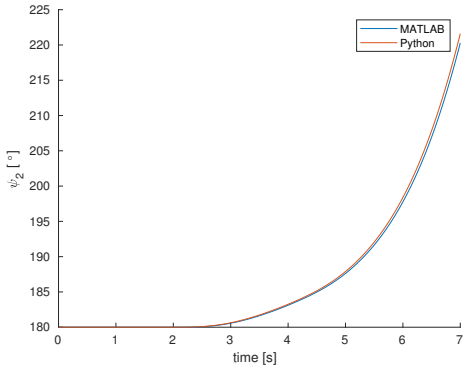


Figure 3.25: The heading angle of the trailer going in reverse.

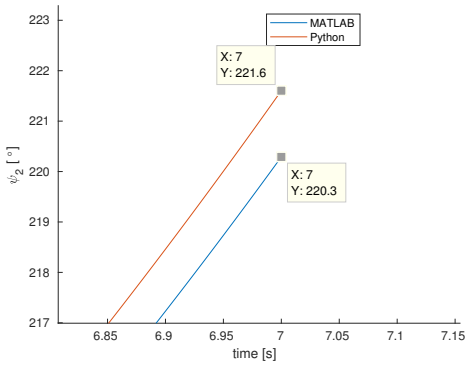


Figure 3.26: The heading angle of the trailer appears to be off by 1.3 degrees.

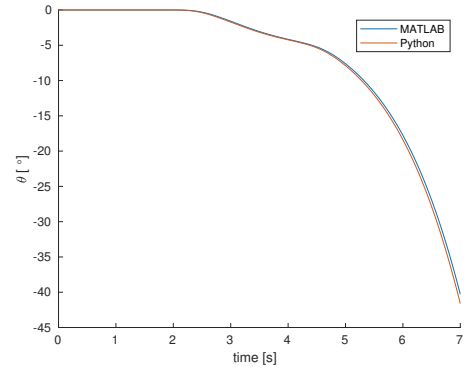


Figure 3.27: The hitch angle going in reverse.

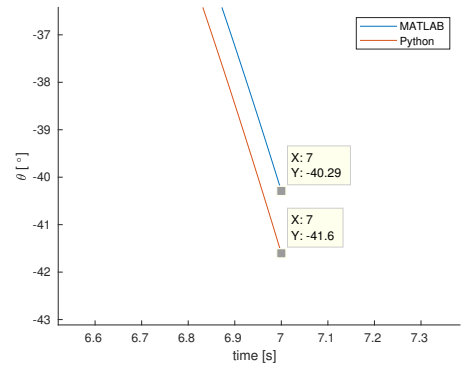


Figure 3.28: The heading hitch angle appears to also be off by 1.31 degrees.

Further investigation of Python's Runge-Kutta method of order 4(5) due to Dormand & Prince led to changing solver parameters. Many parameters were varied, but the ultimate conclusion was that the time spent finding a way to make Python and MATLAB comparable was not worth it. Thus, this thesis makes the comparison of algorithms in Python because it supports Tensorflow– the API useful for machine learning.

3.10 Summary

This chapter derived the kinematic model, which can be formulated to a state space representation useful for modern controls. The non-linear differential equations were evaluated against a multi-body dynamic software called IPG CarMaker. A fixed timestep was selected using a convergence test. Differences in fixed timestep solvers between MATLAB and Python resulted in the decision to use Python.

Chapter 4

PATH PLANNING USING DUBINS CURVES

The same information is fed to both the LQR and DDPG. Representing the path initially began with manually creating tracks such as circles, ovals, and hairpin turns. They were generated using manual equations and represented as a vector of x, y Cartesian coordinates. For example, the oval track in Figure 4.1 consisted of manually plotting points of circles and straights until they lined up to ultimately concatenate the selected points.

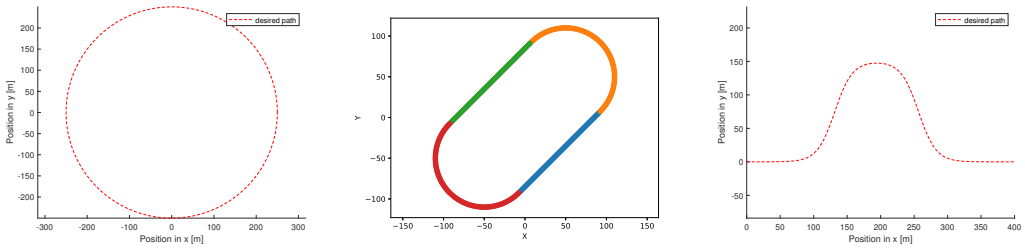


Figure 4.1: Manual tracks created include circles, ovals, and hairpin turns represented by x, y coordinates.

This quickly became tedious and it was theorized that reinforcement learning would need to train on thousands of different tracks if a generalized policy were to be learned. Having a simple track was useful in the early stages, so the function for the hairpin turn is displayed below:

$$y = 75(1 + \tanh(0.038(x - 125) - 0.25)) - 75(1 + \tanh(0.038(x - 250) - 0.25)) \tag{4.1}$$

This section discusses representing the reference path in a manner that the Linear Quadratic Regulator (LQR) can read error from the path. It first talks about curvature and how it is used to run the kinematic tractor-trailer model open loop. Then it describes a problem with the angle orientation error. The section then discusses localisation, i.e. where the vehicle is within the map containing the reference path. This section continues to describe the implementation of Dubins Curves for path generation and how it is modified for a tractor-trailer reversal. It discusses how one determines if the vehicle made it to the goal. The section concludes with describing the animations and tricks for graphics.

Some sections may discuss the concepts in MATLAB because this was where the work started. The MATLAB simulations are standalone from the Python implementations. It is worth mentioning is that the tracks were manually generated in Python ahead of time, including the precalculated curvature and angle orientations. The lessons learned and fixes were then rewritten in Python to be

its own entity. It is important to mention for the future discussions that array indices start at one with MATLAB, but start with zero in Python.

4.1 Curvature

Curvature is a useful property of a path because it can be used for approximating what the steering angle should be. It is more useful than radius because it does not suffer from becoming ∞ when the path is straight. Curvature tends to be mentioned in literature [43, 38] to be $\kappa = \frac{1}{R}$. This is true, however, it is not immediately clear for how to obtain the curvature from the reference path in the map. One could also use the curvature differential equation where $Y' = \frac{dY}{dX}$ and $Y'' = \frac{d^2Y}{dX^2}$.

$$\kappa = \frac{Y''}{(1 + Y')^{\frac{3}{2}}} \tag{4.2}$$

This, however, requires the track to be represented in some continuous form of $Y = f(x)$. Obtaining this from sensors poses additional challenges. The method that was found to be useful is called Menger curvature. It is found by calculating the area of the triangle between the three points and dividing it by the product of the lengths. If the map contained arrays of x and y coordinates, then a single point, p_n can be described. From this point forward, the term vector will be used interchangeably with array to denote the computer science term of a one dimensional array. Figure 4.2 shows a visualization of how Menger curvature is calculated at p_n, using the point before and after it.

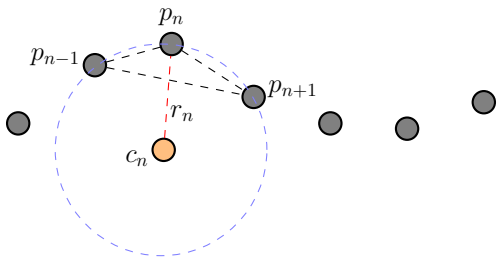


Figure 4.2: Menger Curvature is found by calculating the area of the triangle between the three points and dividing it by the product of the lengths.

The area of the triangle between the three points is denoted as A. The double bars around each of the terms in the denominator of the equation for κ signify a magnitude between each of the points.

$$\begin{aligned} A &= \frac{1}{2}((x_n - x_{n-1})(y_{n+1} - y_{n-1}) - (y_n - y_{n-1})(x_{n+1} - x_{n-1})) \\ \kappa = \frac{1}{R} &= \frac{4A}{||p_{n-1} - p_n||\,||p_n - p_{n+1}||\,||p_{n+1} - p_{n-1}||} \end{aligned} \tag{4.3}$$

As mentioned in Chapter 3, the *PathPlanner.py* file of the environment is a helper class with many functions for determining information about the path. Given points a, b, and c, the implementation of the distance function and the curvature calculation is shown below.

The curvature values will be positive or negative depending on the direction of the path. For each value in the arrays of x, y coordinates, a curvature value can be determined. Terminal conditions are needed when using the function such that the index of the vector does not exceed its size. Repeating indices were found the be adequate to counteract this. When the vehicle drives in reverse, it was discovered that inverting the sign of each of the elements in the curvature vector produced the desired behavior.

```
import numpy as np

class PathPlanner:
    def __init__(self):
        ''' '''
    def distance(self, a, b):
        return np.sqrt((b[0] - a[0]) ** 2 + (b[1] - a[1]) ** 2)

    def get_menger_curvature(self, a, b, c):
        raw_area = (b[0] - a[0]) * (c[1] - a[1]) - (b[1] - a[1]) *\
                   (c[0] - a[0])
        triangle_area = raw_area / 2.0
        return 4 * triangle_area / (self.distance(a, b) *\
               self.distance(b, c) * self.distance(c, a))
```

Now that curvature of a predefined path can be obtained, a good sanity check for checking system equations is to first drive the tractor-trailer model open loop going forwards.

4.2 Open Loop Driving with Curvature

Open loop control refers to instructing a system to go to a desired value without any feedback. In other words, this would be as if one were to close their eyes before performing a lane change maneuver. They have an idea of where they need to move to, but they will lack the visual feedback in order to correct for errors.

The desired steering angle can be calculated by multiplying the path curvature and the tractor length. Due to Ackermann steering angle [36], the open loop steering for cars can thus be approximated as $\delta = \kappa L_1$ according to Boyali and Baur [43, 46]. When driving forward, the tractor-trailer model can use this same input because the trailer simply tracks what the tractor is doing. Figure 4.3 displays a kinematic car model driving open loop on the hairpin course. The response starts on

the path at the circle marker, follows the path decently, and eventually trails off with a steady error ending at the 'x' marker.

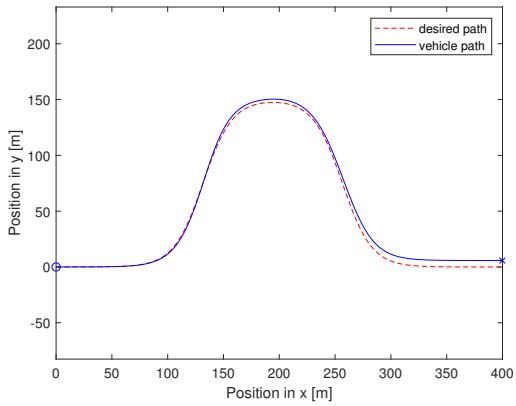


Figure 4.3: A kinematic car model's response to open loop control of the hairpin turn.

This is a great approximation for when going forward, but not for when driving in reverse because the trailer will jackknife. The vehicle follows the path, but has no ability to correct for errors because it is simply feed forward. This works for the kinetic model as well, however, the performance varies greatly with speed and tire slip.

For completeness, the nonlinear and linearized equations for the kinematic car are provided. For brevity, the reader is referred to Boyali [43] for the derivation.

$$\begin{aligned} \dot{x} &= v_x \cos\psi \\ \dot{y} &= v_x \sin\psi \\ \dot{\psi} &= \frac{v_x}{L}\tan\delta \end{aligned} \tag{4.4}$$

$$\begin{bmatrix} \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & v_x \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y \\ \psi \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{v_x}{L} \end{bmatrix} \delta$$

In addition to curvature being used for open loop control, it can also be utilized as a feedforward term. This will discussed in Chapter 5. At this point in the research, localization was not used. The curvature vector is instead interpolated in MATLAB ahead of time such that each desired point is generated based on a fixed change in time. Now that open loop has been discussed, the lateral error from the path can be reviewed.

4.3 Error from the Path

The lateral error, y_{2e}, is used for the controller to make the tractor-trailer converge to a reference point on the path. In order to determine this error, one must consider the tractor-trailer's own

coordinate system and the map's global coordinate system. The tractor-trailer's local coordinate system is defined as the midpoint on the trailer's wheel axle, which has the same angle orientation as the trailer. This coordinate system moves along with the trailer whereas the reference path is defined in the global coordinate system. Figure 4.4 illustrates the position errors from the trailer's coordinates to a reference point (x_r, y_r, ψ_r) on the path.

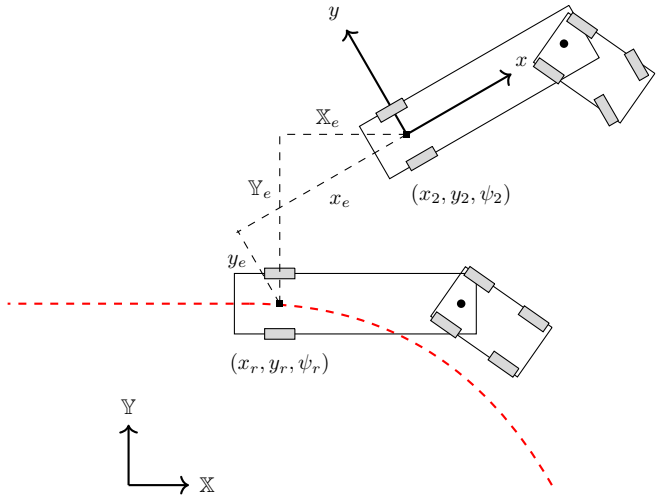


Figure 4.4: The trailer lateral path error is the difference between the reference point and the actual. One can think of the reference point as a ghost trailer that the actual must follow.

The two coordinate systems can be related using the current heading angle of the trailer, ψ_2, because the angles are both defined using a unit circle. The lateral error of the trailer from the path can be described in the local coordinate system using a frame rotation, or a passive rotation. This rotation is known as a passive rotation because it is not actually performing a rotation, rather describes the vector in another coordinate system. Reversing this rotation is known as an active rotation, which is often seen in computer graphics.

The error is first subtracted using the global coordinates. The output of the kinematic tractor-trailer model is actually the global coordinates because that is how they were defined using the initial conditions. The controller operates in the local coordinate system, so one must use a transformation or the frame rotation shown below. In general, the frame rotation used is the Direction Cosine Matrix (DCM) below:

$$\begin{bmatrix} x_e \\ y_e \\ \psi_e \end{bmatrix} = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r - x \\ y_r - y \\ \psi_r - \psi \end{bmatrix} \tag{4.5}$$

The angle orientation that is used in the DCM is the current trailer heading, so ψ_2 is used for the trailer. Similarly for the kinematic car model or the tractor, ψ_1 would be used. Using the DCM means that the longitudinal error, x_e, also needs to be calculated. Although the longitudinal error is not used for control because of constant velocity, it must be measured in order to perform the frame rotations.

In Simulink, the DCM was implemented using a MATLAB function block as surrounded by the red box in Figure 4.5. The MATLAB function blocks are a useful way to create complex functions without having a tremendous amount of extra '*.m*' files. The code is saved within the Simulink file. An item to note about these MATLAB function blocks are that the variables inside are extremely local. Another way to say this is that these MATLAB function blocks cannot even read from the workspace; it literally only has access to information given to the function as an argument.

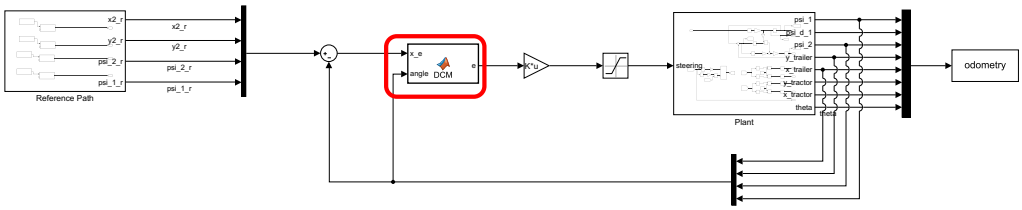


Figure 4.5: The *DCM*() function was implemented using a MATLAB function block in Simulink, shown by the red box.

In Python, the DCM is used inside the *truck_backerupper_env.py* file with the *get_error*() method. The DCM is written as a lambda function, which is a useful local function within Python.

```
    def get_error(self, i):
        self.last_index = self.get_closest_index(self.x2[i], self.y2[i],
                                                 self.last_index,
                                                 self.track_vector,
                                                 self.look_ahead)

        self.last_c_index = self.get_closest_index(self.x1[i], self.y1[i],
                                                 self.last_c_index,
                                                 self.track_vector,
                                                 self.look_ahead)
        self.r_x2[i] = self.track_vector[self.last_index, 0]
        self.r_y2[i] = self.track_vector[self.last_index, 1]
        self.r_psi_2[i] = self.track_vector[self.last_index, 3]
        self.r_psi_1[i] = self.track_vector[self.last_c_index, 3]

        self.psi_2_e[i] = self.path_planner.safe_minus(self.r_psi_2[i],
```

```
                                                    self.psi_2[i])
        self.psi_1_e[i] = self.path_planner.safe_minus(self.r_psi_1[i],
                                                    self.psi_1[i])
        self.x2_e[i] = self.r_x2[i] - self.x2[i]
        self.y2_e[i] = self.r_y2[i] - self.y2[i]

        self.error[i, 0:3] = self.DCM(self.psi_2[i]).dot(
                                            np.array([self.x2_e[i],
                                            self.y2_e[i],
                                            self.psi_2[i]]).T)
        return np.array([self.psi_1_e[i], self.psi_2_e[i], self.error[i, 1]])
```

One may notice additional functions; the *get_closest_index*() function is used for localization and the *safe_minus*() function is used for rectifying an improper calculation of the angle orientation error. The next two sections will cover the topics.

4.4 Fixing the Angle Orientation Error

The angle orientation of the path needs to be determined in the global Cartesian coordinate system to create a desired heading value. Given two points, one can determine the angle orientation of the path using the arc tangent. The *atan*2() function is popular due to its ability to handle the four quadrants, but is limited to the range of $[-\pi, \pi]$. One can modify the result to get $[0, 2\pi]$ using the modulo operator:

$$\psi = (\psi + 2\pi) \quad \% \quad (2\pi) \tag{4.6}$$

The angle orientation error is used for the controller to make the trailer have the same orientation as the reference point on the path. Originally, the orientation error was calculated as $\psi_e = \psi_r - \psi$. It was found in initial runs with the LQR that whenever an error term crossed the $[0, 2\pi]$ axis, then the controller was immediately disturbed. It was as if the desired values were flipped 360° instantaneously. The vehicle would go off path and have trouble getting back on. Just to clarify, this was even after making the angle orientation be within $[0, 2\pi]$ using equation 4.6. Figure 4.6 highlights the exact moment the issue occurs on a circle track. At first the LQR is performing as expected, but then the orientation error suddenly claims it is $-360°$ off.

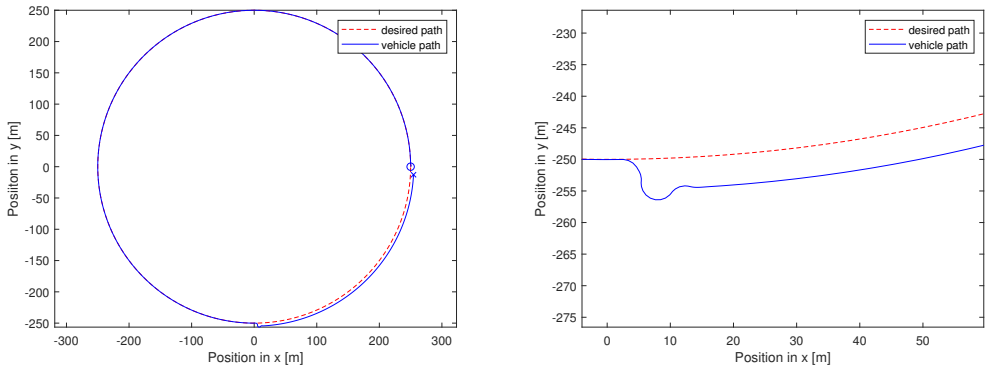


Figure 4.6: The kinematic car path is shown in blue along the circle path. The controller performs well until the angle orientation error suddenly grows. The response is zoomed in on the right.

In fact, when looking at the desired angle orientation values calculated from $atan2()$ for the circle path, a discontinuity is present when the angle increases beyond 2π. The zero crossing issue can be seen in Figure 4.7. When looking at the errors in Figure 4.8, one can confirm the error is now flipped a full unit circle.

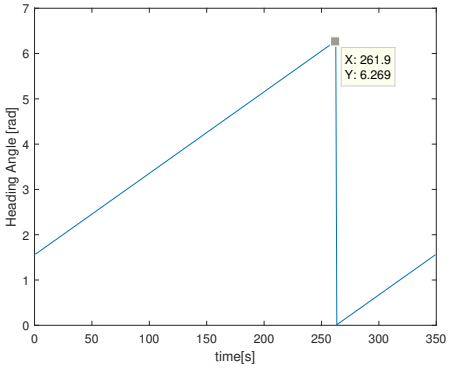


Figure 4.7: The desired angle orientation contains a discontinuity due to $n\pi$ not being accounted for by $atan2()$.

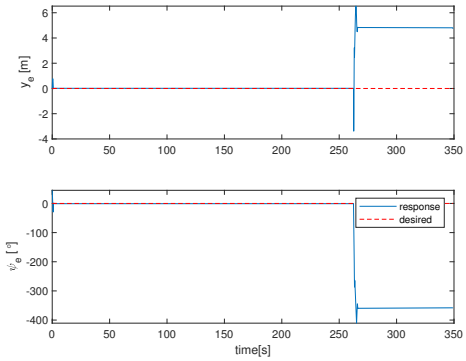


Figure 4.8: The angle orientation error is confirmed to have flipped a full unit circle, but unfortunately the performance of the lateral error is also degraded.

One should recognize the desire is to compute the orientation error in the direction in which the error is smaller; this is to ensure the vehicle takes the shortest way to the path. The vehicle does not take the shortest path when the orientation error is larger than π. The problem occurs whenever there is a discontinuity, so it does not actually matter if the angle range is $[0, 2\pi]$ or $[-\pi, \pi]$. In other words, a rectification will always be needed as long as the orientation is bounded within a region. The orientation should be bounded within a region so the error is deterministic. The rectification

for this problem is outlined by Elhassan [38], which is essentially an if-statement.

$$\psi_e = \begin{cases} \psi_r - \psi & \text{if} \quad |\psi_r - \psi| \leq \pi \\ \psi_r - \psi - 2\pi sign(\psi_r - \psi) & \text{else} \end{cases} \tag{4.7}$$

The solution is to add or subtract 2π if the orientation error is larger than π. Quaternions were investigated due to their ability to handle singularities better, however, this would require redefining plants in terms of quaternion feedback. In addition to this, quaternions still suffer from one singularity where it crosses the zero axis.

```
1    def safe_minus(self, r, c):
2            if abs(r-c) <= np.pi:
3                error = r - c
4            else:
5                error = r - c - 2 * np.pi * np.sign(r - c)
6            return error
```

In Python, the implementation was another method called $safe\_minus()$ contained in the $PathPlanner.py$ file. In Simulink, a MATLAB function block called $safeminus()$ was used to implement this fix which is shown in Figure 4.9. The solution appeared to help significantly, but a remnant of the problem still appeared to exist in the form of a blip which can be seen in Figure 4.10.

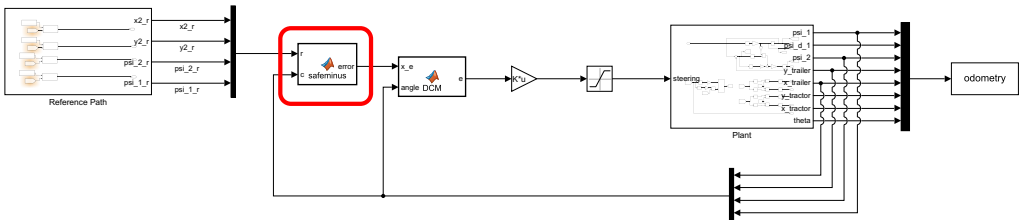


Figure 4.9: The $safeminus()$ rectification was implemented using a MATLAB function block in Simulink, shown by the red box.

Evidently, this was due to interpolating the desired values. The remnant disappeared once interpolation was turned off and holding final value was set in Simulink. The reason interpolation was done was because then one is at the mercy of the resolution of the track points, where controller performance may degrade. Nonetheless, localization was found to be needed, so interpolation was no longer performed. In summary, the orientation error problem is solved by using $safe\_minus()$.

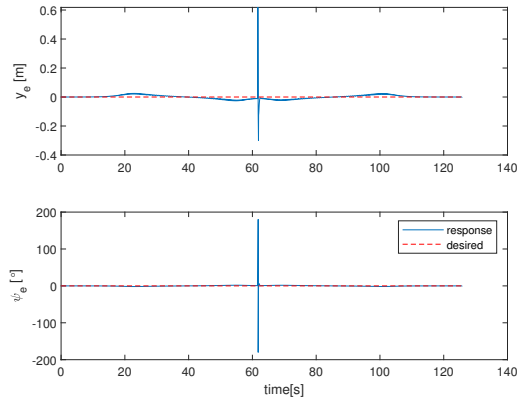


Figure 4.10: The problem was significantly mitigated with $safeminus()$, but interpolation still caused a remnant of the problem. The orientation error is completely solved when using $safeminus()$ and localization.

4.5 Localization

Previous methods for controlling the tractor-trailer had taken the predefined course and fed them into the controller based on a fixed time step calculated from the constant velocity. This meant that it assumed the vehicle started at the beginning of the path and followed it properly. This, however, means that the incorrect error terms are being fed to the control. This results in suboptimal performance. Not only that, but it can prevent the trailer from meeting the goal as shown in Figure 4.12.

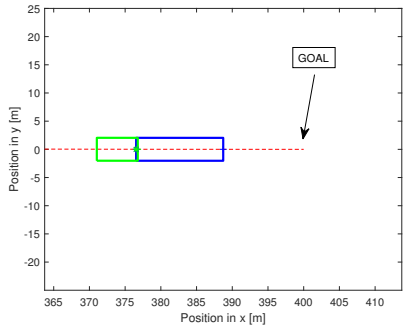


Figure 4.11: The trailer stops short of the goal because it was not getting the correct error signal. It had no idea where it was with respect to the actual path.

To account for this, a function called $get\_closest\_index()$ was written. The idea is to search the track vectors of coordinates and heading. Searching the entire array takes too much computational power and is not very efficient. The function takes the last index and calculates the distance from the trailer to that index of the track. It sets this as the current minimum distance. Then it begins

searching behind to see if the distance is closer behind. If the current distance is less than the minimum distance, then searching forward is not necessary. If the current distance is more than the minimum index, then searching forward is necessary. The idea is visually shown in Figure 4.12.

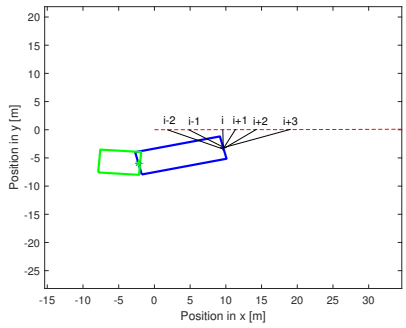


Figure 4.12: The idea is to efficiently search where the trailer is with respect the path using the last index.

Algorithm 7 Get Closest Index

```
1:  function GET_CLOSEST_INDEX(x, y, last_index, track_vector)
2:    search_behind ← n_1
3:    search_ahead ← n_2
4:    min_index ← last_index
5:    min_dist ← distance( track_vector(last_index), current_position )
6:    search_forward ← True

7:    for i ← last_index : last_index + search_behind - 1 do
8:      current_dist ← distance( track_vector(i), current_position )
9:      if current_dist < min_dist then
10:       min_dist ← current_dist
11:       min_index ← i
12:       search_forward ← False

13:   if search_forward is True then
14:     for i ← min_index : search_length do
15:       current_dist ← distance( track_vector(i), current_position )
16:       if current_dist < min_dist then
17:         min_dist ← current_dist
18:         min_index ← i
      return min_index
```

An added bonus of only searching nearby indices is that now paths that intersect or loop over one another can be driven. Two indices are really needed for localization with a tractor-trailer system: the tractor cab and the trailer. For the Simulink implementation, the functionality can be found in the localization subsystem as shown in Figure 4.13. Inside of the subsystem are two identical MATLAB function blocks, one for the tractor and one for the trailer. The output of the functions

are *last_index* and *last_c_index*, where the '*c*' stands for cab. Notice in Figure 4.14, the outputs are fed back into a memory block. This is needed to prevent Simulink errors of algebraic loops.

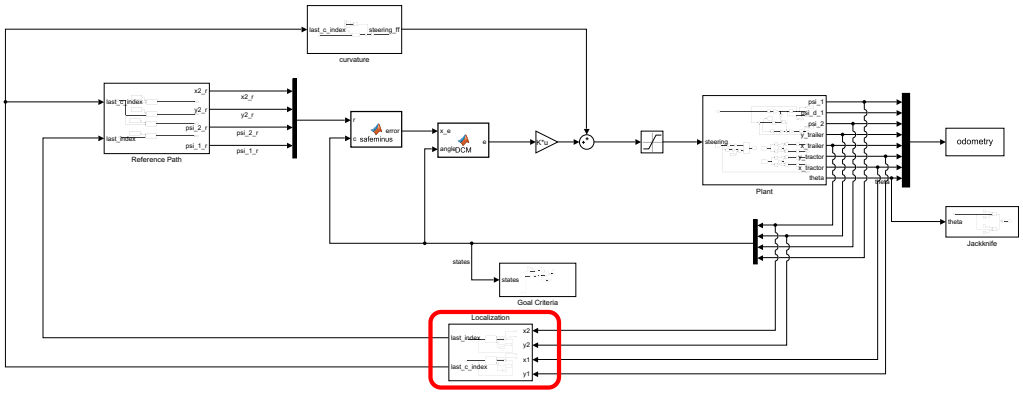


Figure 4.13: The subsystem containing localization functionality is found in the red box.

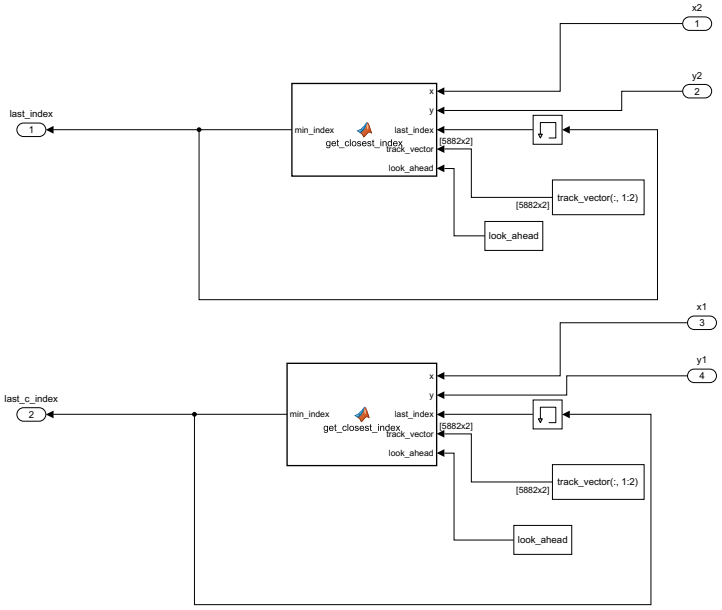


Figure 4.14: Inside the localization subsystem in Simulink, two MATLAB function blocks are used for the two indices needed.

In Python, the *get_closest_index*() function was placed in the *truck_backerupper_env.py* file. The *search_ahead* and *search_behind* variables were found to work for the desired speeds and track resolution, however, these are ultimately tuning parameters. They are both selected to be 10 indices.

```
    def get_closest_index(self, x, y, last_index, track_vector, look_ahead):
        min_index = last_index
        min_dist = self.path_planner.distance([track_vector[last_index, 0],
                                               track_vector[last_index, 1]],
                                               [x, y])
        search_forward = True

        ## search behind
        search_behind = 10
        if last_index > search_behind:
            last_index = last_index - search_behind

        for i in range(last_index, last_index + search_behind - 1):
            cur_dist = self.path_planner.distance([track_vector[i, 0],
                                                   track_vector[i, 1]],
                                                   [x, y])
            if cur_dist < min_dist:
                min_dist = cur_dist
                min_index = i
                search_forward = False

            ## search ahead
            if search_forward:
                search_ahead = 10
                if min_index > len(track_vector) - search_ahead:
                    search_length = len(track_vector)
                else:
                    search_length = min_index + search_ahead

                for i in range(min_index, search_length):
                    cur_dist = self.path_planner.distance([track_vector[i, 0],
                                                           track_vector[i, 1]],
                                                           [x, y])
                    if cur_dist < min_dist:
                        min_dist = cur_dist
                        min_index = i

            ## implement look_ahead
            if look_ahead:
                if min_index >= len(track_vector) - look_ahead:
                    pass
                else:
                    min_index = min_index + look_ahead
        return min_index
```

When looking at the implementation, one might notice that a term called *look_ahead* is also used. This is useful for predicting what is to come ahead and will be discussed in the next section.

4.6 Look Ahead

The path following can be resembled by greyhound track racing [38]. A fake rabbit is a moving reference point, p_r, on the path. The greyhounds chase after the rabbit until the goal is reached,

the first to get to the finish line wins the race. When the lookahead is zero, the reference point that *get_closest_index*() determines is the point that is closest to the rear axle of the trailer. This may result in more aggressive steering towards the path and struggle in transitions between straights and curves. If the lookahead is greater than zero, *get_closest_index*() looks a bit further down the path as shown in Figure 4.15. This would consequently result in a smoother steering response.

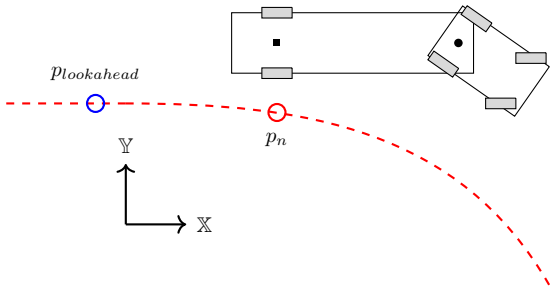


Figure 4.15: The calculated reference point from *get_closest_index*() can be modified to select a few indices ahead to promote a smoother steering response for situations when transitioning between a straight and a curve.

Choosing the lookahead is another tuning variable. Choosing too long of a lookahead will cause the controller to react to what lies ahead too early. It was found that a lookahead of 15 indices caused the controller to jackknife the trailer when the velocity was set to $-2.012\frac{m}{s}$. So violent and unpredictable behavior can result when the lookahead is set too large. It was, however, beneficial to have a lookahead of 10 indices and below when looking at a few tracks. Unfortunately when the velocity was increased, the optimal lookhead was required to be different. Ideally, this value would be a function of velocity or time. Given that this research may continually to change velocity, it was ultimately decided to leave the lookahead at zero and modify the Dubins Curves to better suit a tractor-trailer.

4.7 Modification to Dubins Curves

Dubins Curves use a forward driving kinematic car model to evaluate the shortest distance between six path options of three consecutive actions of left, right, or straight. The path planning method does not account for obstacles, but instead constrain the paths using the minimum turning radius. The minimum turning radius for cars or the tractor can be approximated using the maximum

steering angle and the wheelbase. The following equation is similar to Ackermann steering, except it is more conservative because it approximates the outer wheel of the front axle.

$$R = \frac{L_1}{\sin \delta_{max}} \tag{4.8}$$

With a maximum steering angle of 45° and a wheelbase of $L_1 = 5.74m$, the minimum turning radius of the tractor is $8.118m$. With combination vehicles, however, the trailer really dictates what the minimum turning radius is. The calculation can be performed similarly, except for this time the maximum desired hitch angle and trailer wheelbase is used.

$$R = \frac{L_2}{\sin \theta_{max}} \tag{4.9}$$

Jackknifing the trailer with a wheelbase of $L_2 = 10.192m$ at 90° results in a a minimum turning radius of the wheelbase. For evaluating paths to the loading dock, this is not the maximum desired hitch angle. According to City of Portland Office of Transportation [74], the city's design requirements for minimum turning radius for combination vehicles is 45 feet or $13.716m$. This would result in a maximum hitch angle of 48°. To mimic realistic paths using Dubins Curves, the minimum turning radius was selected as $13.716m$.

Andrew Walker provides a ready-made package to determine Dubins Paths given initial and ending pose [75]. The code is written in C, however, there claims to be a MATLAB and Python wrapper. The Python wrapper is also written by him and works. The MATLAB wrapper written using mex is severely outdated after Walker made significant structural changes. Mex or MATLAB executables are used for translating code between C and MATLAB. The MATLAB wrapper works with the 2016 version of the code on github, but not the one at the time of this work. The Python wrapper translates the C code to Python using a module called Cython. One can download the code directly from github, but it is much easier to install using pip in the terminal of Linux. Some example default Dubins Curves generated are shown in Figure 4.16

```
1    >>> pip3 install dubins
```

Dubins paths are valid for going forward since it accounts for the minimum turning radius, but not so well in reverse. Usually a Dubins path ends on a curve as shown in Figure 4.17 represented in black. This can be mitigated by modifying the Dubins Path to include a straight at the end. The *PathPlanner.py* file uses Dubins, but modifies the generated path to include a straight as shown as the combination of red and blue lines. Given a random starting and ending pose, the *generate*() method offsets the goal by two times the turning radius in the goal orientation. The Dubins package

runs normally as if it would, but then is concatenated with a straight line to the goal. This makes it easier for the trailer to end up in the correct orientation.

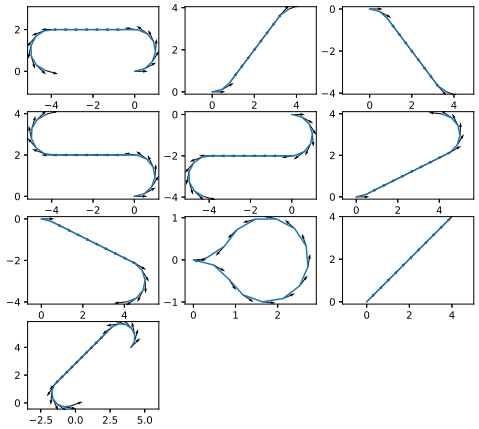


Figure 4.16: The example Dubins Curves that Walker provides are shown.

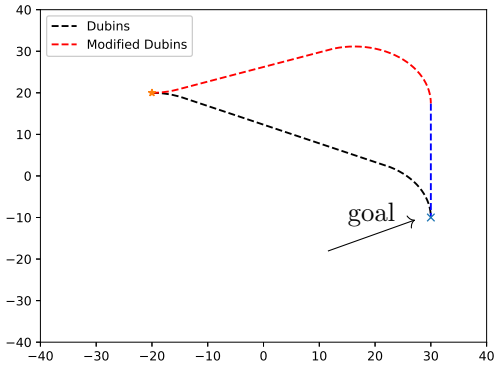


Figure 4.17: Dubins was modified to not end on a curve.

```
    def generate(self, q0, qg):
        self.q0 = q0
        self.qg = qg

        ## Modify dubins to work for a straight offset from goal
        self.q1 = self.qg.copy()
        self.q1[0] -= 2.0 * self.turning_radius * np.cos(self.q1[2])
        self.q1[1] -= 2.0 * self.turning_radius * np.sin(self.q1[2])

        # Dubins
        path = dubins.shortest_path(self.q0, self.q1, self.turning_radius)
        qs, dist_dubins = path.sample_many(self.step_size)
```

```
        qs = np.array(qs)

        ## Concatenate with reverse straight
        s_s = self.distance((self.q1[0], self.q1[1]), (self.qg[0], self.qg[1]))
        n_steps = int(s_s // self.step_size) + 1
        straight = np.array([np.linspace(self.q1[0], self.qg[0], n_steps),
                             np.linspace(self.q1[1], self.qg[1], n_steps),
                             self.qg[2] * np.ones(n_steps)]).T
        qs = np.vstack((qs, straight))

        dist_straight = [dist_dubins[-1]]
        for j in range(len(straight)):
            dist_straight.append(dist_straight[j] + (s_s / n_steps))
        self.dist = dist_dubins + dist_straight[1:] # ignore double counting

        ## x, y, curv, psi, dist
        self.curv = []
        for n in range(len(qs)):
            if n == 0:
                self.curv.append(self.get_menger_curvature(qs[0], qs[n+1],
                                 qs[n+2]))
            elif n == len(qs) - 1:
                self.curv.append(self.get_menger_curvature(qs[n-2], qs[n-1],
                                 qs[n]))
            else:
                self.curv.append(self.get_menger_curvature(qs[n-1], qs[n],
                                 qs[n+1]))

        self.x = qs[:, 0]
        self.y = qs[:, 1]
        self.psi = qs[:, 2]
        return np.column_stack([self.x, self.y, self.curv, self.psi,
                        self.dist])
```

The *generate*() method in *PathPlanner.py* is used whenever the *reset*() method is called from the class in *truck_backerupper_env.py*. As mentioned earlier, the only way to constrain Dubins is with the minimum turning radius. The environment is set within a $80 \times 80m^2$ area to represent a loading dock. It is possible for Dubins to spawn a path that exceeds the boundaries. Even though the random starting and ending locations are constrained, the minimum turning radius can take it outside of the boundary and then back into it. Thus, if any of the points on the generated track exceeds the boundary, *generate*() is simply run again until it finds a random, valid path. This is not the most computationally efficient, but Dubins is much faster than existing path planning since it is so simple.

It is also possible for Dubins to generate paths that overlap the goal as if the trailer would crash into the loading dock. In a similar fashion, *generate*() is also re-ran if the distance from the goal

and any of the points in the track vectors are less than or equal to $5m$. The entire track vectors are searched, except for the last $5m$ because it is obvious that this is true.

Even though the planned paths are provided ahead of time, planning is not performed mid run. Even though the planned paths should be reasonable, the generated paths may have transitions that are not ideal for a tractor-trailer when a large error is already present. A fixed reference path is used to promote the evaluation of controllers and the deviations from a consistent path.

4.8 Terminal Criteria

Terminal criteria are conditions in which the controller does not perform as desired, such as jackknifing. It makes sense to end the run because the tractor-trailer is past the point of recovery. At each step within the environment, the terminal conditions are checked. If it is met, the done flag is set to end the episode and then try a new track. Jackknifing is considered to be when the hitch angle exceeds $\pm 90°$. Another terminal condition is if either the tractor or trailer rear axle exceeds the $80 \times 80m^2$ boundary. The the time for completion is also limited to avoid excessively wandering from the path. It was found that $160s$ was a reasonable amount of time for the various paths that are spawned within the area.

Terminal criteria can also be good; this happens when the rear most point of the trailer meets the goal criteria. The goal criteria is if the rear most point of the trailer is within $15cm$ and the angle orientation of the trailer is within 0.1 radians with the loading dock orientation. The control methods seek to converge the trailer rear axle to the reference path at every given timestep. In order to determine the distance of the rear most point of the trailer to the goal, one can measure the position relative to the rear axle using the following equation.

$$\begin{bmatrix} \bar{x} \\ \bar{y} \\ 1 \end{bmatrix} = \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} + \begin{bmatrix} \cos\psi_2 & -\sin\psi_2 & 0 \\ \sin\psi_2 & \cos\psi_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -b \\ 0 \\ 1 \end{bmatrix} \tag{4.10}$$

The rear most point of the trailer, $(\bar{x}, \bar{y})$, is found by adding an active rotation of the longitudinal bias, b, in the direction of the desired point. This can be thought of as offsetting the length between the rear axle and the end of the trailer in the local coordinate system and then transforming the bias to where it should be in the global coordinate system. Notice the rotation matrix is the transpose of the passive rotation mentioned before; this is an active rotation.

The variable representing minimum distance to the goal is initially set to be a really large number. Every timestep, the minimum distance to the goal is continually updated. If one were to

stop the episode if the the vehicle made it within the criteria, every successful episode would report that the trailer was within $15cm$. Due to the desire to see how close the trailer can actually get to the loading dock, a method was devised to determine which side of the finish line the trailer was.

4.9 Determining Side of Finish Line

Determining when the rear most point of the trailer crosses the finish line is considered the last straw to meet the goal criteria. First a normal line to the goal position is needed. Then determining the side of the finish line can be accomplished by checking if two points are on opposite sides of the line. The normal vector can be determined by essentially applying a 90 degree counterclockwise rotation. First dx and dy should be defined from the last two points on the track, where the subscript g represents the goal.

$$\begin{aligned} dx &= q_{xg} - q_{xg-1} \\ dy &= q_{yg} - q_{yg-1} \end{aligned} \tag{4.11}$$

A line can thus be created between the points $(-dy, dx)$ and $(dy, -dx)$. The values of dx and dy can be scaled to increase the length of the line. The line, however, must be offset by the goal coordinates in order to truly display the intersection through the goal. The resulting end points of the normal line shall be denoted as (x_1, y_1) and (x_2, y_2). Given two points (a_x, a_y) and (b_x, b_y), one can determine if they are on opposite sides of the defined line if and only if the following term is less than zero:

$$((y_1 - y_2)(a_x - x_1) + (x_2 - x_1)(a_y - y_1))((y_1 - y_2)(b_x - x_1) + (x_2 - x_1)(b_y - y_1)) < 0 \tag{4.12}$$

The point (b_x, b_y) can be thought of as the trailer that has crossed the finish line in figure 4.18. Once the condition in equation 4.12 is met, then the fin flag is raised to be true. Fin represents 'finished' and will trigger a termination, but it has not necessarily met the goal criteria.

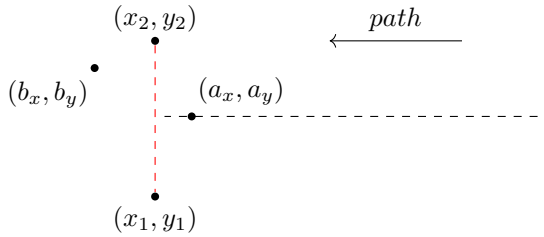


Figure 4.18: Determining if two points are on opposite sides of a line is shown.

If the trailer crosses the finish line, but does not meet the criteria of being within $15cm$ and the angle error of 0.1 radians, it still makes sense to terminate the episode. This is because the

controllers will likely try to loop back around towards the goal, but the tractor may jackknife in the process. Therefore, episodes may end only in a fin flag or both a fin flag and a goal flag.

In order to prevent early termination when the paths start near the goal, this calculation does not start until the distance to the goal is less than $5m$ and the absolute angle error to the goal is within $45°$. Now all the tools necessary to talk about graphics have been discussed. The next section will describe some computer graphics that were found necessary to generate animations to resemble the truck backerupper environment.

4.10 Computer Graphics

Animations are useful for debugging simulations and is particularly useful in this thesis to get an understanding of what the reinforcement learning is training on. For example, animating the simulation in early stages helped discover problems with initial conditions. Animations are simply plots within a for loop, clearing the screen each time.

Video games often measure smoothness with having a high enough FPS or Frames Per Second. Given that the agent interacts with the environment one step at a time, it is challenging to achieve real time rendering. This means having the velocity resemble what it would look like in real life. As it turns out, the *matplotlib* module in Python was clocked to result in about 17 FPS for this animation on the personal machine used.

Apparently what simple video games do to achieve better frame updates is to run the physics fixed timestep at the FPS. More complex video games, like Forza, probably run physics with a variable timestep to account for the FPS change. As mentioned in Chapter 3, a variable timestep was not used to improve the resolution of the simulation. What video games likely do is either interpolate in between the variable time steps or scale things accordingly. In order to keep things simple and since the fixed time step was chosen through a convergence test, real time rendering was no longer pursued.

An interesting discovery was made in plotting the motion of the combination vehicle when it translated and rotated. Initially, simply applying active rotations caused the vehicle to rotate about the origin as seen in Figure 4.19.

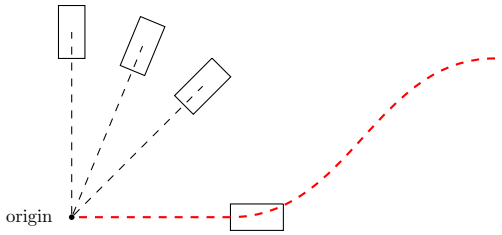


Figure 4.19: The combination vehicle would rotate about the origin instead of an arbitrary point.

It rotated about the origin of the global coordinates, instead of in its own local coordinate system. The computer graphics solution to this is to first translate the coordinates back to the origin, rotate by the angle ψ, and then translate back. The homogeneous rotation is accomplished by adding an additional dimension to the vector transformation. This is represented by the following equation.

$$\begin{bmatrix} \bar{x}_i \\ \bar{y}_i \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \tag{4.13}$$

The subscript i denotes the point of interest with relation to the arbitrary point (x, y). An example of this would be one of the tires in relation to the rear axle. The resultant vector, with the bar on top, is what should be plotted for the animation. Now that the path planning and necessary items for the environment have been explained, the design and implementations of the controllers can be discussed.

4.11 Summary

This chapter demonstrated how to calculate curvature from a reference path of Cartesian coordinates, which is useful for open loop studies with kinematic vehicles. The error from the path is described with passive rotations for the tractor-trailer. The simulation studies ran into issues with zero crossing of orientation angles, so a solution was presented. Localization was needed to get the correct reference values. Lookahead was discussed, but is ultimately not used in this work. The modification to the vanilla Dubins path planner is discussed to make it more realistic for a tractor-trailer. Properly placing the trailer at the loading dock is defined, where a method is discussed to determine the side of the finish line. Now the controllers can be designed and put to the test.

Chapter 5

MODERN CONTROLS IMPLEMENTATION

The modern controls implementation discussed in this chapter is for the Linear Quadratic Regulator (LQR). The design of the gains for the controller relies on having an accurate system model of the tractor-trailer. Various gains are evaluated over 100 tracks generated by the modified Dubins curves; the gains with the best performance are selected.

This chapter describes the implementation in MATLAB and Simulink because that is where the research started. The work assumes that the full state of the tractor-trailer model can be measured. In reality, the trailer position is difficult to measure because trailers tend to not have sensors implemented on them. A way to circumvent this complication is to use a state estimator or observer. Using the Kalman filter as the observer would turn the LQR into the Linear Quadratic Gaussian (LQG) controller.

Not having the full state turns the problem into a Partially Observable Markov Decision Process (POMDP), which also makes it difficult for the reinforcement learning algorithms to learn a good policy. According to Heess and Hunt in [76], a Recurrent Neural Network (RNN) using a history of observations can learn a policy for a POMDP. This can be considered as an analogy of using a state estimator in modern controls.

Thus, this thesis assumes the full state is available and is a Markov Decision Process (MDP). Evaluating the generalization or robustness of the controllers can still be answered using the full state. The design of the gains for the LQR is ultimately determined using Bryson's rule. The closed loop eigenvalues will demonstrate that even though the system is open loop unstable going backwards, the controller will stabilize the system. The system can be thought of as a lateral inverted pendulum where the tractor continuously pushes the trailer to the desired reference point and orientation.

This chapter focuses on the control design around the reversal of the kinematic model of the tractor-trailer, but control of a kinetic model was also investigated. Appendix A discusses the control of a kinetic model in the forward direction only. The kinematic model controller was implemented before any training with machine learning, so the following sections discuss what it took to achieve the performance that is set as the baseline.

5.1 Modern Controller Design

In order to determine the control law, $\underline{u} = K(\underline{r} - \underline{x})$, such that it minimizes the cost function, $J = \int_0^{t_f} (\underline{x}^T Q \underline{x} + \underline{u}^T R \underline{u}) dt$, one must solve the Algebraic Riccati Equation.

$$A^T S + SA + G^T QG - (SB + G^T QH)(H^T QH + \rho R)^{-1}(B^T S + H^T QG) = 0$$

Thus, a model that is in standard state space representation is needed. The linearized kinematic tractor-trailer model equations 3.34 and 3.35 are taken from Chapter 3.

$$\underline{\dot{x}} = A\underline{x} + B\underline{u}$$
$$\underline{y} = C\underline{x} + D\underline{u}$$

$$\begin{bmatrix} \dot{\psi_1} \\ \dot{\psi_2} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ \frac{v_{1x}}{L_2} & -\frac{v_{1x}}{L_2} & 0 \\ 0 & v_{1x} & 0 \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} + \begin{bmatrix} \frac{v_{1x}}{L_1} \\ -\frac{v_{1x} h}{L_1 L_2} \\ 0 \end{bmatrix} \delta$$

$$\begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \delta$$

A tractor with a semi-trailer would be modeled if the hitch length, h, was negative. This means the kingpin and fifth-wheel connection is longitudinally in front of the rear axle of the tractor. The nominal case used for this thesis actually uses a hitch length of zero, which means the connection point is on the rear axle. This is so controller robustness can be evaluated with the trailer connection point moving fore and aft of the rear axle. The velocity is set to negative because the system is driving in reverse towards the loading dock. The nominal kinematic model parameters in Table 5.1 are substituted into the equations.

Table 5.1: The nominal system parameters reside in the middle of the varied parameters.

Parameter	L_1	L_2	h	v_{1x}
-	5.74m	10.192m	0.00m	-2.012 $\frac{m}{s}$

$$\begin{bmatrix} \dot{\psi_1} \\ \dot{\psi_2} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ -0.1974 & 0.1974 & 0 \\ 0 & -2.0120 & 0 \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} + \begin{bmatrix} -0.3505 \\ 0 \\ 0 \end{bmatrix} \delta \tag{5.1}$$

$$\begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \delta \tag{5.2}$$

One must first check to see if the system is controllable using the controllability matrix. This essentially means that poles can be arbitrarily placed with the gains. If it is not controllable, then there is no sense in designing the controller.

$$P = [B\ AB\ A^2B...]$$

If the rank of P is the same as the number of states n, then the system is controllable. Using MATLAB's $rank(ctrb(A, B))$, results in a rank of three. Therefore, the system is controllable!

$$P = \begin{bmatrix} -0.3505 & 0 & 0 \\ 0 & 0.0692 & 0.0137 \\ 0 & 0 & -0.1392 \end{bmatrix} \tag{5.3}$$

In order to design the gains using LQR, one uses the matrices Q and R to penalize for performance. Q and R are both symmetric positive-definite matrices. The matrix Q penalizes the states and the matrix R penalizes the actions in the cost function J.

$$J = \int_0^{t_f} (\underline{x}^T Q \underline{x} + \underline{u}^T R \underline{u}) dt \tag{5.4}$$

The higher the value the term is in the matrix, the better the controller will drive the state's error to zero. The Q and R matrices are essentially used for fine-tuning the gains.

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad R = \begin{bmatrix} 1 \end{bmatrix} \tag{5.5}$$

One utilizes G, which represents the controlled states and H, which is basically what D usually is. For now, G is selected such that each of the states are incorporated into the cost function. When in doubt, make G identical to C.

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad H = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.6}$$

MATLAB's $lqr()$ function requires Q and R to be in a specific form because it writes the Algebraic Riccati equation differently:

$$A^T S + SA - (SB + N)R^{-1}(B^T S + N^T) + Q = 0 \tag{5.7}$$

According to Hespanha [54], it is suggested to write the arguments into the $lqr()$ function as shown below. The ρ variable is a positive constant used to provide a trade-off between conflicting goals of minimizing controlled output and controlled inputs. For the purpose of this thesis, ρ is kept at a constant of 1.

$$\begin{aligned} QQ &= G^T QG \\ RR &= H^T QH + \rho R \\ NN &= G^T QH \end{aligned} \tag{5.8}$$

```
sys = ss(A, B, C, D);
[K S e] = lqr(sys, QQ, RR, NN);
```

The function provides the solution to the Algebraic Riccati equation, S. The gain matrix, K, is derived from S using the following relationship.

$$S = \begin{bmatrix} 10.9119 & -34.5214 & 2.8529 \\ -34.5214 & 170.8540 & -19.3753 \\ 2.8529 & -19.3753 & 4.1131 \end{bmatrix} \tag{5.9}$$

$$K = (H'QH + \rho R)^{-1}(B^T S + (G'QH)^T) \tag{5.10}$$

This results in the following gain matrix, K. The function already returns the gain matrix, so one does not have to manually calculate this every time.

$$K = \begin{bmatrix} -3.8249 & 12.1005 & -1.0000 \end{bmatrix} \tag{5.11}$$

The closed loop eigenvalues are provided with e. As can be seen in Figure 5.2, the controller stabilized the once open-loop unstable system.

$$\lambda_1 = -0.5662 \quad , \quad \lambda_2 = -0.2886 + 0.4033i \quad , \quad \lambda_3 = -0.2886 - 0.4033i \tag{5.12}$$

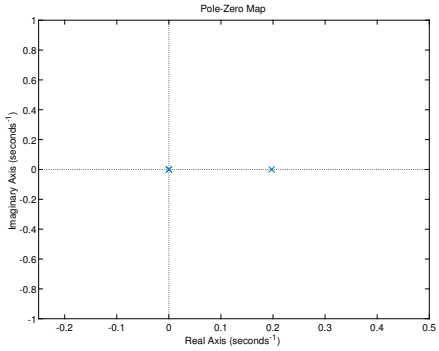


Figure 5.1: The open loop system is unstable.

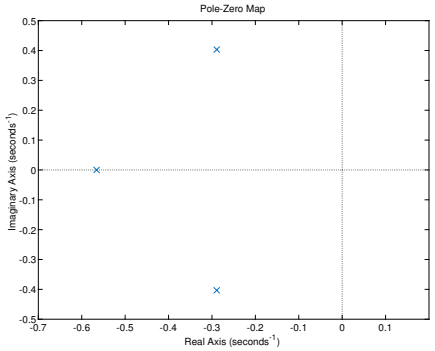


Figure 5.2: The LQR gains shifted the poles to the left half side of the complex plane.

It is suggested to begin evaluating the LQR by placing the trailer with an offset from zero and seeing if the designed gains actually drive the state errors to zero. In other words, the reference value given to the controller is zero. Figure 5.3 demonstrates what this controller looks like. For quickly evaluating the controller performance with only offsets, it was found reasonable to use the linear equations as the plant. The trailer lateral position and orientation should converge back to a horizontal line. When evaluating the performance on a path, there will be changing reference points and the responses will require use of the non-linear equations as the plant.

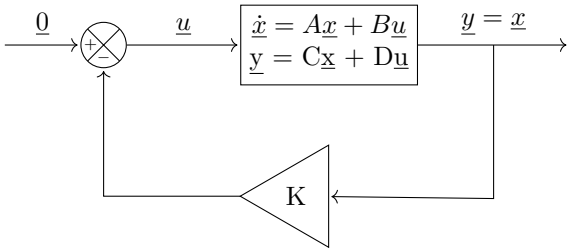


Figure 5.3: The LQR architecture when the reference point set to zero.

Evaluating the control gains using the linear equations for a trailer lateral offset of $1m$ is shown below in Figure 5.4. The controller steers such that the tractor first pushes the trailer closer to the reference line, which actually involves orienting the tractor in the opposite heading direction of the trailer. As the trailer lateral error is minimized, the tractor aligns the heading with the trailer to simply push it along the reference path.

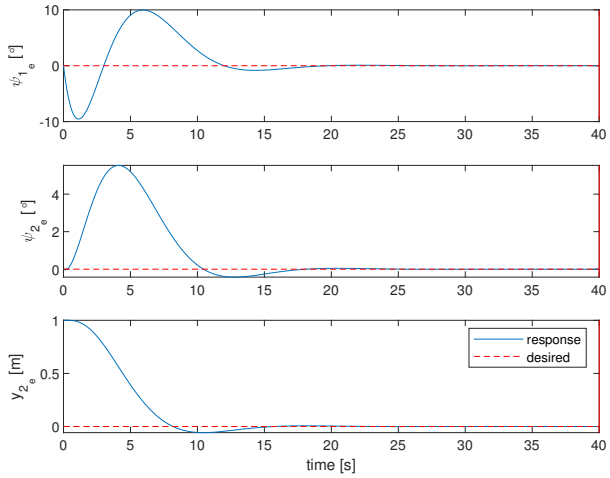


Figure 5.4: Given an offset of $1m$ as the trailer initial condition, the LQR drives all three states to zero. The reference path here can be thought of as a straight line.

This validates the process of designing the gains for the system before implementing the non-linear equations and taking into account the rotation matrices for determining the error from a reference path.

5.1.1 Bryson's Rule

Although a decent response resulted from not altering the Q and R matrices, it is recommended to use Bryson's rule as a starting place for tuning. The idea is to set the maximum allowable deviations for each state and control inputs. Penalizing the errors as such assists with the mixture of units because it scales the variables that appear in the LQR cost function so that the maximum acceptable value for each term is 1.0 [54].

$$Q_{ii} = \frac{1}{max\ acceptable\ value\ of\ z_i^2} \tag{5.13}$$

$$R_{ii} = \frac{1}{max\ acceptable\ value\ of\ u_i^2} \tag{5.14}$$

The goal criteria is for the distance of the trailer to be within $15cm$ and the heading of the trailer to be within ˜5°. The steering angle is saturated at 45°. Surprisingly, using these values performed much worse than not tuning the gains. Further tuning the angles to be less than 2° and the distance to be less than $10cm$ resulted in the best performance. This is likely due to the controller actually being

able to do better along the path. One would think that designing for a larger allowable deviation of the tractor angle would be fine, but surprisingly this always resulted in poorer performance.

$$Q = \begin{bmatrix} 820.7016 & 0 & 0 \\ 0 & 820.7016 & 0 \\ 0 & 0 & 100.0000 \end{bmatrix} \quad R = \begin{bmatrix} 1.6211 \end{bmatrix} \tag{5.15}$$

Solving for the gains again using MATLAB's $lqr()$ function results in:

$$K = \begin{bmatrix} -24.7561 & 94.6538 & -7.8540 \end{bmatrix} \tag{5.16}$$

Figure 5.5 displays the step responses of both gains designed with and without Bryson's rule.

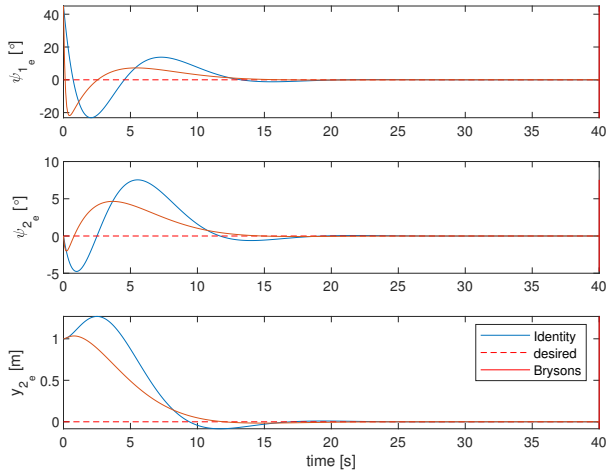


Figure 5.5: Given an offset of $1m$ as the trailer initial condition, the gains are compared between not tuning the controller and with Bryson's rule. Bryson's rule results in a quicker, damped response.

The gains designed with the Q and R as identity matrices result in an overshoot 8.39% for the lateral position error and noticeably responds slower than Bryson's rule. The gains designed with Bryson's rule resulted in a damped response for all three states. Despite the step response improving, the real test is whether or not the controller performs better on an actual reference path.

5.2 Modern Controller Implementation

Now when following a course, one must use the frame rotations outlined in the path planning chapter. It is now suggested to use the nonlinear equations as the plant to allow for changes greater than small angle approximations. The linearized system should just be used for determining the gains.

Simulink was used to evaluate the nonlinear system equations at every timestep. The desired states were retrieved from the $get\_closest\_index()$ function. The desired states are in the global frame. The current states are also in global coordinates, which means an error can be calculated. Even though the x-coordinate is not a state, it is still needed to apply the direction cosine matrix for the frame rotation. In order to prevent any angle values causing problems when looping around 0 and 2π, the $safe\_minus()$ function was used.

The Simulink controller implementation is found in Figure 5.6, where the main control loop is bounded in the red box. There is a red cross of the additional term that is summed after the controller output because this is the feedforward term. This is beneficial, but not necessary–so it will be discussed in a later section.

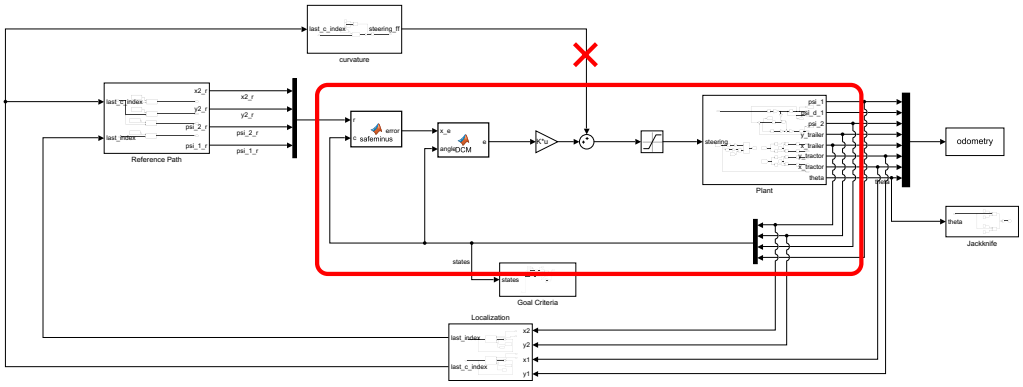


Figure 5.6: The main control loop can be found in the Simulink implementation bounded by the red box.

Whenever the reference values are non-zero, the controller gains must be after the summation junction instead of on the feedback like in figure 5.3. Technically now, the controller is the Linear Quadratic Tracker since new set points are given. This can be thought of as a controller that is regulating the error to zero about new reference values. Given that the Linear Quadratic Regulator is a commonly understood controller, the remainder of this paper will continue to describe the control method as the LQR.

The reference path is determined ahead of time using the modified Dubins curves described in Chapter 4. The reference path is fixed, however, the reference values are generated by determining the track vectors' closest index to the current state from the localization subsystem. The reference values are like a fake rabbit that the greyhounds chase around the track. Figure 5.7 displays the reference value subsystem in the blue box.

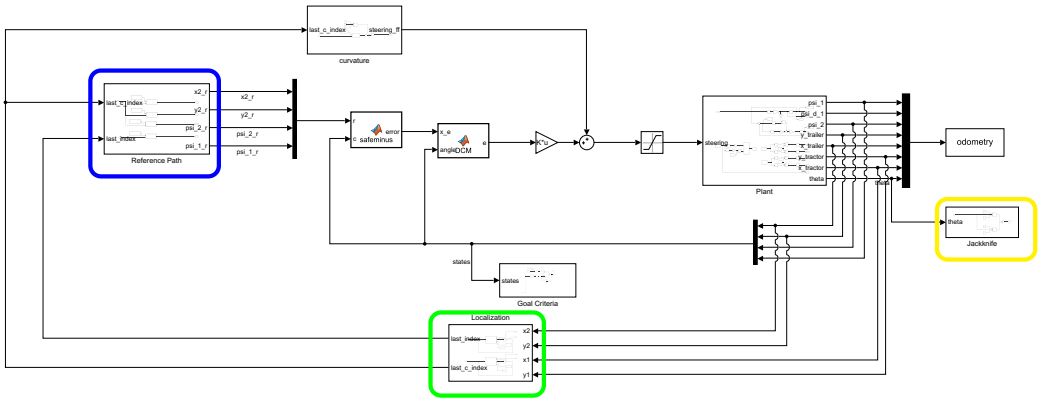


Figure 5.7: The reference path subsystem can be found in the blue box. The goal criteria subsystem is found in the green box. The jackknife condition subsystem is shown in the yellow box.

Variable selector blocks are inside the reference path subsystem. Since the track array is known and fixed, the selected index is used to determine the next reference value for each of the states. This subsystem includes the x-coordinate as well because it is needed to perform the rotation to calculate the correct lateral error term. Figure 5.8 displays how the variable selector blocks are used.

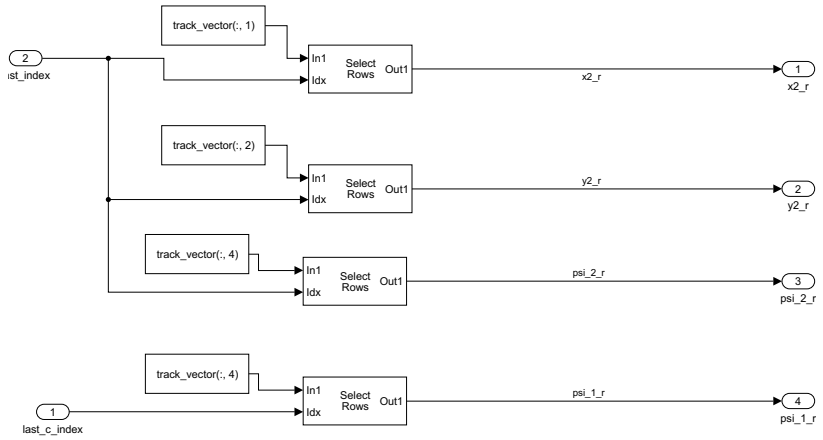


Figure 5.8: The reference path subsystem determines the next reference point values using an index and the array of the fixed path.

The goal criteria is met if the rear-most point of the trailer reaches the goal within $15cm$ and the angle orientation has an error of less than or equal to 0.1 radians. The goal criteria subsystem takes the current states as input to determine if the goal was met; this results in raising the goal flag to high and forces the simulation to stop. The goal pose is the last index of the track array, so the *distance*() MATLAB function block is used to compare the current state to it. A switch block with

the criteria of $15cm$ is used to determine the high or low value of this condition. This condition is fed into a logical AND gate.

The angle orientation error condition is also input to the AND gate as shown in Figure 5.9. The current angle and the loading dock angle are used with $safe\_subtract()$ MATLAB function block to calculate the angle orientation error. If the orientation error is less than 0.1 radians, then the switch will indicate a high value to the AND gate. Lastly, the third input to the AND gate is a condition to prevent the simulation from terminating early if the trailer starts near the goal. A simple example of this occurring is if the reference path is a circle. The logical AND gate will return true only if all of the input conditions are true.

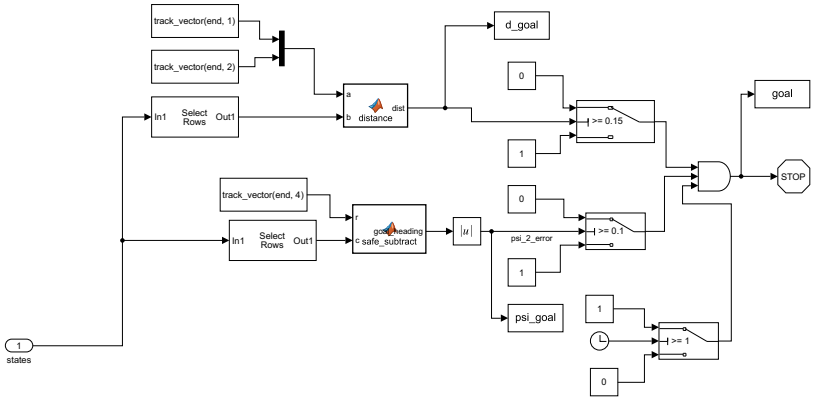


Figure 5.9: The goal criteria subsystem uses switches to determine the goal criteria, which are fed to a logical AND gate.

The jackknife subsystem takes the current hitch angle and determines whether or not the angle between the tractor and trailer exceeded 90° as shown in Figure 5.10. Two switches are used to bound the system to ±90°, ultimately feeding the conditions to a logical OR gate. If either of these conditions are met, the OR gate terminates the simulation.

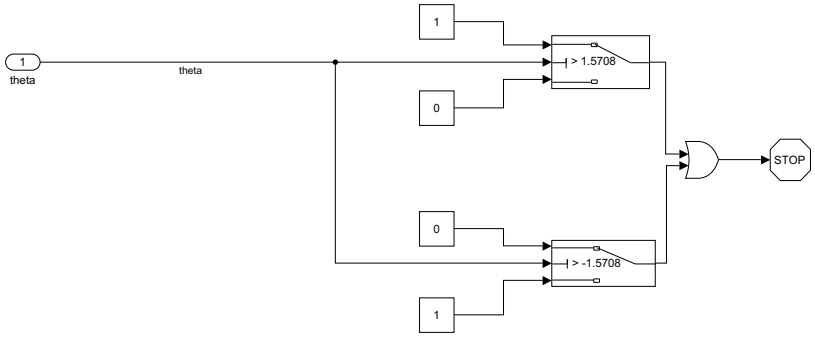


Figure 5.10: The jackknife subsystem terminates the simulation if the logical OR gate has any condition which is true.

Now that the implementation has been discussed, reference paths can now be evaluated. Even though the controller gains designed with Bryson's rule resulted in a better step response, the performance should be evaluated over many tracks with varying transitions from straights and curves.

5.3 Modern Controller Evaluation

Controller gains were evaluated across the same 100 tracks spawned by Dubins Curves. The tracks were stored as individual text files that were read by the *trailerKinematicsLQR.m* file, iterated through a for loop. The left diagram in Figure 5.11 displays the resultant path of the tractor and trailer in an attempt to follow the reference path using the identity matrix for Q and R. The circles represent the initial coordinates and the crosses represent the end of the simulation.

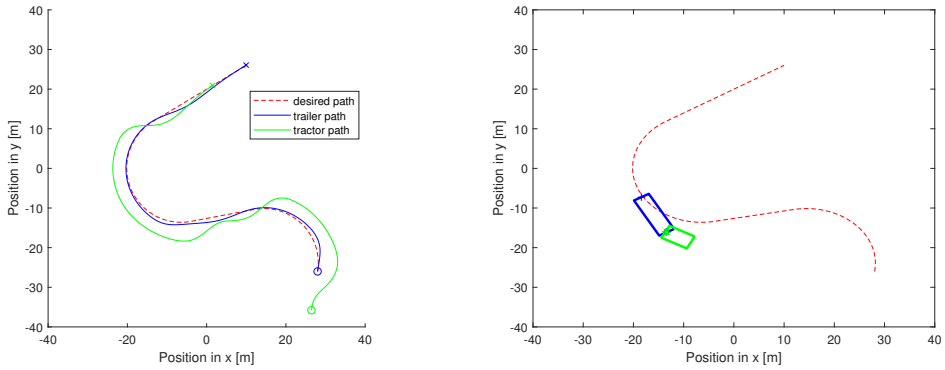


Figure 5.11: Despite giving the tractor a reference orientation along the path, the controller prioritizes the placement of the trailer despite using the identity matrix for Q and R.

The same run is visualized as an animation in the right diagram of Figure 5.11. Considering the tractor lateral position is not a controlled state, it makes sense the trailer pose is prioritized. The reference value given to the controller for the tractor angle orientation is also along the reference path, so it is impressive that this control method reverses the trailer without penalizing the gains for the trailer more.

Since two out of the three states of the system are for the trailer, the controller likely prioritizes it more than the tractor when the gains are not tuned. When using Bryson's rule for designing the gains, a larger allowable angle orientation error of 10° was originally used. The resultant change would decrease the term for the tractor angle orientation error, evidently penalizing the tractor state less. It was theorized the results over 100 paths would reduce the root mean squared error of the trailer states. This was, however, not the case. After exhaustively iterating the penalization, the

tuned Q and R matrices below were determined. Figure 5.12 shows the same path followed using Bryson's rule.

$$Q = \begin{bmatrix} \frac{1}{0.035rad^2} & 0 & 0 \\ 0 & \frac{1}{0.035rad^2} & 0 \\ 0 & 0 & \frac{1}{0.1m^2} \end{bmatrix} \quad R = \begin{bmatrix} \frac{1}{0.785rad^2} \end{bmatrix} \tag{5.17}$$

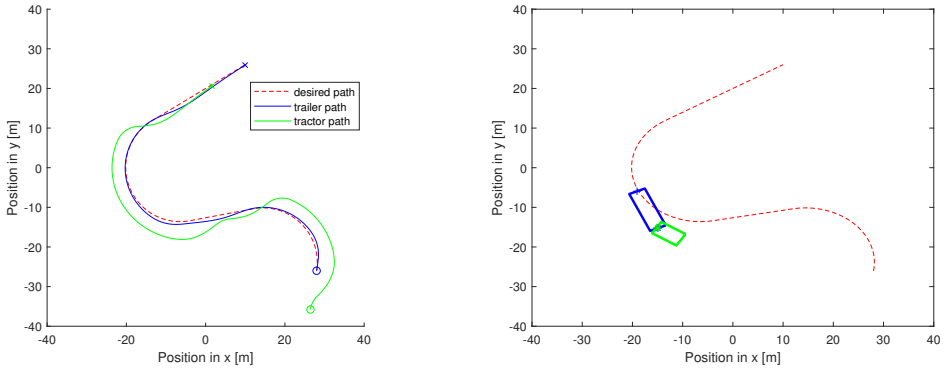


Figure 5.12: Despite giving the tractor a reference orientation along the path, the controller prioritizes the placement of the trailer.

The cost function used in the above cases utilized penalization terms for all three states, which include the heading of the tractor. The tractor heading error is not as important and, in fact, it is expected that the tractor must vary the orientation in order to manipulate the trailer pose along the path. It was theorized that using a $Q_{11} = 0$ would result in a controller that prioritizes the trailer only, but the performance was very similar. The cost function which included all three states already prioritizes the performance of the trailer because the cost is less when both the trailer position and orientation errors are smaller. In fact, values with less strict requirements for Q_{11} did not necessarily improve results. Thus, Q_{11} was set to be equivalent to Q_{33}.

The root mean squared error was used as a metric because it determines a positive, average value that is more easily compared. As shown in Figure 5.13, each of the errors are used to determine three root mean squared errors for the path. All of the tracks are evaluated and the average of these are displayed in Table 5.2.

It is interesting to note that the root mean squared errors tended to be higher than the maximum allowable deviation set using Brsyon's rule, however, this was still an improvement for the metrics shown below. More importantly, the controller designed with Bryson's rule achieved the goal criteria more than not tuning the gains.

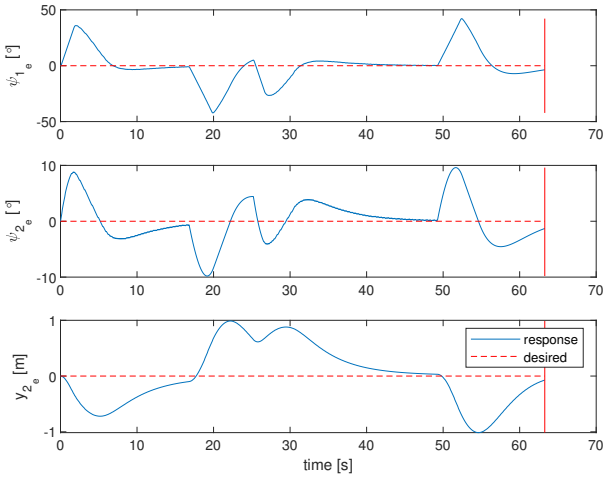


Figure 5.13: Results from each path are recorded and the root mean squared error for each of the three states are calculated.

Table 5.2: This table is the performance with identity and Bryson's Q and R at 80ms.

Metric	Identity	Brysons
rms ψ_{1e} $[rad]$	0.3707 ± 0.1219	0.3226 ± 0.1252
rms ψ_{2e} $[rad]$	0.1457 ± 0.0874	0.1075 ± 0.0837
rms y_{2e} $[m]$	0.7973 ± 0.3504	0.5512 ± 0.2808
max ψ_{1e} $[rad]$	0.8268 ± 0.2720	0.7722 ± 0.2438
max ψ_{2e} $[rad]$	0.2675 ± 0.1307	0.2070 ± 0.1136
max y_{2e} $[m]$	1.4549 ± 0.6936	1.0597 ± 0.8749
bool(goal)	76	82

Notice that the controller does not achieve the goal condition 100% of the time. This was found primarily due to the LQR's inability to impose constraints. As shown in Figure 5.14, the system would jackknife in tight transitions between curves.

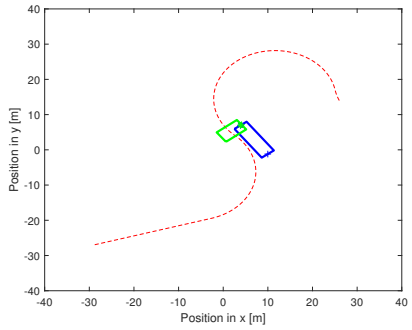


Figure 5.14: The angle between the tractor and trailer would exceed $\pm 90°$ in tight transitions between curves. The controller would ask for too much because there is not the ability to impose constraints with the LQR.

A few ways to mitigate this problem may be to continually plan a few seconds ahead of time or to use Model Predictive Control (MPC) to generate realistic trajectories that impose the constraint of the hitch angle with new gains at every timestep. Control accuracy can also be improved using curvature as the feedforward input.

5.3.1 Curvature as Feedforward

Feedforward can be thought of as an open loop input that is summed after the control input, $\underline{u}$. In theory, the feedforward command assists with known disturbances. It increases the control input by anticipating the desired response. Murray in [45], describes the use of feedforward and feedback in control as a two degree of freedom controller.

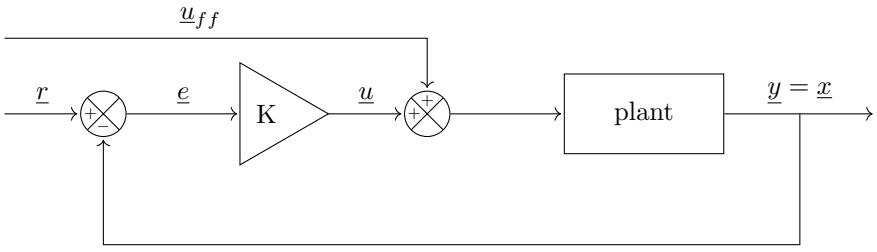


Figure 5.15: The feedforward input gets summed after the control input to account for disturbances

With regard to autonomous vehicle control of kinematic models, the curvature multiplied by the wheelbase is often used due to Ackermann steering [43]. This feedforward command is the same command that was used in open loop testing, $\delta = \kappa L$. Applying it to reversing a trailer works best with the curvature calculated for the tractor, but multiplied by -1.0. Recall that menger curvature

is used to calculate this value ahead of time for each new track, so the last index of the cab is used since that information is readily available.

Figure 5.16 displays where the curvature subsystem is found in the Simulink model. Figure 5.17 shows the curvature subsystem, where a variable selector block is used to determine the correct curvature value from the predefined track vector. The curvature is multiplied by the length of the tractor, L_1. The feedforward value is determined to be positive or negative depending on if the system is driving forward or reverse.

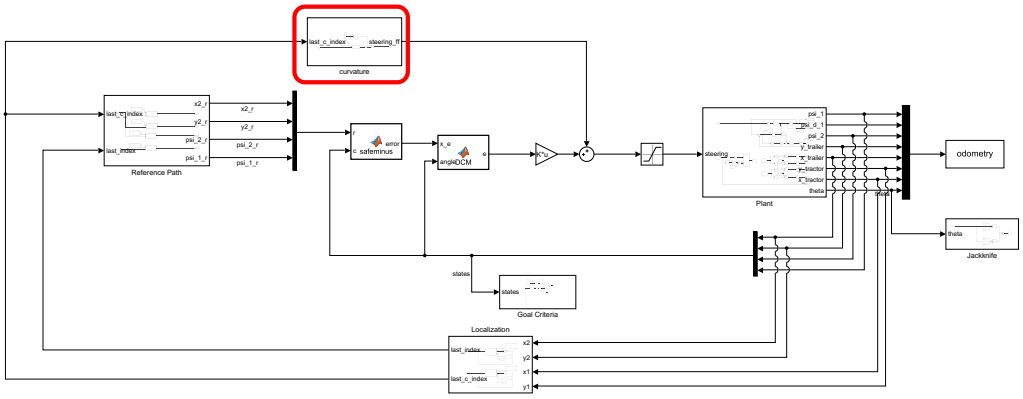


Figure 5.16: The curvature subsystem is located in the red box.

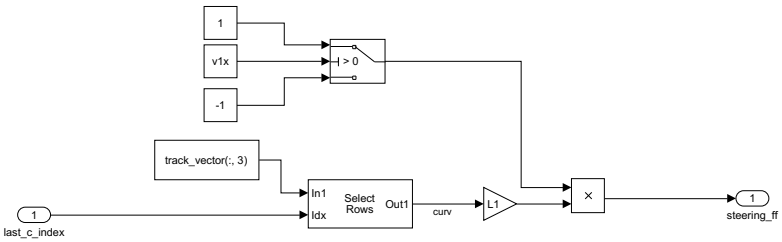


Figure 5.17: Inside the curvature subsystem is a variable selector to determine the predefined curvature at each step based on the closest cab index.

As shown in Table 5.3, the feedforward term decreases the metric averages by less than 15% for the identity column and 3% for the Bryson's column. It unfortunately does not increase the amount of times the controller makes it to the goal.

Table 5.3: This table shows the performance with identity and Bryson's Q and R at 80ms and with curvature.

Metric	Identity	Brysons
rms ψ_{1e} $[rad]$	0.3508 ± 0.1219	0.3205 ± 0.1252
rms ψ_{2e} $[rad]$	0.1335 ± 0.0873	0.1062 ± 0.0835
rms y_{2e} $[m]$	0.6799 ± 0.3729	0.5371 ± 0.2792
max ψ_{1e} $[rad]$	0.7926 ± 0.2646	0.7674 ± 0.2426
max ψ_{2e} $[rad]$	0.2595 ± 0.1207	0.2051 ± 0.1123
max y_{2e} $[m]$	1.3515 ± 1.4492	1.0360 ± 0.8921
bool(goal)	76	82

Although feedforward is beneficial, it was ultimately decided to not be used in the comparison of modern controls and reinforcement learning in this thesis. Initial training utilized curvature as an additional input to the neural network, however, it was soon decided that it could potentially make training intractable. In addition, it does not make sense to send an estimate of steering based on the tractor only.

5.4 Summary

In summary, this chapter discussed the design, implementation, and evaluation of the Linear Quadratic Regulator in MATLAB and Simulink. The gains that were selected were fine tuned with Bryson's rule to obtain a baseline control method to compare against reinforcement learning.

Chapter 6

REINFORCEMENT LEARNING

This section covers the design, implementation, and evaluation of reinforcement learning for the agent in the TruckBackerUpper-v0 environment. The design consisted of the original problem formulation of selecting states and actions to dictate which algorithm to choose. The Deep Deterministic Policy gradient (DDPG) was ultimately chosen to be the reinforcement algorithm due to its demonstrated usefulness for continuous, deterministic actions. The DDPG was first implemented on a toy example called the ContinuousMountainCar-v0, which is provided by OpenAI gym. A discussion of the implementation on a toy example can be found in appendix B.

It is usually challenging to debug reinforcement learning implementations because coding bugs can come from either the function approximator or the reinforcement learning algorithm. Reward functions are problem specific and thus it is usually recommended to get the RL algorithm to work on a toy example first. The number of neurons, layers, learning rates, and regularizers are all hyperparameters that can be tuned for the NN architecture. The RL algorithms also have tunable parameters such as exploration methods, the discount factor, and the reward function. Lillicrap [67] stated that their algorithm worked for over 20 different continuous control environments.

One major difference that was discovered between Lillicrap's description and the implementation on the ContinuousMountainCar-v0 environment was that the L_2 regularizer impeded the speed of learning and performance. The L_2 regularizer is known as a weight decay where a scalar is multiplied against the sum of weights. This is then added to the loss function to penalize the neural network for high weights. Solving the toy problem provided confidence that the machine learning hyperparameters and the reinforcement learning implementation was sufficient. After this, the design items that needed focus included the reward function, terminal conditions, exploration, and convergence criteria. Despite being design related topics, the process of determining these were the result of constant iteration since no prescriptive design criteria exist.

The implementation utilized Tensorflow 1.12 in order to exercise the use of parallelized computations provided by a GPU. Some early training clocked at two and a half days for $20,000$ episodes, but this was due to training on pure noise for exploration. More will be discussed about this, however, the takeaway is that a GPU tremendously assists with speeding up computation. A desktop with an Nvidia GEFORCE GTX 1080 Ti GPU with 3584 CUDA cores and 11Gb of RAM was primarily used in this work.

The Google Compute Engine was also used as a cloud service to outsource the computations using an Nvidia Tesla P100 GPU which also consisted of 3584 CUDA cores, but 16Gb of RAM. At any one point, three instances were at the disposal to run multiple training sessions with minor parameters tweaked. Google Computer is a fee-based resource and this work incurred expenses of over $1000 with Google Compute, despite them offering a $300 trial period. Having one local machine is extremely useful, but it is highly recommended to augment the work with multiple instances for simultaneous studies in the background. Tensorflow offers many API for creating layers to train a Neural Network, however, this chapter will discuss important implementation insights. One of those insights resulted in this work actually writing each layer manually. The implementation topics will include random seeds, batch normalization, and GPU CUDA cores.

An idea of the performance desired was known due to first designing and evaluating the LQR. The evaluation of the DDPG algorithm was really two-fold. First, training was not stopped until it was considered converged. It is difficult to quantify because a reward function should indicate when the performance is good. However, during development, the reward scheme was constantly being iterated. Once convergence is achieved, the second item was to test the NN on the environment. During training, the weights are constantly changing at every timestep. Testing takes the fixed weights and evaluates performance on problem metrics.

Machine learning is stochastic by nature, meaning that training with some random initial seeds will result in one set of weights–but a different initial seed will result in a different set of weights. In other words, if one runs the same code twice, there will be drastically different results! To handle the obfuscation of variation, multiple seed trials should be used to make any evaluation. Random numbers in computers are not truly random, so setting the initial seed should provide a deterministic sequence of numbers. According to Colas in [77], reproducing results for machine learning has become a serious problem. Some people report on performance using five different seeds, but others cherry pick the best results to report. Running multiple seeds takes time and arguably, in the case with Google Compute, can result in significant costs. This thesis will report on three different seeds, but perform the comparison on the best weights and biases from a specific initial seed.

6.1 Reinforcement Learning Design

6.1.1 RL Paradigm

The reinforcement learning paradigm consists of an agent which interacts with an environment to learn a control policy as shown in Figure 6.1. The agent trains by continually trying to maximize

the expected reward in an episodic manner. An episode is a sequence of states, actions, and rewards which ends on a terminal state. The design of the agent uses one of the many reinforcement learning algorithms to update the neural network which represents the control policy.

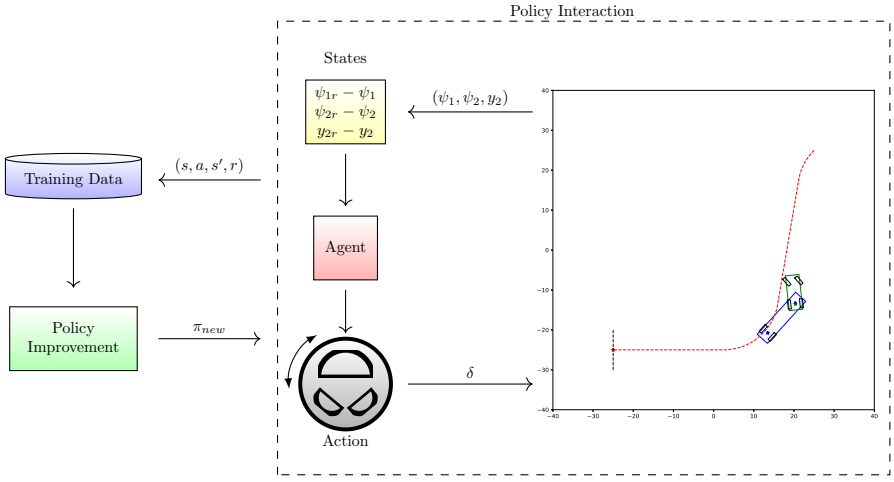


Figure 6.1: The agent takes the error states from the tractor-trailer pose from the reference path and determines the action, which is the steering angle. These state transitions are used to update the policy.

Reinforcement learning is different than supervised learning in the sense that a reward is used for updating the neural network instead of hand labeled datasets. Supervised learning will usually train over many epochs, which are a complete forward and backward passes of all the training examples. Since reinforcement learning will produce more training data from interacting with an environment every episode, epochs are not necessarily used. Instead the neural network may be updated at the end of an episode or at every timestep using a batch randomly selected from the history. The DDPG algorithm updates the neural network at every timestep, for a user set amount of episodes.

Using a neural network as a nonlinear function approximator literally creates an approximation of the state space. The DDPG algorithm is a model free algorithm because it does not use a mathematical model to plan or determine its weights. It does, however, use interaction with the environment to determine a mapping of input to output.

The inputs into the NN are the same as the LQR controller, which are the errors from the path suggested by Figure 6.2. This can be thought of as the same block diagram of a control system as shown in Figure 6.3. The desired values are subtracted by feedback of the current values. The error states include tractor heading, ψ_{1e}, trailer heading, ψ_{2e}, and trailer lateral position, y_{2e}.

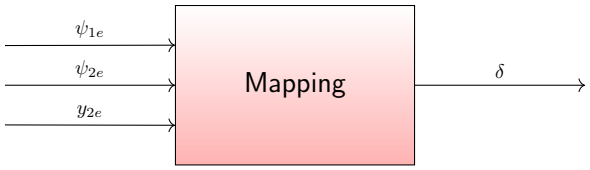


Figure 6.2: The controllers are an input-output mapping of states to actions

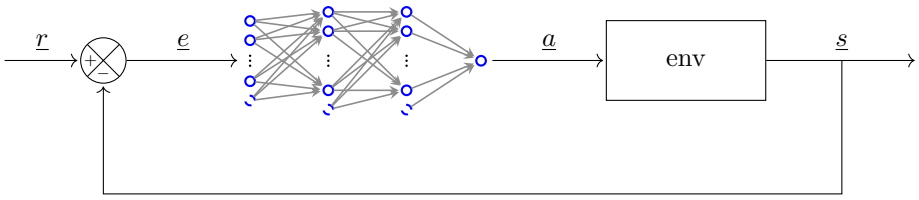


Figure 6.3: The inputs to the NN are the error from the path just as in the modern controls implementation. Considering there is feedback, the NN is trained to act as a closed loop controller.

Figure 6.3 is the desired functional control outcome once the reinforcement learning agent has been trained. This means that after training, the reward function is no longer used. A neural network takes the inputs and determines the next action; this is the control policy in the DDPG algorithm.

6.1.2 Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG) algorithm stems from David Silver's Deterministic Policy Gradient (DPG) algorithm in [78]. The DPG algorithm is an actor-critic approach which allows for continuous, deterministic actions which benefit from both policy based and value based reinforcement learning methods. The actor network, $\mu(s|\theta^{\mu})$, is the control policy and the critic, $Q(s,a)$, provides a state-action value of how well the actor network performs.

Usually with value based methods, a state-action value is only used for discrete action sets. A state-action value is determined for each of the discrete actions, where the maximum Q-value is the selected policy. Applying value based methods to a continuous action space would require a large, time consuming optimization every timestep to find the greedy policy. Instead now with DPG or DDPG, the action taken by the actor is fed into the critic to critique the Q-value of the given states and actions. The critic is learned using the Bellman equation as in Q-learning.

$$Q(s,a) = r + \gamma max_{a'} Q(s',a') \tag{6.1}$$

Policy gradient algorithms are the more popular class of continuous action action reinforcement learning algorithms. Without the critic, they suffer from high variance and being terribly sample inefficient. Initial work of including a critic produced the Stochastic Policy Gradient (SPG) actor-critic algorithms. David Silver showed the DPG algorithm could be considered as a limiting case of the SPG algorithms. He proved that updating the actor by applying the chain rule to the expected return from the start distribution J with with respect to the actor parameters was, in fact, the policy gradient.

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a, \theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s, \theta^\mu)|_{s_i} \tag{6.2}$$

The equation above is the policy gradient, which is used for updating the weights and biases of the actor policy using gradient ascent.

6.1.3 Key Advances from DQN

As with Q-learning, using non-linear function approximators means that convergence is no longer guaranteed. The Deep Deterministic Policy Gradient (DDPG) improves upon the Deterministic Policy Gradient (DPG) by two major changes inspired by the Deep Q-Network (DQN) paper [28]. The first of those changes include the use of a replay buffer and the second is using target networks.

The replay buffer, also known as experience replay, is a history of sequences that the algorithm can randomly sample from. The sequences consist of the state transitions, action taken, and associated reward–(s_t, a_t, r_t, s_{t+1}). According to Lillicrap's work, the replay buffer size used was 10^6. Whenever the replay buffer was full, the oldest samples were popped off the stack.

The DDPG algorithm randomly samples a batch of sequences from the replay buffer to perform updates to the neural networks through backpropagation. The minibatch size is 64 at each timestep. The effect of using a replay buffer makes training more similar to supervised learning, which minimizes the correlation between samples. If one were to train with only one sequence at a time, the algorithm would be prone to forgetting good policies or unlearn. Since the DDPG is an off-policy algorithm, the replay buffer can be used.

The target network is copy of the online network used for determining the target values or labels used for backpropagation. The online network is the neural network where predictions are made for exploring and exploiting, where the parameters are updated constantly. Without the target network, the $Q(s, a|\theta^Q)$ network that is being updated is also used for calculating the target value, y_i. The

update minimizes the loss function $L = \frac{1}{N}\sum_i (Q(s_i, a_i, \theta^Q) - y_i)^2$, therefore, is prone to divergence. It can be similarly thought of as a dog chasing its tail since the weights and biases would get updated using predictions from itself.

$$y_i = r_i + \gamma Q(s_{i+1}, \mu(s_{i+1}, \theta^\mu), \theta^Q) \tag{6.3}$$

Using a copy of the networks provides consistent targets during the training at each timestep of the episode. The target networks are denoted using the prime symbol. Both the critic and actor networks benefit from having target networks, meaning a total of four networks are used in training the DDPG algorithm. The critic target network is $Q'(s, a|\theta^Q)$ and the actor target network is $\mu'(s|\theta^{\mu'})$. The modified critic target would, therefore, appear as below:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}, \theta^{\mu'}), \theta^{Q'}) \tag{6.4}$$

DQN implementations copy the weights over from the online network to the target network every $10,000$ iterations. This can sometimes be a disruption to training. Lillicrap's DDPG implementation instead uses soft target updates using polyak averaging with $\tau = 0.001$. The target values are constrained to change slowly, vastly improving the stability of learning. The slow updates are performed to both the actor and critic target networks.

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \tag{6.5}$$

It has been found that the usage of the replay buffer and target networks greatly stabilizes learning for temporal difference, off-policy algorithms with neural networks as their function approximators. These improvements are what differentiate DPG from DDPG.

6.1.4 Exploration Noise

There is a trade-off between exploration and exploitation in reinforcement learning. To obtain more reward, a reinforcement learning agent must prefer actions (exploit) that it has tried in the past that have proven to obtain a higher reward. However, in order to discover these actions, it has to try (explore) actions it hasn't before.

A typical exploration approach is the ϵ-greedy policy with discrete actions. A typical value of ϵ is 0.1. Most of the time or $1-\epsilon$, the agent chooses the discrete action that has the maximal estimated state-action value. The agent will explore the remaining ϵ, such as 10% of the time. Exploring in discrete action spaces simply means to select a random discrete action. It is typically suggested to decay the amount of exploring logarithmically over time.

Since the DDPG algorithm uses continuous actions, exploration is instead injected as stochastic noise to the action using Ornstein & Uhlenbeck [68]. The method adds some momentum to the noise so the agent could learn following some action path for a while. This works because the DDPG algorithm is off-policy. Figure 6.4 shows an example of the exploration noise.

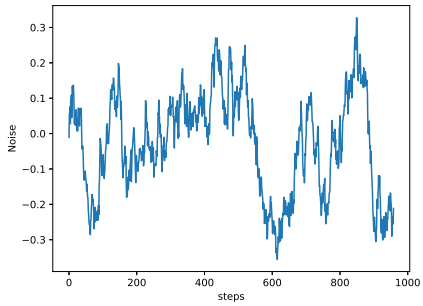


Figure 6.4: The Ornstein-Uhlenbeck process is a stochastic process which is centered about a mean.

The Ornstein-Uhlenbeck process is a method to add some stochastic noise to a deterministic, continuous policy. The noise will hover around a selected mean, μ. For steering the tractor-trailer, it makes sense to set $\mu = 0$.

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \tag{6.6}$$

Lillicrap's paper suggested they used $\theta = 0.15$ and $\sigma = 0.2$. The θ refers to how fast the noise reverts towards the mean or the inertia. The σ refers to the variance of the noise. The dW_t is some random normal noise. The variables describing the Ornstein-Uhlenbeck noise is not to be confused with those of the neural networks. The important take-away is that the noise, N_t, is added to the output of the actor network.

$$a \leftarrow \mu(s_t, \theta^\mu) + N_t \tag{6.7}$$

Since the noise hovers around a mean of zero, the agent will explore only when the noise is non-zero.

6.1.5 Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient can be summarized in the following pseudo algorithm 8. First the critic network, $Q(s|a, \theta^Q)$, gets randomly initialized with weights and biases θ^Q. Then the actor network, $\mu(s, \theta^\mu)$, gets initialized with a different set of weights and biases θ^μ. Both the target networks, Q' and μ', gets the weights copied over from the online networks. A replay buffer, B, of selected size gets created to a specific size, waiting to be populated with sample transitions.

Algorithm 8 Deep Deterministic Policy Gradient

Randomly initialize critic network $Q(s|a, \theta^Q)$ with weights θ^Q
Randomly initialize actor network $\mu(s, \theta^\mu)$ with weights θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer B

for n episodes **do**
  Initialize a random process N for action exploration
  Observe initial s
  for each step of episode **do**
    $a \leftarrow \mu(s_t, \theta^\mu) + N_t$
    Take action a, observe reward r and next state s'
    Store transition (s_t, a_t, r_t, s_{t+1}) in B
    Sample a random minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from B
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}, \theta^{\mu'}), \theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (Q(s_i, a_i), \theta^Q) - y_i)^2$
    Update the actor policy using the sampled policy gradient:
$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a, \theta^Q)|_{s=s_i, a=\mu(s_i)}) \nabla_{\theta^\mu} \mu(s, \theta^\mu)|_{s_i}$$
    Update target critic network, $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    Update target actor network, $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$

For n episodes or while the agent's training hasn't converged, the following process loops. At the start of an episode, the exploration noise from Ornstein-Uhlenbeck is randomly re-initialized. Also at the beginning of the episode, the initial states are observed. Then a for loop for the maximum number of steps per episode begins, starting with the first action predicted by the actor network combined with the exploration noise. The initial outputs of the networks are nearly zero due to how Lillicrap specified the weight and bias initialization. This means that the beginning of training is overpowered by the noise in order to not propagate the training in one direction or another due to the random beginning weights.

The action is then taken in the environment, returning the reward and the next state. The transition is stored into the replay buffer, B. If the replay buffer does not have at least the batch size or if a warm-up period has not been reached, then the agent will continue to interact with the environment and not make updates to the weights. Once the algorithm is allowed to sample from the replay buffer, the critic is trained first using backpropagation. Taking a batch size of say, randomly 64 samples, the online critic network forward propagates or predicts what the output would be.

The labels for each of the critic predictions derives from the Bellman equation, but the key insight is that the target networks are used to determine $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}, \theta^{\mu'}), \theta^{Q'})$. The mean squared loss for each of the batch samples are calculated from $L = \frac{1}{N}\sum_i (Q(s_i, a_i, \theta^Q) - y_i)^2$. The loss is then used to calculated the partial derivatives and the chain rule is used across all the layers in order to change the weights and biases for the critic network. The target networks are only

used for updating the critic network to provide consistent target values to prevent the critic training to diverge.

Next the online actor network takes the same sample batch and predicts what the actions are. The gradient of the Q-values from the online critic network with respect to the predicted actions are then calculated. The actor network gradients are then determined using the action predictions, online actor network variables, and the negative gradient of the Q-values with respect to the actions. The reason the negative gradient is used is so that it minimizes the loss, in other words, it performs gradient ascent.

There isn't really a loss for updating the weights of the actor network. Instead, the idea is to update the neural network in the direction of the maximizing the expected return; this is the policy gradient. This can also be thought of as maximizing the Q-function, which makes sense as to why the critics usually need to be trained first. This update reinforces good policies and punishes bad policies. Figure 6.5 visually displays the agent in the DDPG algorithm.

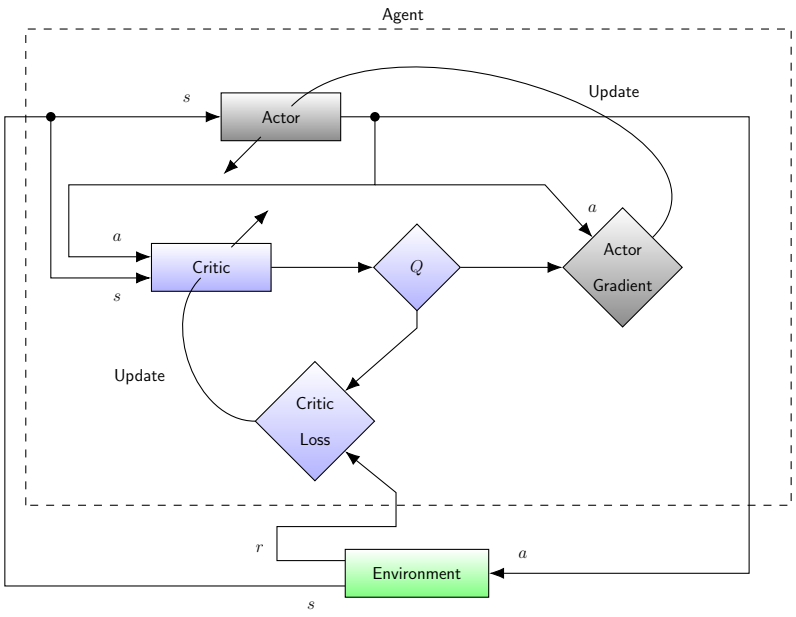


Figure 6.5: The DDPG algorithm conveniently uses continuous actions from the actor and the a critic provides a measure of how well it did; hence the interplay between the actor and critic

After changing the weights of the actor, the target critic network is updated slowly using $\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$. Immediately after, the target actor network is also updated using $\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$. Once this finishes, then another step of the current episode continues on to interact with the environment and update the networks accordingly. Once the episode is terminated by a certain condition or the maximum steps have been taken, the next episode begins. The noise and initial

conditions reset. The process repeats itself until the number of episodes run out or training has been deemed converged according to a convergence criteria discussed later.

6.1.6 Critic

The critic network takes inputs of the state and action to output a Q-value. The action is not included until the second hidden layer. The states are the currently observed errors from the path, so a vector of shape $(3, 1)$. There are no neurons associated with the input layer or the 0^{th} layer.

$$\underline{s} = \begin{bmatrix} \psi_{1e} \\ \psi_{2e} \\ y_{2e} \end{bmatrix} \tag{6.8}$$

There is only a single action for the constant velocity tractor-trailer, which is the steering angle, δ. The first layer has 400 neurons, where the weights and biases are initialized from uniform distributions of $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer. The fan-in is simply the number of outputs from the preceding layer.

The activation function is a rectified linear unit or RELU which has the non-linear properties of $\sigma = max(0, z)$. Recall that z is the output of the previous layer. Figure 6.6 displays what a RELU activation function looks like.

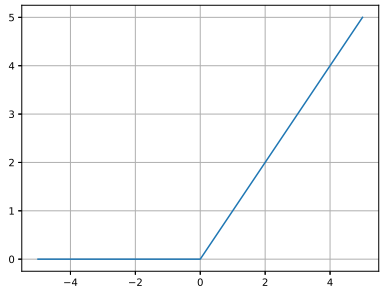


Figure 6.6: The RELU activation function is also known as a ramp function that has really been popularized for training deeper networks.

The computation of the first layer of the critic network is thus:

$$\underline{h}_1 = relu(W_1 \, \underline{s} + \underline{b}_1) \tag{6.9}$$

The second layer has 300 neurons, where all the weights and biases are similarly initialized with the fan-in. The layer also uses the RELU for the activation function. What is mathematically different

about this layer is that the action is augmented to look like the following:

$$\underline{h}_2 = relu(W_2\,\underline{h}_1 + W_2\,\underline{a} + \underline{b}_2) \tag{6.10}$$

The output layer has one neuron because there is only one action. The weights and biases of the third layer are initialized from a uniform distribution of $[-.003, .003]$ so that the output begins with initial estimates near zero. The output has a linear activation function.

$$\underline{Q} = W_3\,\underline{h}_2 + \underline{b}_3 \tag{6.11}$$

Figure 6.7 displays the critic network with the correct states and the action applied to the second layer. In Lillicrap's original work, batch normalization was used on the first layer, but was not used in this work as depicted in red in Table 6.1. This will be discussed in the implementation section. As mentioned before, the L_2 regularizer was not found to be helpful which is also shown in red.

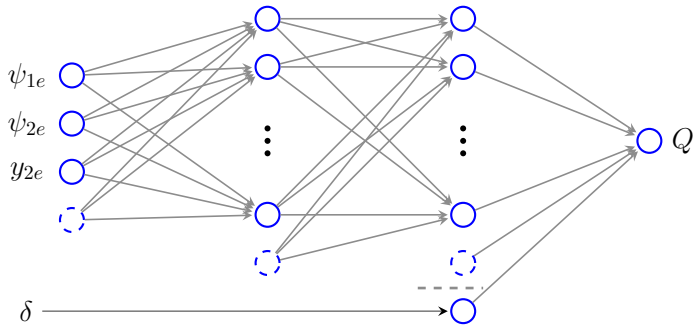


Figure 6.7: The action of the actor is fed into the critic, but at the second hidden layer.

Table 6.1: Critic Hyperparameters

Hyperparameter	Inputs	Layer 1	Layer 2	Outputs
# Neurons	3	400	300	1
Activation	None	RELU	RELU	linear
Initializer	None	$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$	$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$	[-.003, .003]
Batch Normalization	None	None	None	None
L_2 Regularizer	None	None	None	None

The learning rate for the critic is chosen as $\alpha_c = .001$. The discount factor is selected as $\gamma = 0.99$. With maximizing the future expected return, the target label is $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}, \theta^{\mu'}), \theta^{Q'})$. The weights and biases of the critic, therefore, get updated as the following:

$$\theta^Q \leftarrow \theta^Q - \alpha_c \sum_i \nabla Q(s_i, a_i, \theta^Q)(Q(s_i, a_i, \theta^Q) - y_i) \tag{6.12}$$

6.1.7 Actor

The actor networks take inputs of the state in order to predict the next action as shown in Figure 2.27. The states have the same shape as the critic since it is the same information. Lillicrap's original algorithm uses batch normalization in all the layers, including the input. Table 6.2 highlights in red where these were removed.

The first layer has 400 neurons and the second has 300. Both layers initialize the weights and biases using the fan-in procedure. They also both use the RELU activation function. The output layer initializes the weights and biases within a uniform distribution of $[-.003, .003]$ so that the initial output is nearly zero. The output layer has a single neuron with the tanh as the activation function which is demonstrated in Figure 6.8. The hyperbolic tangent activation function outputs values within the range of $(-1, 1)$.

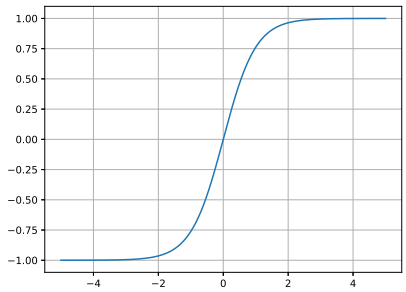


Figure 6.8: The tanh activation function is used to bound the steering angle within the minimum and maximum values.

If scaled with the minimum and maximum steering angle values, the outputs of the actor network are reasonable actions for the policy, π. The maximum steering angle is selected to be $\delta_{max} = 45°$, which is 0.785 radians. The actor network can thus be summarized below:

$$\begin{aligned} \underline{h}_1 &= relu(W_1\,\underline{s} + \underline{b}_1) \\ \underline{h}_2 &= relu(W_2\,\underline{h}_1 + \underline{b}_2) \\ \pi &= \delta_{max} tanh(W_3 \underline{h}_2 + \underline{b}_3) \end{aligned} \tag{6.13}$$

A diagram of the actor neural network in this thesis is depicted in Figure 6.9.

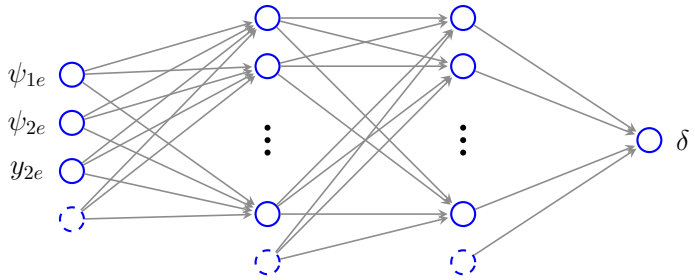


Figure 6.9: The actor is the policy network. Once trained, the actor is simply used as the controller.

Table 6.2: Actor Hyperparameters

Hyperparameter	Inputs	Layer 1	Layer 2	Outputs
# Neurons	3	400	300	1
Activation	None	RELU	RELU	tanh
Initializer	None	$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$	$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$	[-.003, .003]
Batch Normalization	None	None	None	None
L_2 Regularizer	None	None	None	None

Keep in mind that the variables W_i and $\underline{b}_i$ are the trainable parameters. The learning rate for the actor is chosen as $\alpha_a = .0001$. The actor typically learns slower than the critic because the critic is used to judge the policy that is being trained. Changing the weights and biases in the direction of the policy gradient or rather, gradient ascent looks like the following:

$$\theta^{\mu} \leftarrow \theta^{\mu} + \alpha_a \sum_i \nabla_a Q(s, a, \theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s, \theta^{\mu})|_{s_i} \tag{6.14}$$

6.1.8 Batch Normalization

When learning with reinforcement learning algorithms, the various states may have different physical units. The ranges of positions, velocities, and angles can vary across environments. This can make it difficult for the neural networks to learn effectively, making it difficult to generalize across environments with different scales of state values. Furthermore, large state values may create large weight updates. Batch normalization is a machine learning method to circumvent this.

Batch normalization manually scales the features so they are in similar ranges across units and environments. This method normalizes each dimension over the samples in a batch, B, to have a mean, μ_B, of zero and a variance, σ_B^2, of one. During training, this technique is used to minimize

the covariance shift by ensuring that each layer receives a whitened input according to Ioffe and Szegedy [69].

$$\begin{aligned} \mu_B &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma_B^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \end{aligned} \tag{6.15}$$

The output of the batch normalized layer is the normalized to produce $\hat{x}_i$. This is then scaled and shifted to result in the regularized output, y_i. These variables are not to be confused with the reinforcement learning variables. In regards to batch normalization, γ scales the normalized value and β shifts it. The constant, $\epsilon = 0.001$, is added to the variance for numerical stability.

$$\begin{aligned} \hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \end{aligned} \tag{6.16}$$

During testing, however, it uses a stored running average of the mean and standard deviation. With reinforcement learning, the running averages are supposed to be used during exploration and testing. The DDPG algorithm uses batch normalization for the state input and all the layers of the actor network. The algorithm also uses batch normalization for the layer prior to the action input. Batch normalization should speed up convergence of training since the values are smaller, however, more computations do occur which can inadvertently slow the training clock time.

The initial implementation of the DDPG algorithm on the toy example found that batch normalization did speed up the convergence of training by 28.2%, but the average test reward was slightly less than without. More information on this can be found in appendix B. Due to the added complexity of batch normalization, initial training of the tractor-trailer system did not include it. It is investigated again after the reward function is selected.

6.1.9 L_2 Regularizer

Regularization is used in machine learning models to prevent deep learning neural networks from overfitting. The L_2 regularizer is the sum of the squared Euclidean norm of the weight matrix for the layers. The scalar, λ, is used to control the strength of the regularization. The scaled regularized term is then added to the loss, L, such that it penalizes for large weights in stochastic gradient descent. Lillicrap's implementation of the DDPG algorithm uses the L_2 weight decay in the critic with a $\lambda = 0.01$. Technically, the L_2 weight decay is the L_2 regularizer, but with a $\frac{1}{2}$ term according

to Loshchilov [79].

$$L \leftarrow L + \frac{1}{2}\lambda||\theta^Q||^2 \tag{6.17}$$

Taking the gradient of the regularized loss will, therefore, make the weights have a tendency to decay towards zero. It was discovered that using the L_2 weight decay alone did not allow for convergence the initial implementation of the DDPG algorithm on the toy problem. When combining the L_2 weight decay with batch normalization, however, the neural network converged once out of three times. The average test reward was not better than the baseline which did not include the weight decay and batch normalization. More information on this can be found in appendix B.

Therefore, initial training the the tractor-trailer system does not include the L_2 weight decay even though the original algorithm does. Loshchilov's paper suggests that adaptive gradient methods like Adam may not work well with L_2 weight decay due to a changing learning rate. The Adam optimizer has become the default choice for training neural networks instead of simply stochastic gradient descent optimizers. The Adam optimizer was used, but not the L_2 weight decay. For the purpose of this thesis, investigative time was instead spent on more important aspects of training such as the reward function.

6.1.10 Reward Function

Reward functions are unique to problems, so many reward functions were devised and evaluated. Rewards can be positive or negative. They can be continuous or discrete, which is also known as shaped or sparse. A large positive or negative reward is typically awarded for good or bad terminal conditions, in addition to the shaped or sparse amount.

Especially for continuous control problems, one wants shaped rewards so the agent gets an idea of how good the transitions are. Positive rewards encourage the agent to continue accumulating the reward. Positive rewards can, however, cause unexpected behaviors. One gets results for what is incentivized, not necessarily desired.

The cobra effect is a good example of this according to Hammond [80]. The government wanted to reduce the amount of venomous snakes in the area, so they gave a cash handout for whoever brought in a venomous snake to dispose of. People started gaming the system and started breeding venomous snakes so they could receive more cash handouts. This inadvertently caused there to be more venomous snakes in the city. In this regard with positive rewards, the agent may avoid terminals and hover around the goal point so that it can rack up as many reward points as possible.

One should design terminal rewards such that they yield a very high reward–a step reward at the goal.

Negative rewards encourage the agent to reach the terminal state as quickly as possible to avoid accumulating penalties. If one is using negative terminals, it is desirable to usually stick with positive rewards where majority of the episodes end on a positive value. This being said, it is known that having sparse rewards can make it difficult for policy gradient methods to learn. An example of a sparse reward is a step function as shown on the left in Figure 6.10. A shaped reward is displayed on the right.

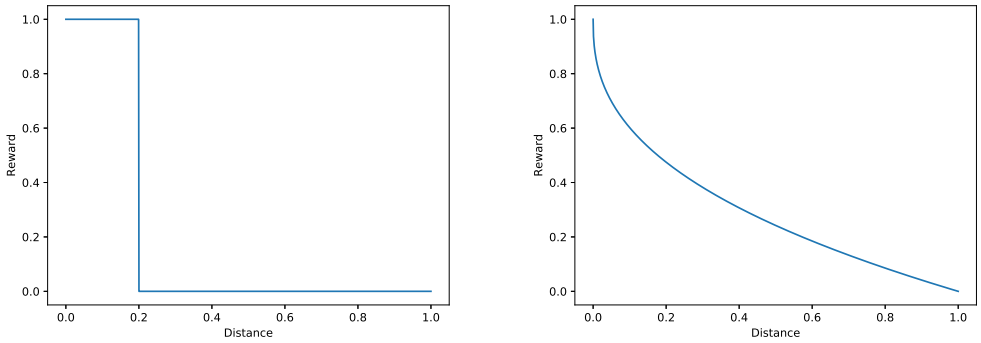


Figure 6.10: A step function is an example of a sparse reward on the left. A shaped function provides a smooth gradient to approach the objective.

It is usually wise to bound the rewards between 1 and -1 so the weight updates are not large. A few of the reward schemes that were investigated for the design of the DDPG algorithm are discussed below. Each of the different rewards lose a reward of a 100 for each terminal condition of jackknifing, going out of bounds, and running out of time. The large amount is outside of the suggested range because this is reserved for providing a clear signal of a terminating condition.

Jackknifing is considered when the angle between the tractor and the trailer exceeds $\pm 90°$. Going out of bounds occurs when the rear axle of the trailer or the rear axle of the tractor falls outside of the $80 \times 80m^2$ environment area. Lastly, each episode is limited to $160.0s$ in simulation time, which equates to a maximum of 2000 steps. The episode is terminated to prevent training on transitions where the agent found a way to stay within the bounds, but will not likely make it to the goal. In summary all of the rewards below are also subject to these conditions:

$$r = \begin{cases} +100 & \text{if} \quad goal == True \\ -100 & \text{if} \quad \theta > 90° \quad \text{or} \quad \theta < -90° \\ -100 & \text{if} \quad (x_2, y_2) \quad \text{exceeds} \quad (\pm 80, \pm 80) \\ & \quad \text{or} \quad (x_1, y_1) \quad \text{exceeds} \quad (\pm 80, \pm 80) \\ -100 & \text{if} \quad t > 160 \\ 0 & \text{else} \end{cases} \tag{6.18}$$

Reward Scheme A

Reward scheme A exhibits a high-sloped reward near the path, and gradually shallows toward zero. The maximum distance is set to $5.0m$ and the maximum angle is set to $45°$. The maximums are consistent for the rest of the rewards schemes. This reward has the ability to provide negative values as well if the error is greater than the maximum. Raising the exponent to a higher value, but still less than one, would result in a more linear shape to the goal. Changing the scalar of 0.5 in front of each of the negative terms would prioritize one over the other. This reward function was heavily influenced by the work of Bonsai in [81] for orienting a robotic arm's joint angles to pick up a block.

$$r = 1.0 - 0.5(\frac{|y_{2e}|}{y_{max}})^{0.4} - 0.5(\frac{|\psi_{2e}|}{\psi_{max}})^{0.4} \tag{6.19}$$

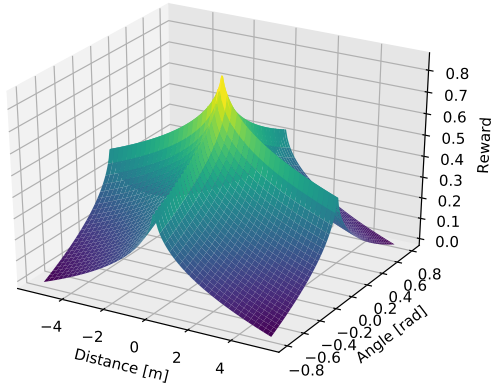


Figure 6.11: Reward scheme A provides a sharp gradual to zero error with respect to the path.

Reward Scheme B

This reward scheme resembles a Gaussian reward function where the reward increases the smaller the error is near the mean of $\mu = 0.0$. This resembles a summation of bell curves. The Gaussian reward does not provide any negative rewards. This reward function was inspired by Martinsen in [30] which performed path following of marine vessels. Given that it is desired to get the trailer to follow the position and orientation, a combination of weighted Gaussian reward functions are used. A standard deviation of $\sigma = 3$ was found to provide a desirable shape.

$$r = 0.5e^{-\frac{(y_{2e}-\mu)^2}{2\sigma^2}} + 0.5e^{-\frac{(\psi_{2e}-\mu)^2}{2\sigma^2}} \tag{6.20}$$

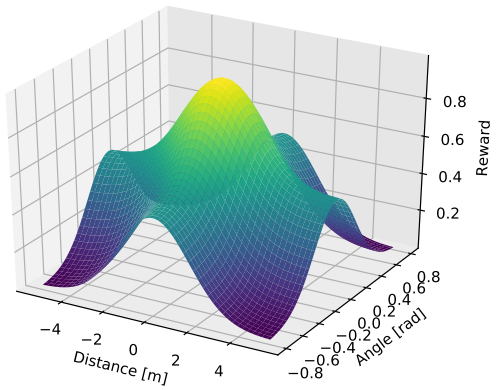


Figure 6.12: The Gaussian reward function provides continuous, increasing reward for zero error with respect to the path.

Reward Scheme C

Reward Scheme C takes into account the cosine of the trailer angle and a sigmoid function with only absolute values. This promotes angles to the path because cosine of zero is one. The sigmoid function with only absolute values of the distance to the path creates a valley within a bounded region. The nature of this reward function gives positive and negative rewards. Jaritz in [82] utilized this reward scheme C, but with velocity multiplied against it to promote increasing speed.

$$r = \cos\psi_{2e} - \frac{1}{(1 + e^{-4(|y_{2e}|-0.5\,y_{max})})} \tag{6.21}$$

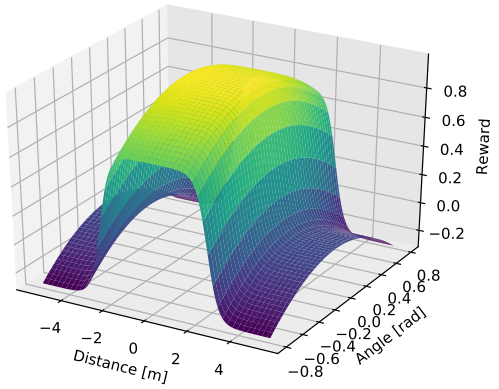


Figure 6.13: Jaritz utilized this reward scheme C, but with velocity multiplied against it to promote increasing speed.

Reward Scheme D

This reward scheme investigates using the LQR cost function, but instead as a subtraction in order to maximize it. Q here is not be confused with the Q-value from the Critic. The trailer is not rewarded for anything in regard to the tractor. Something to note is that this reward function penalizes for high actions. It is difficult to generate a meaningful visual for this reward scheme due to having to sweep through various actions as well.

$$r = -\int_0^{t_f} (\underline{s}^T Q \underline{s} + \underline{a}^T R \underline{a}) dt \tag{6.22}$$

$$Q = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} R = \begin{bmatrix} 1 \end{bmatrix} \tag{6.23}$$

Reward Scheme E

Reward scheme E is a combination of reward scheme B and D. This takes advantage of shaped positive rewards, but also benefits from a cost for a high action and errors.

$$r = 0.5e^{-\frac{(d_{2e}-\mu)^2}{2\sigma^2}} + 0.5e^{-\frac{(\psi_{2e}-\mu)^2}{2\sigma^2}} - \int_0^{t_f} (\underline{s}^T Q \underline{s} + \underline{a}^T R \underline{a}) dt \tag{6.24}$$

Reward Scheme F

Reward scheme F is a combination of reward functions A and D.

$$r = 1.0 - 0.5(\frac{|y_{2e}|}{y_{max}})^{0.4} - 0.5(\frac{|\psi_{2e}|}{\psi_{max}})^{0.4} - \int_0^{t_f} (\underline{s}^T Q \underline{s} + \underline{a}^T R \underline{a}) dt \tag{6.25}$$

A plethora of reward function combinations were investigated, but the aforementioned ones provide meaningful insight for what reward functions may need to look like from the design perspective.

6.1.11 Machine Teaching

The primary design decisions have now been discussed for the original DDPG algorithm for the tractor-trailer system. The vanilla implementation struggled to learn on a very simple track, so the following additional methods were discovered to assist in training the agent to back the trailer up to the loading dock. Machine Teaching consists of techniques to get information from humans as much as possible to the algorithm. They are not necessarily a part of the algorithm, but are used in this work to augment the DDPG agent's learning.

Guided Training

DDPG is an off-policy algorithm, which means that it is possible to train the algorithm on policy transitions not generated from the learned policy. Usually with the DDPG algorithm, exploration occurs with the Ornstein-Uhlenbeck noise. This was found to unfortunately take substantially longer to converge on the simple course of a straight line.

An expert policy could also be used to teach the agent. It is possible to use the fixed gains from the LQR to provide good examples. This, however, has a caveat. When one trains on only expert policies, there becomes a mismatch in the population distribution between the actor and the expert policy. This is because the expert policy only visits a portion of the state space. What this means is that during testing time, the actor may not perform as well because it has not generalized from the entire state space.

Martinsen [30] uses guided training for path tracking of marine vehicles using DDPG. He mentions that in order to counteract the potential bias from the expert policy, one should use a blended policy where ω is the blending factor. In order to reduce the bias, it is suggested to sample from the blended policy with probability p and use the learned policy with probability $1 - p$.

$$\pi_{blended} = (1 - \omega)\pi_\theta + \omega \pi_{guide} \tag{6.26}$$

Machine learning assumes the data is independent and identically distributed. This means the training data has the same probability distribution and are mutually independent of each other. As long as this holds true, training will continue to improve. A good example of this is a coin toss for heads or tails. The coin has no memory, so all throws are "independent." The probability is 50% and will stay this way, so it is "identically distributed." With this logic, a blended policy is not necessarily needed.

A similar method to introduce an expert policy was discovered through the work of this thesis. The idea is to switch with a probability of 50% between the original DDPG exploration policy centered around the prediction of the neural network and the LQR. At every timestep of an episode, the action is randomly toggles between the original DDPG algorithm and what the LQR would do. The LQR actions would carry the performance closer to the goal to ensure the replay buffer gets populated with good samples as well.

$$a = \begin{cases} Ks & \text{with} \quad p = 0.5 \\ \mu(s) + N & \text{with} \quad p = 0.5 \end{cases} \tag{6.27}$$

Considering this thesis compares the LQR to DDPG for performance, it may appear that using the LQR as an expert policy may bias results. Furthermore, any flaws that the LQR has may be taught to the agent. Reinforcement learning theory holds that the DDPG agent should only improve upon the expert policy. Additionally, the neural network is an approximation conceived from samples it learns from. This means there is potential for the DDPG agent to generalize when the trailer parameters change.

Transfer Learning

In the context of supervised learning, initializing the neural network with one set of weights can propagate training to take longer to achieve the desired performance. Starting training with a different set of initial weights could speed up the training time. This is part of the stochastic nature of training. The other part is randomly selecting batches from the training data; this gets rid of the temporal correlation between samples.

Transfer learning is a method used to extrapolate weights from one solved problem to a new, similar problem. The weights are loaded up to a new neural network and training begins on a new problem. Instead of training from scratch, transfer learning should speed up training because it starts from a stronger base. Humans learn by solving a simple problem first, then tackle harder problems.

For example, children start playing minor league baseball before entering the major league. In the context of reinforcement learning, transfer learning can be used in lesson plans or a curriculum.

The tractor-trailer, for example, may begin with training on straight paths. The next lesson plan may be to learn a left turn. The agent could then continue to learn a right turn, then a combination of straights and turns, and so forth. Instead of immediately learning on a multitude of complicated paths, transfer learning was found necessary and useful.

Convergence Criteria

Convergence criteria is useful to ensure the agent stops training when the average of performance meets the desired outcome. It saves computational time and, in fact, can result in better performance than just stopping after a set amount of episodes. In the machine learning jargon, convergence is also considered to be early stopping.

Instead of designing a convergence criteria, one could save the weights as a checkpoint after every episode. However, for $10,000$ episodes, the storage capacity required grows tremendously. Having a convergence criteria really lets the algorithm run on its own such that the developer does not have to manually observe through the night to make sure it stops training at the correct time.

The convergence criteria for the toy example was already suggested by the environment: convergence occurs when the average reward for the past 100 episodes is greater than 90. This becomes much more challenging when trying to determine the ideal reward function for a new problem. Different reward functions will provide vastly different upper bounds on the rewards. One can use a known control method like the LQR to determine what a "good" average reward to target is, however, this cannot be done easily when the track lengths and shapes continue to change.

Another natural idea may be to evaluate an average of the LQR performance metrics such as root mean squared error of all the states. This, however, is not in the spirit of optimizing the control performance with respect to itself. One could also early stop when the agent makes it to the goal consistently in the past 100 episodes, however, it is difficult to determine the exact number out of 100 provides a good measure of training. In addition to this, this does not necessarily ensure the agent follows the path by minimizing the error. The determination of the convergence criteria will be discussed in the evaluation section of this chapter.

6.2 Reinforcement Learning Implementation

6.2.1 Minimal Example with TruckBackerUpper-v0

A custom OpenAI gym environment was written in Python using the modules *numpy*, *scipy*, *gym*, *matplotlib*, and *dubins*. The environment consists of two classes, one for the TruckBackerUpper-v0 environment and the other that contains Path Planning helper functions.

The Path Planning class is a wrapper around *dubins* to modify the Dubins paths for an additional straight at the end. Since Dubins Curves do not have a way to bound the paths within an area, the Dubins paths are randomly generated until a path is found within the $80 \times 80m^2$ area.

The implementation of the environment has already been discussed in Chapter 3. The basic methods include *seed*(), *reset*(), *step*(), and *close*(). A minimal example of using the LQR controller in the Python environment is useful in this chapter to describe the implementation of the DDPG algorithm.

```
import gym
import gym_truck_backerupper
import numpy as np

env = gym.make('TruckBackerUpper-v0')

done = False
s = env.reset()

while not done:
    env.render()

    # Example action from LQR, replace with RL policy
    K = np.array([-3.8249, 12.1005, -1.000])
    a = np.clip(K.dot(s), env.action_space.low, env.action_space.high)

    s_, r, done, info = env.step(a)
    s = s_
env.close()
```

In order to interact with the environment, the script containing the controller will need to import *gym*, the environment and *numpy*. An object, usually called 'env,' will need to be created from the gym module for the specific environment. Before the episode begins, a variable called 'done' should be set to False. The initial states should be determined through resetting the environment. Resetting will spawn a random track, defining the starting positions. While there are no terminal conditions met, the agent will loop through the following items. The environment should render to show incremental progress of where the trailer is moving like in Figure 6.14.

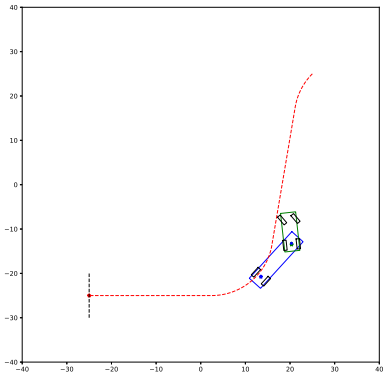


Figure 6.14: The rendering displays the sequential movement of the tractor-trailer system in its objective of following the dashed red path to the loading dock. The path can be recreated using Dubins Curves with $[25.0, 25.0, 225.0], [-25.0, -25.0, 180.0]$ **as the starting and ending configuration** $q(x, y, \psi)$

Line 14 of the minimal example instantiates the gains for the LQR. Line 15 multiplies the gains against the vector of states to produce the desired steering action, but also clips the action to realistic values. These lines can be replaced with the reinforcement learning policy, or simply the prediction for the actor neural network.

Once the desired action is determined, a step can be taken in the environment. The output of this method returns the next state, $s\_$, the reward, r, terminal conditions, done, and a dictionary of information for why the episode may have ended. Once the step has completed, the current state is set to the next state so the process can loop. Once completed, the close method exits from the environment rendering.

Any reasonable controller can be used to interact with the TruckBackerUpper-v0 environment because each action results in a new state. The control method is left up to the user for creation. If more episodes are desired, a for loop should be written where each episode begins by declaring $done = False$ and resetting the environment.

6.2.2 Unique Implementations Regarding Tensorflow

Tensorflow Starting Seeds

When making hyperparameter or reward function changes, it is useful to set the initial seed in order to get the same sequence. Just running the same code twice can produce wildly different results in terms of convergence. Computers use a pseudo random number generation method which usually uses the current timestamp as the initial seed. Sources of random seeds are introduced by

the track spawning, noise for exploration, batch randomization, and most importantly initial weights and biases. The reason why the method for training neural networks is called Stochastic Gradient Descent is due to the randomization of mini-batches and initial weights. To make the sequence deterministic, one must set the initial seed for *numpy* and *random* modules of Python. In addition to these, one must set the initial seed for $tensorflow$.

It was discovered that $tensorflow$ continues to ensure stochastic results despite offering a method for setting a global seed, $tf.set\_random\_seed()$. This is useful if the weights are already set and one wants to reproduce results. The problem is when using lower level $tensorflow$ operations; some require the initial seed to be set for each operation. This occurs especially with weight initialization. When one does not set these seeds, it uses the operation id number as its starting seed. Unfortunately, this does not seem to be constant. Furthermore, even when setting these local operation initial seeds, it appeared to not be deterministic still.

Machine learning is still simply using math, but this causes many to be perplexed as to why the results are still not deterministic for training. One can begin to question whether or not if machine learning really is a sentient Artificial Intelligence. A solution, however, was found using $tf.contrib.stateless\_random\_uniform()$. It allowed for weight and bias initialization without dependency on the operation id numbers. The drawback is that one has to manually implement all the layers instead of implementing $tf.dense()$ layers.

```
        with tf.variable_scope("hidden1"):
            w1 = tf.Variable(self.fan_init(self.n_states) *
                             (2.0 * tf.contrib.stateless.stateless_random_uniform(
                                        [self.n_states, self.n_neurons1],
                                        seed=[self.seed, 0]) - 1.0),
                                        trainable=trainable)
            b1 = tf.Variable(self.fan_init(self.n_states) *
                             (2.0 * tf.contrib.stateless.stateless_random_uniform(
                                        [self.n_neurons1],
                                        seed=[self.seed+1, 0]) - 1.0),
                                        trainable=trainable)
            hidden1 = tf.matmul(s, w1) + b1
            hidden1 = tf.nn.relu(hidden1)

    def fan_init(self, n):
        return 1.0 / np.sqrt(n)
```

Manually implementing a layer in $tensorflow$ consists of initializing the weights and biases as tensorflow variables using the $tf.contrib.stateless\_random\_uniform()$ method. The *contrib* method returns a random float between the range of [0, 1) based on a specific starting seed. Since the

DDPG algorithm suggests to initialize the weights and biases using the fan in method, the trainable parameters are scaled and shifted to achieve this. The weights get multiplied by the inputs or the previous layer outputs, then shifted by the biases. The output of this then gets put through the activation function.

Batch Normalization Details

The DDPG algorithm suggests that using batch normalization help with varying units of states. It was found to be helpful, but something about the implementation was unintuitive. When it comes to evaluating and exploring, the target networks should use the population mean and variance. Reinforcement Learning is different than supervised learning because the agent continually interacts with the environment by making predictions on the input and output mappings. One could think of this as assuming the network is good for the time being and seeing how well the predicted action is.

The code snippet below displays how batch normalization is implemented on a layer if bn = True. The $tf.contrib.layers.batch\_norm()$ method has an argument for setting $is\_training$ True or False.

```
        hidden1 = tf.matmul(s, w1) + b1
            if bn:
                hidden1 = self.batch_norm_layer(hidden1,
                                                train_phase=self.train_phase_actor,
                                                    scope_bn=n_scope+'1')
                hidden1 = tf.nn.relu(hidden1)

    def batch_norm_layer(self, x, train_phase, scope_bn):
        return tf.contrib.layers.batch_norm(x, scale=True,
                                                updates_collections=None,
                                                decay=0.999, epsilon=0.001,
                                                scope=scope_bn)
```

It was found that setting is_training = False for the target networks produced worse results in terms of convergence on the ContinuousMountainCar-v0 environment. It was initially thought it was due to the target network population mean and variance layer values were not getting updated correctly due to polyak averaging. If one tried to manually copy over the population mean and variance to the target network from the main network, it actually performed worse. So despite the theoretical grounding, it was found that setting $is\_training = True$ during the exploration was needed. This is further discussed in appendix B.

Saving Weights and Biases

The implementation of the DDPG algorithm uses *tensorflow* 1.12, which instantiates a computational graph. Instead of looping through Python code, *tensorflow* refers to the computational graph to take advantage of the GPU. The computational graph gets created in a tensorflow session. A visual of the computational graph can be found using *tensorboard* in Figure 6.15, which is a resource to view progress of training on a web browser.

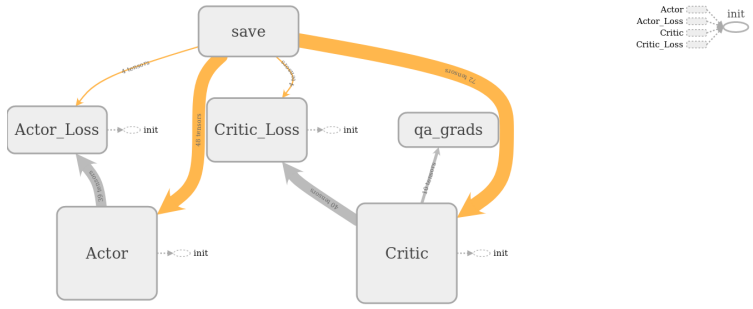


Figure 6.15: The computational graph displays joint computational graph consisting of the actor and critic.

```
with tf.Session() as sess:
    ''' Construct neural networks here '''

    sess.run(tf.global_variables_initializer())
    saver = tf.train.Saver()

    ''' Train neural networks until convergence '''

    saver.save(sess, checkpoint_path)
```

Inside the tensorflow session, the various weights and biases of all the neural networks should be constructed. This is where incorporating the starting seed matters. The computational graph is not created until the session runs the global variables initializer. An object is created so the save method can be used. Training of the neural networks should then begin and end once convergence has been met. Then the save method takes the session and saves the model, consisting of weights and biases, at a desired file location.

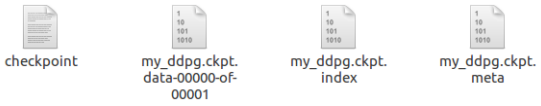


Figure 6.16: The saved model consists of four files: one that consists of text and the other three are binary.

The saved models from *tensorflow* are usually referred to as checkpoint files because the user can restore checkpoints if the training gets interrupted. Due to the files primarily being binary, the ability to open the file to view the weight values is not available. The model, however, can be restored within another tensorflow session for viewing.

6.2.3 Restoring Model

Restoring a model was used in this thesis to implement transfer learning. More importantly, it is also used for deploying the neural network of the actor policy for testing. Inside a new tensorflow session, the computational graph with the saved weights and biases can be restored using *tf.train.import_meta_graph*() and the restore method.

```
with tf.Session() as sess:
    saved = tf.train.import_meta_graph(checkpoint_path + '.meta',
                                       clear_devices=True)
    saved.restore(sess, checkpoint_path)

    # The specific terms below are needed when testing
    state = sess.graph.get_tensor_by_name('Actor/s:0')
    train_phase = sess.graph.get_tensor_by_name('Actor/train_phase_actor:0')
    learned_policy = sess.graph.get_tensor_by_name(
                                    'Actor/pi_online_network/pi_hat/Mul_4:0')
```

If transfer learning is desired, then the computational graph simply needs to be restored in the DDPG class and training can continue. If testing is desired, it is not necessary to run the critic network any longer in the original script. The learned policy can be used directly in a new script, but some knowledge of the specific names of variables on the computational graph are needed. The output of the actor policy network is the learned policy. In order to predict with it, the states that get fed into it are needed again. Since the computational graph is written in a general way such that one could use batch normalization, the train phase named argument is also needed.

6.2.4 DDPG Class and Method Diagram

The DDPG algorithm was implemented in Python using packages such as tensorflow, numpy, and gym. Python is an object oriented programming language. The implementation consists of five different classes: one for each the Actor, Critic, Ornstein-Uhlenbech Noise, Replay Buffer and finally DDPG. The DDPG class uses the other classes to train and test the agent. The Unified Modeling Language (UML) class diagram for the DDPG algorithm is shown in Figure 6.17.

The UML class diagrams connect different classes with various relationships. Three compartments are displayed a single class rectangle. The top section is for the class name, the middle is for attributes, and the last is for the operations. The plus symbol in the attributes compartment represent the idea that the attribute is for public visibility.

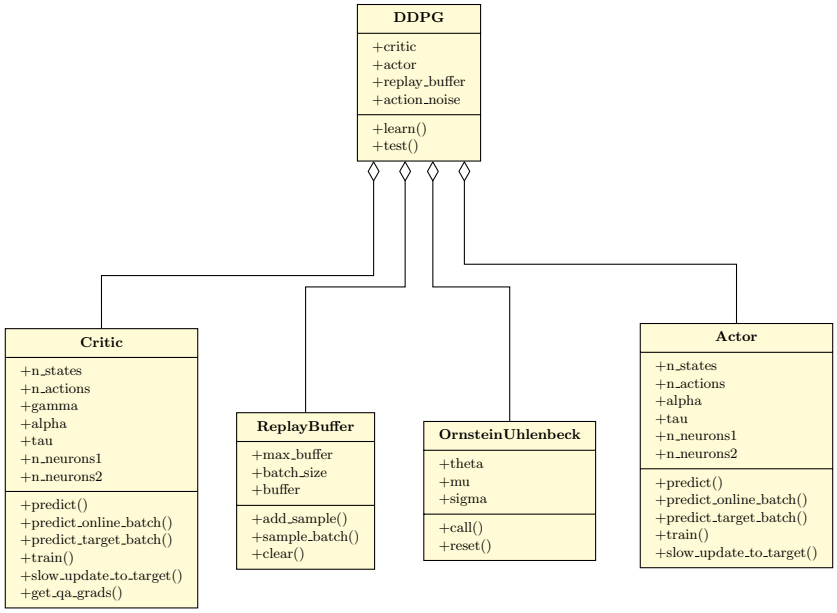


Figure 6.17: The class diagram for the DDPG algorithm implementation in Python explains how the DDPG class has a relationship called aggregation with the other classes, which is denoted with the white diamond.

The DDPG class has a relationship called aggregation with the other classes, which is denoted with the white diamond. Aggregation occurs because the DDPG class stores the reference to the other classes for later use. The construction of the DDPG class stores objects for the critic, actor, replay buffer, and Ornstein-Uhlenback classes as attributes. Aggregation protects the integrity for the assembly of objects by defining a single point of control. Inside the DDPG class, the methods for *learn*() and *test*() use information and operations from the stored classes it interacts with.

6.2.5 Replay Buffer Class

The replay buffer class has two main methods: add_sample() and sample_batch(). The class uses the collections module to house the buffer of transitions which include state, action, reward, next state and terminal conditions. The main script defines the replay buffer size of 1 million samples and the batch size of 64.

```
    def add_sample(self, sample):
        if self.count < self.max_buffer:
            self.buffer.append(sample)
            self.count += 1
        else:
            self.buffer.popleft()
            self.buffer.append(sample)

    def sample_batch(self, batch_size):
        batch = []

        if self.count < batch_size:
            batch = random.sample(self.buffer, self.count)
        else:
            batch = random.sample(self.buffer, batch_size)

        s_batch = np.array([_[0] for _ in batch])
        a_batch = np.array([_[1] for _ in batch])
        r_batch = np.array([_[2] for _ in batch])
        s__batch = np.array([_[3] for _ in batch])
        d_batch = np.array([_[4] for _ in batch])

        return s_batch, a_batch, r_batch, s__batch, d_batch
```

The DDPG class will use the *add_sample*() method after ever step in the environment. The buffer will continue to grow until the maximum buffer size has been reached. Once this occurs, then the oldest samples will be popped off the left. When training occurs, the same minibatch will be used for both the critic and actor. The *sample_batch*() method randomly selects transitions from the buffer and returns them as separate batches for each of the terms.

6.2.6 Exploration Noise Class

The Ornstein_Uhlenbeck class has the sole purpose of returning the exploration noise. If the parameters are not specified in the construction, the default is $\sigma = 0.3$, $\theta = 0.6$, $\mu = 0.0$, and $dt = .02$. These default parameters are different than what Lillicrap's paper suggested worked for 20 continuous toy examples. Increasing θ meant that the steering wheel returned to center much

more often. The variance, σ, was also increased so more samples of larger steering was introduced to the replay buffer.

```
import numpy as np

class Ornstein_Uhlenbeck:
    def __init__(self, mu, sigma=0.3, theta=0.6, dt=1e-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)
```

What may appear as odd is the fact that the timestep of the noise does not match with the tractor-trailer environment timestep of 80ms. This is intentional because increasing the Ornstein-Uhlenbeck timestep unfortunately made the exploration too noisy for learning on. These parameters were self evaluated and tuned to some extent, however, future work can be done to investigate a better exploration noise for the tractor-trailer system. Instead, time was spent on bettering the guided training with an expert policy.

The DDPG class will reset the noise before every episode. During every timestep, the DDPG class will call this class for the next noise to add to the action. The noise may or may not be used every step due to the expert policy being selected the other 50% of the time.

6.2.7 Critic Class

The critic class primarily houses the critic neural network and provides the training method. Constructing the critic begins with initializing the number of states, the number of actions, the discount factor, the learning rate, the number neurons, etc. The construction of the computational graph to include the critic is also performed in the initialization of the class.

Inside a variable scope named 'Critic,' contains the various variables and tensorflow operations such that it can be easily found on the computational graph. The actor will also exist on the same computational graph, so it makes sense to organize variables as such. First, several terms are set

up as tensorflow placeholder variables, which are the pipelines of feeding data to the computational graph. These variables are not known ahead of time and will constantly be changing. The placeholder variables for the critic include the state, next state, action, predicted action, target label and the training phase. Table 6.3 summarizes the variables mentioned above.

Table 6.3: This table summarizes the placeholder values for the critic implementation.

Placeholder variable	Description
s	state
s_	next state
a	action
a_	predicted action from target actor
y	target label for critic
train_phase_critic	boolean used for toggling batch normalization mean and variance

Nested inside the 'Critic' variable scope are a few more variable scopes. The 'Q_online_network' variable scope is where the main neural network is built using the state and action as inputs. The critic network needs to be built twice since there is an online and a target network, thus it is a class method called *build_network*(). Similarly, the variable scope for the target network is 'Q_target_network.'

The *build_network*() function has arguments for the states, actions, booleans for making the parameters trainable, boolean if regularization is needed, and a string to decide whether or not the target or online network is built. Determining which neural network is built is necessary only in the critic due to writing the class such that it can generally choose L_2 weight decay or not.

First the weights are initialized as a tensorflow variable as mentioned before. If the online network is desired, an if-else statement checks if the L_2 weight decay has been set. If so, the $tf.contrib.layers.l2\_regularizer()$ is set as the regularizer. If not, then the regularizer is set to None. A variable scope titled 'reg' is created to contain the weights of the online network with the regularizer. The tensorflow function called get_variable() is used to create new variables with the regularizer using the initialized weights. This specific syntax was found to work in order to obtain proper seed initialization and regularization. If the target network is being built, then no regularization needs to be used.

The layers of the neural networks are then built with the proper number of neurons. What is unique to the critic is the need to augment the action to the second hidden layer. The following code snippet shows the details of the augmentation.

```
        with tf.variable_scope("hidden2"):
            augment = tf.matmul(hidden1, w2) + tf.matmul(a, w3) + b2
            hidden2 = tf.nn.relu(augment)
```

Next, the trainable parameter variables of the online and target networks are collected so the target could be updated slowly from the online network in a tensorflow operation. Finding the trainable parameter variables was the simplest method for determining the neural network parameters from the computational graph even though the target network is not trainable. In order to use the tensorflow operation, the session needs to run the name of the tensorflow operation on the computational graph. Thus, a class method called *slow_update_to_target*() is used.

```
        self.vars_Q_online = tf.get_collection(
                                    tf.GraphKeys.TRAINABLE_VARIABLES,
                                    scope='Critic/Q_online_network')
        self.vars_Q_target = tf.get_collection(
                                    tf.GraphKeys.TRAINABLE_VARIABLES,
                                    scope='Critic/Q_target_network')

        with tf.name_scope("slow_update"):
            slow_update_ops = []
            for i, var in enumerate(self.vars_Q_target):
                slow_update_ops.append(var.assign(
                        tf.multiply(self.vars_Q_online[i], self.tau) + \
                        tf.multiply(self.vars_Q_target[i], 1.0-self.tau)))
            self.slow_update_2_target = tf.group(*slow_update_ops,
                                                 name="slow_update_2_target")

    def slow_update_to_target(self):
        self.sess.run(self.slow_update_2_target)
```

The critic loss or temporal difference error is determined between the prediction of the online Q network and the target label which uses the discount factor for future reward. If regularization is used, then the regularization term gets added to the loss. The tensorflow API for backpropagation is set up as the training operation using the Adam optimizer. This is where the learning rate gets included. In order to use the optimizer operation from the computational graph, a function named train() is used to run this from the session.

```
        with tf.name_scope("Critic_Loss"):
            td_error = tf.square(self.y - self.Q_online)
            self.loss = tf.reduce_mean(td_error)
            if self.l2:
                reg_term = tf.losses.get_regularization_loss()
                self.loss += reg_term

```

```
            optimizer = tf.train.AdamOptimizer(self.alpha)
            self.training_op = optimizer.minimize(self.loss)

    def train(self, s_batch, a_batch, y_batch, train_phase=None):
        self.sess.run(self.training_op, feed_dict={
                        self.s: s_batch, self.a: a_batch, self.y: y_batch,
                        self.train_phase_critic: train_phase})
```

It is probably worth mentioning that the tensorflow operation will automatically run other operations in order to complete the operation. Since the training operation requires the loss operation to be calculated, it will go ahead and run a prediction of the Q_online network. Care simply needs to be taken that all of the necessary placeholders are fed into the session run.

The last operation to set up for the computational graph is the ability to calculate the gradients of the Q value with respect to the actions from the actor. Just as before, in order to use this operation a class method was used to run this in the session.

```
        with tf.name_scope("qa_grads"):
            self.qa_gradients = tf.gradients(self.Q_online, self.a)

    def get_qa_grads(self, s, a, train_phase=None):
        return self.sess.run(self.qa_gradients, feed_dict={self.s: s, self.a: a,
                             self.train_phase_critic: train_phase})
```

The critic's computational graph has now been created. The variable scopes that were discussed can be visualized in the tensorboard visualization. The scopes have been expanded to display what has been created in Figure 6.18.

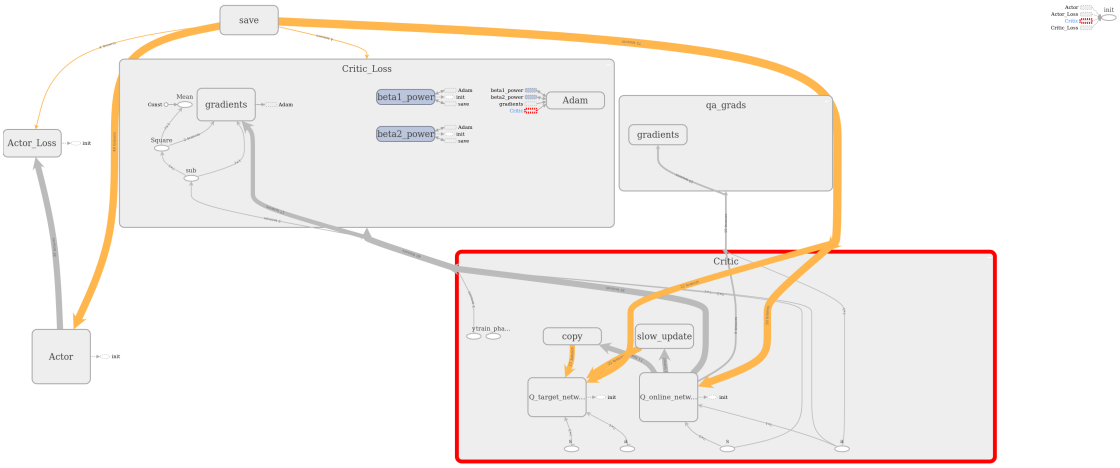


Figure 6.18: The visualization of the critic's portion of the computational graph shows how tensors are passed around for tensorflow operations.

The DDPG class calculates the target labels, y, for updating the critic network because it also depends on the target network of the actor. In order for the DDPG class to calculate the target labels, it uses the critic class to predict the Q values for a batch. Both the online and target critic networks are needed in this process, so class methods were created for them to run the neural networks in the session.

```
    def predict_online_batch(self, s, a, train_phase=None):
        return self.sess.run(self.Q_online, feed_dict={self.s: s, self.a: a,
                             self.train_phase_critic: train_phase})

    def predict_target_batch(self, s_, a_, train_phase=None):
        return self.sess.run(self.Q_target, feed_dict={self.s_: s_, self.a_: a_,
                             self.train_phase_critic: train_phase})
```

The prediction operations run from the computational graph, so it grabs all the necessary operations preceding the output of the neural network to compute a forward pass. The online prediction uses the state and action, whereas the target prediction uses the next state and the action prediction from the target actor. The full implementation of the critic class can be found in appendix C.

6.2.8 Actor Class

The actor class has the main purpose of maintaining the actor neural network, providing the training method. The output of the actor network is the action; this is the reinforcement learning policy or controller. The actor is constructed by initializing the number of states, number of actions, the learning rate, number of neurons, and the upper bounds of the action. The computational graph continues its construction with the actor tensors and operations within the initialization of this class.

Just like the critic, the placeholder variables are created inside a variable scope named 'Actor.' The data that is fed into the computational graph for the actor include the state, next state, the gradient of the Q values with respect to the actions, the train phase, and the batch size. Table 6.4 summarizes the variables.

Table 6.4: This table summarizes the placeholder values for the actor implementation.

Placeholder variable	Description
s	state
s_	next state
qa_grads	the gradient of the q values with respect to the actions
train_phase_actor	boolean used for toggling batch normalization and mean variance
batch_size	minibatch size from replay buffer used for normalizing gradients

Nested inside the 'Actor' variable scope include additional operations located inside additional variable scopes. The 'pi_online_network' variable scope contains the creation of the online actor neural network using build_network(). Once created on the computational graph, the DDPG class can use the predict() method to run a session with the tensorflow operation to make a single prediction of the action.

```
        with tf.variable_scope('pi_online_network'):
                self.pi_online = self.build_network(self.s,
                                                    trainable=True,
                                                    bn=self.bn,
                                                    n_scope='batch_norm')
    def predict(self, s, train_phase=None):
        return self.sess.run(self.pi_online,
                             feed_dict={self.s: s.reshape(1, s.shape[0]),
                                        self.train_phase_actor: train_phase})
```

The *build_network*() method has arguments for the states and batch normalization if needed. If batch normalization is set, the state inputs are first batch normalized. This is very standard for supervised learning neural networks. The first hidden layer of this implementation is built manually, so the initial seed is properly set for the weight and bias initialization. If batch normalization is necessary, the layer is batch normalized before the RELU activation function.

Similarly, the second hidden layer weights are initialized so the computation of the weights times the inputs plus the bias is performed. The inputs to the second hidden layer is the output of the previous layer. The layer checks if batch normalization is requested and then applies the activation function.

What is unique to the actor is the output layer because of the tanh activation function. First the weights and biases are initialized to the small range of $[-0.003, 0.003]$ to ensure small initial outputs. The layer is constructed with the tanh activation for the output, but it is important to

scale the result so the action is resembles the steering angles in radians. Since the output of the tanh activation function is a maximum of $[-1, 1]$, the result is scaled by 0.785 radian.

```
        with tf.variable_scope("pi_hat"):
            w3 = tf.Variable(0.003 * (2.0 *
                             tf.contrib.stateless.stateless_random_uniform(
                                       [self.n_neurons2, self.n_actions],
                                       seed=[self.seed+4, 0]) - 1.0),
                                       trainable=trainable)
            b3 = tf.Variable(0.003 * (2.0 *
                             tf.contrib.stateless.stateless_random_uniform(
                                       [self.n_actions],
                                       seed=[self.seed+5, 0]) - 1.0),
                                       trainable=trainable)
            pi_hat = tf.matmul(hidden2, w3) + b3
            pi_hat = tf.nn.tanh(pi_hat)
            pi_hat_scaled = tf.multiply(pi_hat, self.action_bound)
```

The *build_network*() method is used identically for the actor target network, but is under the variable scope of 'pi_target_network.' Now that both the online and target networks have been created on the computational graph, operations to collect the trainable parameters are used. An operation to slowly update the target network weights and biases from the online network is ultimately created on the computational graph. This operation can be requested by the DDPG class by using the *slow_update_to_target*() method of the actor class.

```
            self.vars_pi_online = tf.get_collection(
                                       tf.GraphKeys.TRAINABLE_VARIABLES,
                                       scope='Actor/pi_online_network')
            self.vars_pi_target = tf.get_collection(
                                       tf.GraphKeys.TRAINABLE_VARIABLES,
                                       scope='Actor/pi_target_network')

            with tf.name_scope("slow_update"):
                slow_update_ops = []
                for i, var in enumerate(self.vars_pi_target):
                    slow_update_ops.append(var.assign(
                            tf.multiply(self.vars_pi_online[i], self.tau) + \
                            tf.multiply(self.vars_pi_target[i], 1.0-self.tau)))
                self.slow_update_2_target = tf.group(*slow_update_ops,
                                                     name="slow_update_2_target")
    def slow_update_to_target(self):
        self.sess.run(self.slow_update_2_target)
```

The actor network can be updated using the training operation defined in the 'Actor_Loss' variable scope. There isn't a real loss associated with updating the actor because it is a Deterministic

Policy Gradient method. Instead the weights and biases are updated in the direction of the policy gradient, performing gradient ascent.

The actor gradients are calculated using the tensorflow gradient method using the forward prediction of the online network, its variables, and the negative of the pre-calculated gradients of the Q values with respect to the actions. The term is negative to perform a maximization for gradient ascent. The pre-calculated gradients of the Q values with respect to the actions are determined from the critic class method, *get_qa_grads*(). The DDPG class takes that information and feeds it into the placeholder variables for *qa_grads*.

The actor gradients are then normalized for the batch size using a lambda function, which is a local function. This information is fed to the Adam optimizer using the learning rate. The training operation then applies these gradients to the variables of the online network. The training operation can be requested using the *train*() method of the actor class.

```
        with tf.name_scope("Actor_Loss"):
            raw_actor_grads = tf.gradients(self.pi_online,
                                           self.vars_pi_online,
                                           -self.qa_grads)
            self.actor_grads = list(map(lambda x: tf.div(
                                    x, self.batch_size),
                                    raw_actor_grads))

            optimizer = tf.train.AdamOptimizer(self.alpha)
            self.training_op = optimizer.apply_gradients(
                                                zip(self.actor_grads,
                                                self.vars_pi_online))

    def train(self, s_batch, qa_grads, batch_size, train_phase=None):
        self.sess.run(self.training_op, feed_dict={
                              self.s: s_batch, self.qa_grads: qa_grads,
                              self.batch_size: batch_size,
                              self.train_phase_actor: train_phase})
```

Now that the construction of the actor network is complete for the computational graph, a visual of the tensors and operations is shown in Figure 6.19. The scopes have been expanded only for the actor.

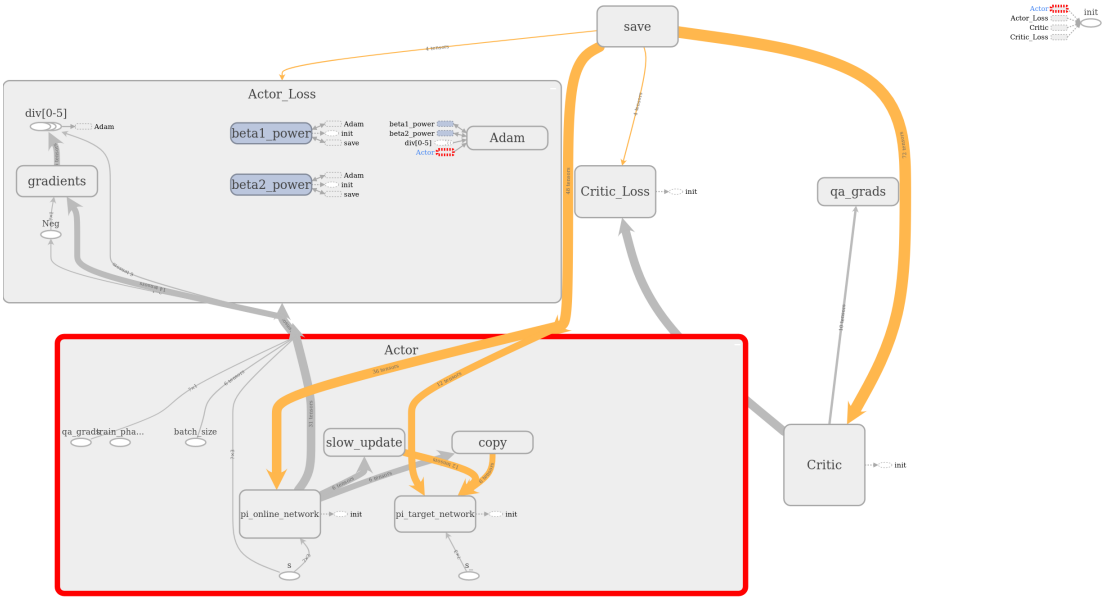


Figure 6.19: The visualization of the actor's portion of the computational graph shows how tensors are passed around for tensorflow operations.

Finally, additional methods for the actor class were needed to run predictions on batches so the DDPG class can update both neural networks. The target actor network is needed to determine the target labels for the critic, which promotes stability in training the critic. The DDPG algorithm requires the prediction from the target actor network to use inputs of the next state.

```
    def predict_online_batch(self, s, train_phase=None):
        return self.sess.run(self.pi_online, feed_dict={self.s: s,
                                        self.train_phase_actor: train_phase})

    def predict_target_batch(self, s_, train_phase=None):
        return self.sess.run(self.pi_target, feed_dict={self.s_: s_,
                                        self.train_phase_actor: train_phase})
```

In addition, a batch prediction of the online network is needed to create the placeholder variables to feed into the training operation of the actor. The inputs to the online network are the current states. The prediction operations, once again, grab all the necessary operations from the computational graph in order to compute a forward pass. This means that once the actor network has been saved, the user can use the output layer of the network on the computational graph to directly compute the action for control. The full implementation of the actor class is found in appendix C.

6.2.9 DDPG Class

The DDPG class is where the reinforcement learning algorithm primarily resides, using the actor and critic classes for predicting and training. It uses the replay buffer class to store and sample batches. The DDPG class uses the Ornstein_Uhlenbeck class to add some noise to the actor predictions of the action.

Outside of the DDPG class is the main loop, where objects are created for each of the classes. The objects for the other classes are passed into the DDPG class so two primary methods can be used: learn() and test(). The code snippet below displays a skeleton of how the DDPG class is used in the main loop to interact with the other classes.

```
from Critic import Critic
from Actor import Actor
from ReplayBuffer import ReplayBuffer
from Ornstein_Uhlenbeck import Ornstein_Uhlenbeck

class DDPG:
    def __init__(self, sess, critic, actor, action_noise, replay_buffer):
        ''' Initialize DDPG class '''

    def learn(self, env, EPISODES, TRAIN_STEPS, COPY_STEPS, BATCH_SIZE,
              WARM_UP):
        ''' The DDPG agent interacts with the environment to train '''

    def test(self, env, policy, state, train_phase, sess):
        ''' The DDPG agent uses the learned policy '''

if __name__ == '__main__':
    for seed_idx in range(len(SEEDS)):
        with tf.Session() as sess:
            critic = Critic(sess, env.observation_space.shape[0],
                            env.action_space.shape[0], GAMMA, ALPHA_C, TAU,
                            N_NEURONS1, N_NEURONS2, BN, L2, SEEDS[seed_idx])
            actor = Actor(sess, env.observation_space.shape[0],
                          env.action_space.shape[0], ALPHA_A, TAU,
                          env.action_space.high, N_NEURONS1, N_NEURONS2, BN,
                          SEEDS[seed_idx])

            sess.run(tf.global_variables_initializer())
            action_noise = Ornstein_Uhlenbeck(mu=np.zeros(
                                              env.action_space.shape[0]))
            replay_buffer = ReplayBuffer(MAX_BUFFER, BATCH_SIZE)
            replay_buffer.clear()
            agent = DDPG(sess, critic, actor, action_noise, replay_buffer)
            saver = tf.train.Saver()

            agent.learn(env, EPISODES, TRAIN_STEPS, COPY_STEPS, BATCH_SIZE,
                        WARM_UP)

        with tf.Session() as sess:
```

```
            saved = tf.train.import_meta_graph(checkpoint_path + '.meta',
                                               clear_devices=True)
            saved.restore(sess, checkpoint_path)
            state = sess.graph.get_tensor_by_name('Actor/s:0')
            train_phase = sess.graph.get_tensor_by_name(
                                          'Actor/train_phase_actor:0')
            learned_policy = sess.graph.get_tensor_by_name(
                            'Actor/pi_online_network/pi_hat/Mul_4:0')

            r, info = agent.test(env, learned_policy, state, train_phase, sess)
```

Due to the stochastic nature of training, multiple initial seeds are trained on. This was found to be helpful in making algorithm and hyperparameter changes to get a sense of whether or not training improved. Three initial seeds are trained on, which meant that three models were saved once or if the training converged. Once a model is saved after using the *learn*() method of the DDPG class, then a new tensorflow session is created.

The saved model is restored as if it was deployed to a entirely new implementation using the specifically named output operations. The agent then uses the *test*() method, only computing the action using the actor network. The agent is tested on the same, specific tracks if lesson plans were instructed. If not, the agent is then tested on 25 random tracks depending on the seed. Once testing is completed, the information of its performance is saved in memory. The main loop repeats itself to instantiate new objects of the classes to be fed into a new object of the DDPG class inside a new tensorflow session.

The process repeats itself until all of the seeds have been trained and saved on. A summary of the mean and standard deviations of the performance on the seeds are saved to a text file. The performance includes test reward, number of episodes, number of times the models converged, and the clock time of training. Additional performance that was found useful were whether or not the agent made it to the loading dock goal for each seed. Lastly, the reward normalized for the length of the path was reported for each of the seeds. The script can be run using the following code snippet in a terminal.

```
>>> python3 DDPG.py
```

Various implementation details for the DDPG class and the main loop will now be discussed in order of computation. Once the program is run, the many hyperparameters to construct the neural networks in the respective classes are created as constants for the main loop. Other terms needed for the reinforcement learning algorithm are also first declared at the top of the script. For example,

the number of episodes get created as a variable and is passed to the *learn*() method of the DDPG class.

The main loop then begins by creating empty lists to be populated for the summary to be reported at the end. A folder gets created for the saved models. The environment is created using the gym package and the TruckBackerUpper-v0 module. The environment is unwrapped so the internal parameters can be accessed such as the track vector for normalizing the reward per length.

As mentioned before, the following process will loop for each seed. A new root directory gets created for the tensorboard log files, using the current time so the log files will not overwrite itself. In order to ensure determinism and repeatability when training, the initial seeds are set for *numpy*, *tensorflow*, the *random* module, and the environment. As discussed before, a new tensorflow session is created where all the objects for the classes are instantiated with the necessary information.

The script can handle two command line arguments for restoring a model or suggesting a lesson plan. If no arguments are given, then the script will continue to train from scratch and randomly spawn tracks. Since the checkpoint files storing the saved models include the string 'seed' in the file path, this signifies the computational graph should restore the latest checkpoint. The lesson plans are defined in a text file which may look like the following. The order of the inputs are $[x, y, \psi]$ in units of meters and degrees.

```
#[[25.0, 25.0, 225.0], [-25.0, -25.0, 180.0]]
[[25.0, 0.0, 180.0], [-5.0, 0.0, 180.0]]
```

The command line arguments can be included when running the script in a terminal like the following. The arguments are separated by a space. They can be included in any order and one can be used without the other.

```
>>> python3 DDPG.py models/sample_seed_0 single_fixed.txt
```

The csv module is used to parse the starting and ending positions and orientations for the path planner. The environment will take the information to spawn a manual course with the modified Dubins planner. If a row begins with a number sign, then it is ignored. The text file containing the lesson can have as many tracks as desired.

```
            if len(sys.argv) >= 2:
                for arg_idx, arg in enumerate(sys.argv[1:]):
                    if "seed" in arg:
```

```
                saver.restore(sess,
                        tf.train.latest_checkpoint("./models/" +
                        sys.argv[arg_idx+1] + "/"))
                print('~~~~~~~~~~~~~~~~~~')
                print('Model Restored')
                print('~~~~~~~~~~~~~~~~~~')
            if ".txt" in arg:
                with open(sys.argv[arg_idx+1], newline='') as csvfile:
                    readCSV = csv.reader(csvfile, delimiter='\n')
                    lesson_plan = []
                    for row in readCSV:
                        if row[0].startswith('#'):
                            continue
                        else:
                            lesson_plan.append(ast.literal_eval(row[0]))
                print('~~~~~~~~~~~~~~~~~~')
                print('Lesson Planned')
                print('~~~~~~~~~~~~~~~~~~')
                agent.lesson_plan = lesson_plan
```

learn() Method

Now it is time to begin the clock for training. The *learn*() method of the DDPG class is then used with the necessary information. After this completes, the current time is sampled to calculate the training time. The *learn*() method is really where the DDPG algorithm is performed. Before the algorithm is looped for a defined number of episodes, the file writer for the tensorboard functionality is created. In addition to this, two flags are set to False: 'start_rendering' and 'settle_model'.

For each episode, a new list is created so necessary logging information can be appended to them. The DDPG algorithm suggests to start by resetting the action noise. If a lesson plan was set as a command line argument, then a course will be randomly sampled from the text file. It is then used to implement a manual course in the environment. Next, the environment is reset to observe the initial states of the tractor-trailer system.

While no terminal condition is met, the episode will loop in the following manner. Training takes longer when the machine has to render the graphics, therefore, the application will not render by default. Sometimes it is useful to check up on the training. It also might be desirable to force the model to save in the middle of training despite not converging. Thus, the user is able to interrupt the running application for either of these conditions. If the user types into the command line 'render' or 'settle,' the flags will set to True.

```
                if sys.stdin in select.select([sys.stdin], [], [], 0)[0]:
                    command = input()
                    if command == 'render' or command == 'settle' \
                        or command == 'decay':
                        if command == 'render':
                            start_rendering = True
                            print('set render...')
                        elif command == 'decay':
                            self.decay_flag = True
                            print('set decay...')
                        elif command == 'settle':
                            print('settling..')
                            settle_model = True
                    else:
                        start_rendering = False
                        env.close()
                        print('hide render...')
```

If there are no interruptions, then the episode will loop as planned. The exploration noise is then sampled. The action is predicted from the actor neural network using the current state. Even though the action is determined, it is not necessarily used because guided training was used 50% of the time. If some random number between $[0, 1]$ is less than 0.5, then the LQR action is used. If this isn't true and the probability was decayed due to the convergence process, then the pure output of the actor is used as the action. Finally, if the former two conditions are not met, then the agent uses the action plus the exploration noise.

Next the agent takes a step in the environment by applying the action. The environment returns the next state, the reward associated with the transition, the done boolean, and any info associated with the transition. The sample transition is then added to the replay buffer. If the done boolean is True, then the info variable containing a dictionary of stats are printed to the console. This explains why the episode ended. It may be good because the agent reaches to the goal or it may be bad due to it jackknifing.

It may not be desirable to update the neural networks at every timestep, so an if statement checks to see if the current step matches the cadence of the desired training steps. The train steps for the DDPG algorithm should, however, be every step since it is a temporal difference method. In addition, training may not want to start before a 'warm-up' period to ensure the replay buffer has a sufficient diversity of samples. As will be discussed later, it will also not be desirable to change the weights again after the agent meets the desired performance. The if statement checks for all of these before proceeding with the training of the neural networks.

First, a minibatch is randomly sampled from the replay buffer. In order to calculate the target labels for the critic, the target network of the actor is used to predict the actions from the minibatch. The target network of the critic is then used to predict the Q values for the next states, but the trick is to use the actions predicted from the target network of the actor. This stabilizes training. The train phase for both target networks is set to True because it was found necessary for batch normalization if used.

A target label is created for each of the minibatch samples. If the sample transition consists of a goal criteria, then there is no future expected reward. The critic train method then gets called to perform backpropagation on the critic neural network. The train phase is set to True for batch normalization. This only matters if batch normalization is used.

Next, a vector of actions are predicted from the minibatch using the online actor network. Then the gradient of the Q values with respect to the predicted actions from the online actor network are retrieved from the critic class. Both of these operations have the train phase set to False for batch normalization if used. This information is used to update and train the actor neural network. The train phase for the actor backpropagation is set to True for batch normalization.

The weights and biases are adjusted every timestep in the online network. The parameters are slowly copied from the online to the target network. Next the current state is set to the next state so the episode can continue. The total reward is updated from the current step. The code snippet below shows the necessary steps in running the DDPG algorithm; it omits environment specific computations.

```
        for episode in range(EPISODES):
            done = False
            total_reward = 0.0
            ep_steps = 0
            self.action_noise.reset()
            s = env.reset()

            while not done:
                N = self.action_noise()
                a = self.actor.predict(s, train_phase=False)[0]

                if np.random.uniform(0, 1) < self.p:
                    a = np.clip(K.dot(s), env.action_space.low,
                                env.action_space.high)
                elif self.p < 0.5:
                    a = np.clip(a, env.action_space.low, env.action_space.high)
                else:
                    a = np.clip(a + N,
                                env.action_space.low, env.action_space.high)

```

```
                s_, r, done, info = env.step(a)
                self.replay_buffer.add_sample((s, a, r, s_, done))
                                if self.steps % TRAIN_STEPS == 0 and \
                    self.replay_buffer.size() >= WARM_UP \
                    and self.evaluate_flag == False:
                    s_batch, a_batch, r_batch, s__batch, d_batch = \
                                    self.replay_buffer.sample_batch(BATCH_SIZE)

                    a_hat_ = self.actor.predict_target_batch(s__batch,
                                                             train_phase=True)
                    q_hat_ = self.critic.predict_target_batch(s__batch, a_hat_,
                                                              train_phase=True)
                    y_batch = []
                    for i in range(BATCH_SIZE):
                        if d_batch[i]:
                            y_batch.append(r_batch[i])
                        else:
                            y_batch.append(r_batch[i] + \
                                           self.critic.gamma * q_hat_[i])

                    self.critic.train(s_batch, a_batch,
                                      np.reshape(y_batch, (BATCH_SIZE, 1)),
                                      train_phase=True)

                    a_hat = self.actor.predict_online_batch(s_batch,
                                                            train_phase=False)
                    qa_grads = self.critic.get_qa_grads(s_batch, a_hat,
                                                        train_phase=False)
                    self.actor.train(s_batch, qa_grads[0], BATCH_SIZE,
                                     train_phase=True)

                if self.steps % COPY_STEPS == 0:
                    self.actor.slow_update_to_target()
                    self.critic.slow_update_to_target()

                s = s_
                total_reward += r
                ep_steps += 1
```

Once an episode completes, information from it gets appended to the logging variables. The specific reward is calculated by normalizing the reward with respect to the length of the track. If the current specific reward is larger than the previous one of the last 100 episodes or less, then it will be set as the best specific reward. This will be discussed more in the convergence criteria section.

The script then prints out information to update the terminal. The episode number, reward, specific reward, and average maximum Q value are just some of the information presented to the developer. Some information is also useful to display in *tensorboard* to get a sense of the progress of training. Successful training should have a reward that increases as the number of episodes increase.

The output of the critic is useful to understand the progress as well. The average maximum Q value should also improve over time since the critic should give higher Q values for better actions taken.

The output of the actor network is also useful to visualize in *tensorboard*. If the average action of the actor plateaus and seems to continually pick the same action, something is probably wrong. The number of steps is also useful to visualize to get a sense of if the agent is surviving in the environment longer. The specific average reward was instrumental for this thesis to determine, from the history, if the training will likely converge soon.

In addition to having *tensorboard* report progress, it was also found useful in this thesis to report items relating to the convergence criteria such as the specific average reward to beat and the percent error change between the last 200 and 100 episodes.

Convergence Criteria

The convergence criteria is not a required implementation of the DDPG algorithm, but was found to be instrumental in increasing the objective productivity. This is an excellent example of machine teaching. A substantial amount of time was spent on convergence criteria, so algorithm 9 summarizes how it works. The implementation of this can be found in appendix C.

Algorithm 9 Convergence Criteria

```
1: if evaluate_flag is True then
2:   if goal and specific_reward >= 82% max(last 100 specific_reward) then
3:     Converged
4:   else
5:     Reset p ← 0.5
6:     decay_flag ← False
7:     decay_steps ← 0
8:     evaluate_flag ← False

9: if decay_flag is False then
10:  if mean(last 100 specific_reward) >= 82% max(last 100 specific_reward) and
11: perc_error(mean(last 200 specific_reward), mean(last 100 specific_reward)) <= .05 then
12:    decay_flag ← True

13: if decay_flag is True then
14:  if p <= 0.1 and
15: mean(last 100 specific_reward) >= 82% max(last 100 specific_reward) and
16: perc_error(mean(last 200 specific_reward), mean(last 100 specific_reward)) <= .05
17: and goal then
18:    evaluate_flag ← True
19:    p ← 0.0
20:  else if p <= 0.25 and sum(last 100 goal) <= 50 and episode > 100 then
21:    p ← 0.5
22:    decay_flag ← False
23:    decay_steps ← 0
24:  else
25:    Pass
```

The convergence criteria consisted of checking if the specific reward of that last 100 episodes was greater than or equal to 82% of the maximum specific reward. Since the track lengths and radii continually change during training, a specific reward is calculated with respect to the distance of the desired path. This is similar to trying to normalize the reward for different lengths. The specific reward is not fed to the neural network for training, rather it is used for human comprehension.

This idea was inspired from designing structural components for specific stiffness, where one tries to maximize the stiffness, but minimize the mass. The maximum specific reward that is evaluated against is determined from the preceding 100 episodes. The 82% percentage value is a knock-down factor that strikes a balance between ignoring outlier high specific rewards and having a sufficiently high average performance. The percentage is a tuning factor that must be determined for unique problems.

Another condition that was required in determining if convergence was met was if the percent error of the specific reward changed less than 5% between the last 200 episodes and 100 episodes. The average specific reward could exceed the 82% criteria of the maximum specific reward really

quick, however, the maximum is a moving target. The percent error check of less than 5% really signals when training has started to plateau. This was inspired by mesh convergence studies in Finite Element Analysis.

The amount of exploration should logarithmically decrease as the agent improves. More importantly, since an expert policy is used 50% of the time, reliance on guided training should decrease. The probability of selecting an expert policy, thus, started to decay once the average specific reward of the last 100 episodes exceeded 82% of the maximum specific reward of the last 100 and the trend started to plateau using the 5% criteria. The decay was set to $\lambda = 2.75e - 5$ to equate to approximately 100 episodes.

$$p = 0.01 + (0.5 - 0.01)e^{-\lambda \ decay\_steps} \tag{6.28}$$

It is entirely possible the agent begins to perform worse as it becomes more reliant on the neural network only. Thus, the probability of using the expert policy can reset to 50% if the probability is below or equal to 25% barring the number of episodes the agent made it to the goal in the past 100 is less than or equal to 50.

As mentioned earlier, the samples should be independent and identically distributed. Decaying the probability reduces the identically distributed criteria. However, training is not truly independent and identically distributed because the actor network's policy also changes over time. This changes the sample distributions already. Therefore, reinforcement learning is not truly independent and identically distributed because the neural networks produce the samples. Thus, decaying the probability is deemed sufficient, especially since it demonstrated the ability to improve results.

Lastly, it is important to note that the weights change after every step in the DDPG algorithm. This means that if the agent meets the convergence criteria even after decaying and the desire is to save the model, the weights change one last time before saving. This was found to be a problem. One would not necessarily get the performance during training due to an additional training step. Therefore, an evaluation episode was implemented such that saving the model did not occur unless the agent reached the goal.

The script continues in the main loop by appending the convergence logging variable with whether or not the training from the specific seed converged. The number of episodes is also appended to another logging variable. The model is then saved to the root directory for the training seed. The computational graph is then reset to clear it before restoring the model for testing. This is simply done to ensure the saved model is reusable.

test() Method

The *test*() method of the DDPG class takes the restored weights and biases and applies the learned policy to some new test scenarios. A new tensorflow session is created to restore the most recent training event. If a lesson plan was used, then the number of demonstrations will be however many tracks are in the lesson plan text file. If not, then the number of demonstrations will be 25 to get a sense of the population distribution.

For each demonstration, the *test*() method of the DDPG class is used to return the performance results, reward and any other necessary information. The reward is stored to a testing reward logging list. The minimal structure of testing the agent is displayed in the code snippet below, ignoring the restoring of the computational graph because it was discussed already.

```
            if agent.lesson_plan:
                n_demonstrate = len(lesson_plan)
            else:
                n_demonstrate = 25
            for ep in range(n_demonstrate):
                r, info = agent.test(env, learned_policy, state, train_phase, sess)
```

The *test*() method runs a single episode using the policy from the saved actor. The critic is no longer needed. If a lesson plan was used, then the script selects the track from the lesson plan text file in order of appearance. The environment is reset to observe the initial states of the tractor-trailer. The episode will continue as long as no terminal conditions are met. The action is selected from the learned policy using the current state. The train phase is set to False just in case batch normalization is used.

The agent then uses the action to take a step in the environment. This step returns the next state, reward, and any information regarding terminal conditions. If the episode is done, the information dictionary will be printed to the terminal. The current state is then set to the next state so the episode can continue. The total reward is updated with the current transition's reward. The process continues until a terminal condition is met. The test() method returns the total reward and the information dictionary with respect to the episode.

```
        done = False
        if self.lesson_plan:
            course = self.lesson_plan[self.lesson_idx]
            env.manual_course(course[0], course[1])
            self.lesson_idx += 1
        s = env.reset()
```

```
        total_reward = 0.0
        steps = 0
        while not done:
            #env.render()
            a = sess.run(policy, feed_dict={state: s.reshape(1, s.shape[0]),
                                            train_phase: False})
            s_, r, done, info = env.step(a)
            if done:
                print()
                for key, value in info.items():
                    if value:
                        print(key, value)
                print()
            s = s_
            total_reward += r
            steps += 1
        return total_reward, info
```

The main loop then resets the tensorflow computational graph so that training can begin for the next initial seed. Once all of the seeds have been used to train, save a model, and test, a summary of the process is printed to the terminal. It is also printed to a text file for review at a later time.

6.2.10 Google Compute GPU Cores

Having three additional instances on Google Compute to run training sessions on saved a substantial amount of time. First of all, the Nvidia Tesla P100 GPU felt twice as fast as the GEFORCE GTX 1080Ti on the personal machine. Plus at any given time, one parameter could be changed on each of the instances and have all running at once. This was extremely useful for determining trends. The specifications for the GPUs are found in Table 6.5.

Table 6.5: The 1080Ti and the Tesla P100 had the same number of CUDA cores.

Specs	GTX 1080Ti	Tesla K80	Tesla P100
CUDA Cores	3584	2x2496	3584
Memory	11GB	2x12GB	16GB
Compute Capability	6.1	3.7	6.0

One should caution whether compute platforms from one machine to another give comparable results. Originally, this research used the Tesla K80 on Google Compute because it was the cheaper option at $0.45 per hour. That was until some numerical instability was found caused by summation operations of all the different nodes. The same code on the desktop ran twice would result in the

same training results if the initial seed was set properly. If the same code was run on the Tesla K80 on Google Compute, the training initially looked similar but the results would drastically diverge!

This was a frustrating discovery because a table of different parameter changes were performed with both machines; this meant that the comparisons already finished were meaningless! Fortunately, it was determined that this was due to having a different number of CUDA cores. Since the GPU parallelizes the computations to speed training up, summation operations occur at different points when the number of CUDA cores are different. The Tesla P100 had the same number of CUDA cores as the 1080Ti, so the results were identical using these machines as long as the initial seed is set properly. The unfortunate thing was that the Tesla P100 cost a $1.46 per hour.

Once Google Compute is installed on a Linux machine, connecting to the Google Cloud Shell is possible from the terminal. Once connected to the Google Cloud Shell, the instance can be entered by selecting the server zone and the instance name like in the code snippet below.

```
>>> gcloud alpha cloud-shell ssh
>>> gcloud compute ssh --zone us-west1-b rl-p100-a
>>> screen python3 DDPG.py
```

Numerous things can cause the instance to crash on Google Compute such as losing internet connection, the host machine shutting down, or accidentally closing the terminal. It is suggested to use the screen command in the terminal to prevent the instance from accidentally shutting down for any of these reasons. The screen command will allow the program to continue to run on the server and allow the host machine to reattach to the running simulation. Do not get addicted to watching the progression of training, it is not good for your mental health! It is recommended to take advantage of cloud computing services in addition to a desktop. Let training run in the background and periodically check up on the progress.

6.3 Reinforcement Learning Evaluation

Training with the TruckBackerUpper-v0 initially began with how the environment was created– spawn random, new tracks at the beginning of every episode. The first reward function that was implemented included discrete information. At each time step, the agent would receive plus one reward per state error from the path based on a boolean check if the performance was comparable to what the LQR was capable of. If the agent met undesired terminal conditions like jackknifing, it would be penalized by a large amount. Training on completely random paths was posing ambitious, so the path from Figure 6.14 in a previous section was used as the baseline.

The agent started by jackknifing, but after 1000 episodes, the tractor appeared to follow the path. This is because the agent was being rewarded for the heading of the tractor as well using the discrete reward. After 2000 episodes, the agent started to prioritize the trailer on the path because it realized it could obtain more reward for the lateral position of the trailer and its orientation. When episode 2500 began, the performance noticeably got worse so training was halted. One of the problems with this reward function was that the rewards tended to grow large, making it more difficult for the agent to learn. It was also then decided to not reward for the tractor heading and only focus on the trailer.

Various other rewards were investigated such as primarily having negative rewards using the quadratic cost function also found in the LQR. After 1750 episodes, the agent learned to move away from the path, i.e. turn the wrong way and go out of bounds. Other ideas tested were to subtract a reward of one per timestep to punish the agent for taking a long time to get to the goal. The negative rewards, however, encouraged the agent to reach a terminal state as quickly as possible to avoid accumulating penalties.

A combination of positive and negative rewards were also tried. The agent was given continuous positive reward based on a probability density function from the goal. It was also given the quadratic cost function found in the LQR to ensure reward was also based on information from the states. After 1944 episodes, the agent learned to get close to the path with the trailer, but it often overshot the path and the trailer would end up perpendicular to the path. This reward actually gives reinforcement based on information the agent wouldn't necessarily observe, so it was ultimately dropped.

The probability density function was also used for rewarding the error states of the trailer since it would give positive, continuous information from the path. This was not proving fruitful, so the feedforward curvature was removed as an input to the neural network to reduce training complexity. The feedforward information is not an error and only helped marginally with the LQR.

The convergence criteria was originally set to be if the agent reached the goal 75 out of the 100 previous episodes. This was selected because that was the apparent amount the LQR could realistically do with the paths randomly spawned. The aforementioned trials were not meeting the criteria and would continue training for the set maximum number of episodes. Even after completing the maximum allotted episodes, the resultant neural network would not perform well.

Training was taking substantial time because the maximum time allotted was 160s and was also hardly making it to the goal. It was decided to simplify the scenario to make sure implementation

details were correct. The evaluation and training process was not linear, but can be best described with the flowchart below in Figure 6.20.

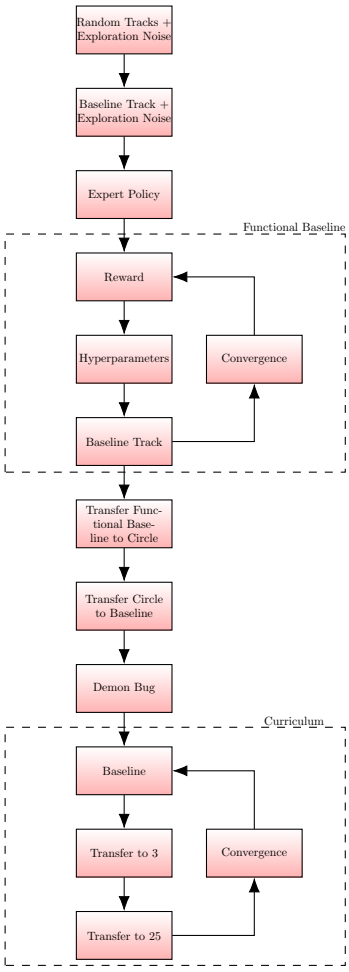


Figure 6.20: The training flowchart is presented to ground the topics that will be discussed in the evaluation section.

6.3.1 Achieving the Functional Baseline

The functional baseline was simplified to a single track as shown in Figure 6.21. The sanity check is a straight line to the goal, which spans 30m. The trailer is started $2m$ laterally from the path so the neural network has to learn to to minimize the errors from the path. The reason for initially simplifying the problem to such a simple scenario is to reduce the simulation time, hoping to overfit. With supervised learning, it is usually suggested to first overfit on a new problem to make sure the implementation is correct before scaling up to more data. Reducing the simulation time to 18s allowed for quick iteration to determine the reward function.

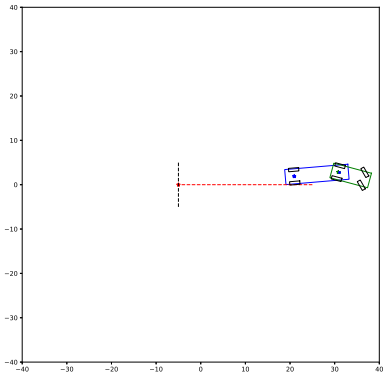


Figure 6.21: This simple track was initially trained for a sanity check before training on more difficult courses. The course can be recreated using Dubins Curves with $[25.0, 0.0, 180.0], [-5.0, 0.0, 180.0]$ **as the starting and ending configuration** $q(x, y, \psi)$

The reward schemes A-F that were described in the design section of this chapter were thus formalized at this point to devise a training evaluation matrix. Trying the reward schemes on even the functional baseline was posing difficult because the agent would hardly make it to the goal. It was theorized that the exploration noise used to train from was insufficient for the agent to learn this complex task.

Exploration Noise was Too Difficult to Learn from

The Ornstein-Uhlenbeck noise used for exploring proved to not be adequate for learning on the functional baseline. Lillicrap's implementations of the DDPG algorithm trained for a maximum of 2.5 million steps and worked for 20 different environments. Training with purely Ornstein-Uhlenbeck as the exploration policy for the functional baseline took more than 4 million steps and convergence was not apparent. This was more than $20,000$ episodes for what was thought to be a simple scenario. One can plot the reward over episodes and visually see when training appears to converge; this was not the case.

The noise was tuned in hopes of achieving faster training, but at the end of the day, maybe random noise with momentum is not the best teaching strategy for steering of a tractor-trailer. The DDPG algorithm is an off-policy method, which means it is capable of learning from an expert policy. As it turns out, an expert policy was already designed: the Linear Quadratic Regulator. Using guided training with the LQR was the first time the agent appeared to converge and the resultant neural network reached the goal during evaluation.

The objective then became determining the best method of employing an expert policy for training on the functional baseline, only using reward scheme A. Four different expert policy implementations are displayed here and are evaluated based on the specific ratio. The specific ratio is calculated as the average test reward divided by the average convergence episode. Table 6.6 displays the summary of the evaluation matrix over three seeds. If the agent achieved the goal on a certain trained seed, a circle 'O' is represented. If the agent failed to achieve the goal criteria, an 'X' is shown.

Table 6.6: Summary of various expert policies over 3 seeds

Expert Policy	Avg Test Reward	Avg Convergence Ep	Convergence	Avg Train Time [h:min:s]	Specific	Goal
K(s) v. $\mu(s)$ with p=0.5	167.397±51.489	1142±410	3/3	0:42:07±0:15:42	0.1466	XOX
$(1-\omega)\mu(s)+\omega K(s)$ with $\omega = 0.5$	137.781±52.154	762±454	3/3	0:27:17±0:16:20	0.1808	XOX
K(s) v. $(\mu(s)+N)$ with p=0.5	170.872±50.011	564±77	3/3	0:20:46±0:2:58	0.3030	OXO
$(1-\omega)\mu(s)+\omega K(s)$ v. $(\mu(s)+N)$ with p=0.5	172.824±52.914	906±189	3/3	0:32:45±0:07:34	0.1908	OOX

Before discussing the four expert policy implementations, one might wonder if training with only the LQR actions be used? The answer is unfortunately no. When this was attempted, the resultant neural network achieved a reward of -81.084 after 2600 episodes. If the DDPG algorithm is off-policy, can't training be accomplished in a supervised learning manner? Fujimoto explains why this is not the case in [83]:

> "*We demonstrate that this inability to truly learn off-policy is largely due to exploration error, a phenomenon in which unseen state-action pairs in the learning target are erroneously estimated to have unrealistic values, which may make them more attractive than observed actions. In other words, the uncertainty of unseen state-action pairs and generalization from function approximation, can cause poor value estimates and suboptimal action selection.*"

Basically, only using an expert policy 100% of the time for training cannot work because of the critic's function approximator. Due to the critic only seeing good states/actions, the function approximator could falsely make unseen state-action pairs more desirable.

The expert policy implementation in the first row of Table 6.6 toggles between using the LQR, K(s), and the prediction of the actor neural network, $\mu(s)$, with a probability of 50%. This method offers a chance for the training to steer in the direction of a known, good policy for getting the trailer

to the loading dock. It does, however, also train on samples generated from the actor network so the agent will learn on bad samples as well. This policy received the lowest specific ratio, which consisted of maximizing reward and minimizing the episodes until convergence.

The next row contains the blended policy from [30], using the blending factor of $\omega = 0.5$. At every timestep, the action consists of the summation of both the LQR and the actor neural network. Initially, the output of the neural network is close to zero, so training starts by primarily what the LQR would do. This guided training method is notable for having the lowest average test reward. After a while, a blending factor of 0.5 may be too large since both methods would suggest the correct action in theory. This would result in double the actual desired action. Lower values of the blending factor were also tried such as 0.01 and 0.001, but were not as fruitful.

The row highlighted in yellow is the guided training method that was ultimately chosen because the specific ratio was highest. With a probability of 50%, the LQR action is chosen or the actor prediction plus the Ornstein-Uhlenbeck noise is selected. This means that half the time is what the LQR would do and the other half is training just like the normal DDPG algorithm would. The LQR helps drive the training to the goal much sooner with samples from an expert. The stochastic noise allows for additional exploration and potentially bad samples for the critic to train on.

The expert policy implementation in the fourth row uses the blended policy, but also the Ornstein-Uhlenbeck noise 50% of the time. In other words, the normal DDPG algorithm is used half the time for exploring and potentially giving bad samples for the critic. This expert policy resulted in the highest average test reward, but the specific ratio suffered due to the number of episodes it took to converge.

Even though all four of the expert policy implementations converged on all three seeds, it is apparent that the agents did not necessarily achieve the goal criteria of reaching the loading dock within 15cm and with an orientation error of 0.1 radians. After the training on a seed of 0, the saved neural network would then be reloaded to the computational graph to test on the same functional baseline path. Even though it technically converged, it was still possible to not reach the goal. The process is continued for the seeds of 1 and 12 where various results occurred amongst the different expert policy implementations. This strongly suggested that a different reward function may be needed.

Determining the Reward Function

Determining the expert policy implementation was based on maximizing the specific ratio of test reward per the least amount of convergence episodes, barring the resultant neural networks

performed well when it came to test time. The reward function, however, becomes a little more complicated because the specific ratio will change depending on the reward scheme. Therefore, a new specific ratio was devised to promote reward agnosticism based upon an equivalent LQR reward:

$$\text{specific ratio} = \frac{1.0}{\text{convergence\_ep}|LQR_r - NN_r|} \tag{6.29}$$

In order to calculate the specific ratios in the evaluation matrix of Table 6.7, the equivalent reward that the LQR could achieve, LQR_r, is first calculated for each of the reward schemes. To account for the stochastic variation of training, the evaluations are performed over the seeds 0, 1, and 12. The average test reward from these seeds are then taken as NN_r to calculate the specific ratio. The equivalent LQR rewards vary for each of the schemes simply because as the reward changes, the LQR performance is deterministic. Table 6.8 is provided for reference to the various reward schemes described in the design section.

Table 6.7: Summary of various rewards over 3 initial seeds.

Reward Scheme	LQR Reward	Avg Test Reward	Avg Convergence Ep	Convergence	Avg Train Time [h:min:s]	Specific	Goal
A	211.556	170.872±50.011	564±77	3/3	0:33:42±0:08:50	4.3580e-5	OXO
B	258.566	80.438±106.761	742±194	2/3	0:22:56±0:02:57	7.5660e-6	XXX
C	272.825	128.996±140.078	833±235	1/3	0:26:37±0:07:34	8.3460e-6	XOX
D	82.766	-31.582±6.849	744±114	3/3	0:26:41±0:03:22	1.1754e-5	XXX
E	241.332	93.225±134.024	679±235	2/3	0:20:51±0:04:03	9.9439e-6	XXO
F	194.332	185.696±10.957	522±162	3/3	0:18:41±0:05:43	0.0002	OOO

Table 6.8: The reward evaluation matrix provided for reference.

Reward Scheme	Function
A	$r = 1.0 - 0.5(\frac{\lvert y_{2e}\rvert}{y_{max}})^{0.4} - 0.5(\frac{\lvert \psi_{2e}\rvert}{\psi_{max}})^{0.4}$
B	$r = 0.5e^{-\frac{(y_{2e}-\mu)^2}{2\sigma^2}} + 0.5e^{-\frac{(\psi_{2e}-\mu)^2}{2\sigma^2}}$
C	$r = \cos\psi_{2e} - \frac{1}{(1+e^{-4(\lvert y_{2e}\rvert - 0.5\ y_{max})})}$
D	$r = -\int_0^{t_f} (\underline{s}^T Q \underline{s} + \underline{a}^T R \underline{a}) dt$
E	$r = 0.5e^{-\frac{(d_{2e}-\mu)^2}{2\sigma^2}} + 0.5e^{-\frac{(\psi_{2e}-\mu)^2}{2\sigma^2}} - \int_0^{t_f} (\underline{s}^T Q \underline{s} + \underline{a}^T R \underline{a}) dt$
F	$r = 1.0 - 0.5(\frac{\lvert y_{2e}\rvert}{y_{max}})^{0.4} - 0.5(\frac{\lvert \psi_{2e}\rvert}{\psi_{max}})^{0.4} - \int_0^{t_f} (\underline{s}^T Q \underline{s} + \underline{a}^T R \underline{a}) dt$

The objective is to select the reward scheme which maximizes the specific ratio. Setting up the experiment this way lets the designer compare each of the rewards with a single number, which in

one way or another, encompasses the additional information in Table 6.7. The more times the agent achieves the goal, the higher the average test reward will be.

The reward scheme A converged all three times, however, did not achieve the goal for all three seeds during testing. On seed 0, the agent's steering policy reached the goal but had a slight jitter. The agent smoothed it's steering on seed 1, but did not steer enough to reach back to the path. The agent's steering policy on seed 12 was smooth, but had a little bit of hunting.

Reward scheme B is the Gaussian reward function. It converged two out of the three seeds, but unfortunately never reached the goal during testing. The agent jackknifed on seed 0, slightly ignored the distance on seed 1, and barely missed the goal on seed 12.

Reward scheme C provided continuous orientation error reward plus a softmax of the lateral error. It converged only one out of the three seeds, but at least achieved the goal once on the second seed. The agent also jackknifed on seed 0. On seed 1, the agent reached the goal with a damped, smooth steering policy. However, on seed 12, the agent had a bang-bang policy where it was switching the steering back and forth constantly. In the end, the trailer angle error was too high at the loading dock.

The reward scheme D was inspired by introducing the negative quadratic cost function, which is a reward. Surprisingly, it converged on all three seeds. It, however, performed extremely poorly because the average test reward was negative where as the LQR was able to at least achieve a positive reward since it reached the goal. The agent had a bang-bang policy again for seed 0, but hardly steered at all on seed 1. On seed 12, the agent appeared to ignore the distance from the path completely.

Reward scheme E attempted to combine the Gaussian reward with the quadratic cost function. It converged two out of the three seeds, but failed to reach the goal two out of the three tests. Unfortunately seed 0 jackknifed and on seed 1, the trailer angle error increased as it got closer to the goal. The agent, however, produced a smooth, damped steering policy on seed 12.

Reward scheme F combined reward scheme A with the quadratic cost function; this was the first time training was successful in converging on all three seeds and reaching the goal after testing the neural networks. This reward scheme resulted in the highest specific ratio, which is why it is highlighted in yellow on Table 6.7. The learned policies appeared to be quite different, which is the interesting part. On seed 0, the learned policy was smooth and damped. The learned policy on seed 1 appeared to have a small overshoot and had to correct. Lastly on seed 12, the learned policy was bang-bang all the way to the loading dock.

Various other reward functions were investigated such as adding small action costs and varying the factors in front of the terms for the trailer orientation and lateral error from the path. They did not improve upon reward F, so they are not presented.

Hyperparameters

The majority of hyperparameters were kept similar to the vanilla DDPG algorithm. Unlike the original DDPG algorithm, batch normalization and the L_2 regularizer were not used initially. Using reward F, the two methods were investigated in assisting with the training. According to Table 6.9, the addition of batch normalization and the L_2 regularizer degraded the training result.

Table 6.9: Batch normalization and the L_2 regularizer did not improve training. Remember the LQR performance would achieve a reward of 194.332 using reward F.

-	Avg Test Reward	Avg Convergence Ep	Convergence	Avg Train Time [h:min:s]	Specific	Goal
Reward F	185.696±10.957	522±162	3/3	0:18:41±0:05:43	0.0002	OOO
Batch Normalization	107.178±28.242	958±194	1/3	0:41:11±0:02:06	1.1977e-5	XXX
Batch Norm + L_2	129.890±1.894	869±184	1/3	0:37:26±0:07:48	1.7857e-5	XXX
L_2	133.971±87.709	565±221	3/3	0:19:50±0:07:57	2.9322e-5	XOX

The inclusion of batch normalization actually increased the average convergence episodes and resulted in an actor policy that did not achieve the goal criteria. Similar results occurred with the batch normalization plus the L_2 weight decay, but the specific ratio is only slightly improved from purely batch normalization. This is an interesting find because on the ContinuousMountainCar-v0 environment, batch normalization reduced the number of episodes to convergence. Only using the L_2 weight decay converged all three seeds, however, the agent only reached the goal once during testing.

A noteworthy hyperparameter that was investigated to be altered was the batch size because this directly influences the diversity of the update at each time step. Typically batch size is increased if unlearning occurs. It was thought that a batch size of 64 may be too small. Batch sizes of 256, 512, and 1024 were studied. Usually it is wise to change these hyperparameters by powers of 2 because it helps the GPU parallelize. Increasing the batch size increased the average test reward, but at the expense of increasing the clock time of training. It was found to actually slightly decrease the convergence episode, but the computer had to do many more computations with the increased batch size. Thus, it was decided that a batch size of 64 was still adequate to continue with.

After selecting the hyperparameters, expert policy, and reward function, the original baseline path was attempted again. The agent still struggled to learn the baseline path! It was theorized the convergence criteria was not sufficient for learning the more challenging path.

Determining the Convergence Criteria

The convergence criteria for determining the expert policy implementation and reward function in the previous sections actually consisted of the number of times the agent achieved the goal as well as the root mean squared errors from the path for the trailer. The agent needed to reach the goal 90 times out of the past 100 episodes. In addition to this, the average orientation error of the trailer in the past 100 episodes needed to be less than or equal to 0.0897 radians, which is what the LQR could do. Simultaneously, the average lateral position of the trailer from the path needed to be less than 1.0542m. This was working for the functional baseline of a straight line, starting with a 2m lateral offset. It was not necessarily working for other paths, so improving the convergence criteria was investigated on the functional baseline again.

Decaying the probability of using the expert policy was found to be beneficial, in fact, could result in a test reward exceeding that of what the LQR could do. Once the agent begins to demonstrate convergence, the probability of using the handicap of the expert policy decreases logarithmically. This introduces more samples from the actor policy into the replay buffer, which appears to assist with promoting stable resultant weights. The trick is to first introduce some good samples from the expert policy, then slowly reduce the dependence on the expert. The logarithmic decay decreases the probability every time step, inspired by approximately decaying over 100 additional episodes. In fact, once the decaying starts, it was that it is better to solely use the actor policy without noise.

It is entirely possible to decay and have the training performance degrade, which is okay because it means that the training has not stabilized. It is not desirable to let training continue without additional exploration when training does not appear to be approving, so a method of resetting the probability of using the expert policy to 50% was implemented.

When using the decay of the probability of using the expert policy, it was found the decay process could be initiated sooner with a warm-up period. A warm-up period allows the replay buffer to populate to a size of for example, $10,000$ samples, before backpropagation is calculated to change the weights. This creates a diversity of samples, which is a better starting place for training.

Simply increasing the number of goals required in the past 100 episodes was not promoting a better result and took longer to train. Introducing the root mean squared error criteria required a priori knowledge of what the LQR could do over specific tracks, which is not in the spirit of training

to best ability of the agent. Thus, using the number of goals and root mean squared error criteria were abandoned. Using the reward as the convergence criteria was actually found to be faster and resulted in a better performing neural network.

Table 6.10 displays the relevant training results on the functional baseline using reward scheme F. Now that the agent is able to actually achieve a higher reward than what the LQR performance can give, the specific ratio was modified to add an equivalent term if this situation is true.

$$\text{specific ratio} \mathrel{+}= \text{bool}(NN_r > LQR_r)\frac{(LQR_r - NN_r)}{\text{convergence\_ep}} \tag{6.30}$$

Table 6.10: The convergence criteria benefits from decaying the probability of using the expert policy and warm-up.

-	LQR Test Reward	Avg Test Reward	Avg Convergence Ep	Convergence	Avg Train Time [h:min:s]	Specific	Goal
No decay, no warm-up	194.332	185.696±10.957	522±162	3/3	0:18:41±0:05:43	0.0002	OOO
decay until p=0.1 using rms criteria	194.332	192.977±3.990	658±217	3/3	0:19:25±0:05:40	0.0011	OOO
Warm-up and decay until p=0.1 using rms criteria	194.332	194.820±2.375	602±66	3/3	0:18:22±0:02:02	0.0042	OOO
decay reward >= 180, converge reward >= 194.332, warm-up, p=0.1	194.332	201.720±1.607	496±63	3/3	0:15:06±0:02:00	0.0152	OOO
decay reward >= 180, converge reward >= 194.332, warm-up, p=0.05	194.332	200.065±0.772	509±110	3/3	0:15:01±0:03:22	0.0116	OOO

Simply converging based on maximizing the reward is common in literature, however, also requires knowledge of the environment. In addition, the number of steps in the episode will increase for a longer path. In turn, this increases the total reward possible for one path versus another. Therefore, the idea of using a specific reward normalized for path length was introduced. This kept training to continually improve with respect to its own experience. Plus this assists with training on various path types. The specific reward was not used for updating the weights, only for evaluating convergence. Literature was not found for determining when reinforcement learning has converged; this was a challenging research area in this thesis.

It was, however, still possible to begin decaying the probability of using the expert policy early on in the training using the average specific reward comparison of the past 100 episodes. Thus, the percent error change of less than five percent is used to indicate if the training improvement has stabilized. The maximum specific reward should continually increase until an approximate optimal policy has been discovered. The percent error change indicates when the performance starts to plateau, or an optimal policy has been discovered. That is, until the maximum specific reward increases again.

The last topic within the convergence criteria to mention is the fact that the weights change at every time step. Despite getting a better sense of when the neural networks were converging, it was still possible for the trained actor policy to perform poorly during test time. It was realized that since the weights changed one last time before saving the model, it was not guaranteed to be the stable (converged) model that was desired. Therefore, an evaluation episode is used before saving the weights and biases.

In summary, if the model appears to converge at a high enough specific reward and has stabilized because of the percent error, the training begins to decay the probability of using the expert policy. If the probability decays to less than or equal to 10% and still performs well, then an evaluation episode is performed where the actor neural network is solely used. If the model achieves the goal and the specific reward of the this episode exceeds the desired specific amount, then the model is saved. If not, however, the training will reset the decay probability and start the process over again.

Transfer Learning

After revamping the convergence criteria and obtaining results exceeding the LQR on the functional baseline, the baseline track was attempted again. The agent still, however, did not show signs of learning progression after $20,000$ episodes. It was theorized this was due to the difficulty of having to learn a policy which must handle both a straight line and curves. Transfer learning was used by taking the learned weights from the functional baseline straight path and initializing them to a circle path. Table 6.11 displays how successful and quick transfer learning was from a straight path to a curved path. Figure 6.22 shows the circle path used, along with the other paths used in transfer learning.

Table 6.11: Transfer learning looked promising for extrapolating to new paths.

Avg Test Reward	Avg Convergence Ep	Convergence	Avg Train Time [h:min:s]	Goal
738.428±13.493	81±3	3/3	0:12:24±0:00:38	OOO

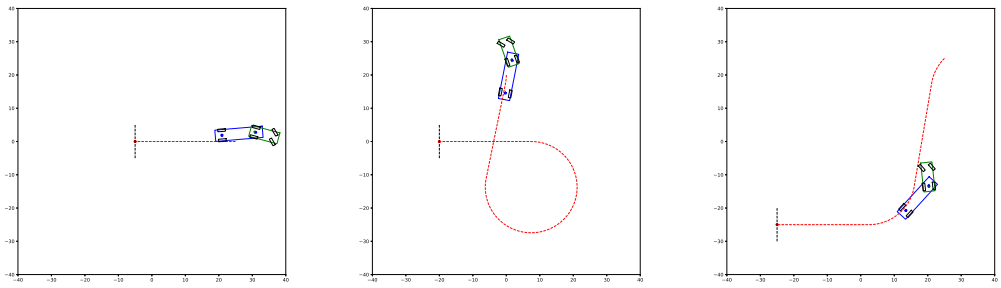


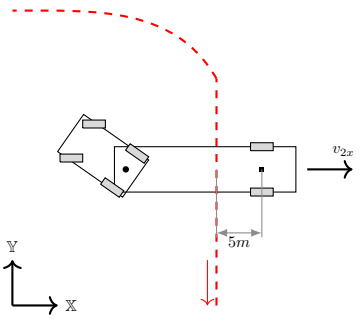
Figure 6.22: Transfer learning is shown with complexity increasing from left to right.

The obvious next step was to transfer the weights from the circle path to the baseline path with straights and curves. It, unfortunately, did not work so well. This shed light on the fact that there must be something more going on with complex paths. When visually evaluating the training progression on the baseline path, a significant amount of time was spent far from the path.

6.3.2 The "Demon Bug"

Training on the baseline track was difficult. As it turns out, the neural networks see significantly more bad states when the tractor-trailer is farther away from the path on the more complex paths. The expert policy of the LQR really only performs well when the error from the path is small because the gains are designed using a model with the equilibrium points at zeros. There were no terminal conditions when the distance from the path grew large, so the replay buffer simply continued to be populated with bad samples.

Why was the agent continually finding it beneficial to increase the distance from the path? The answer stems from the very fact that the control policy is focusing on lateral error from the reference point on the path, and not longitudinal. The reward function F gives a higher reward when minimizing the lateral error. However, it is entirely possible to achieve zero lateral error when the trailer is perpendicular to path as shown in Figure 6.23.



$$\begin{bmatrix} \cos 0 & \sin 0 & 0 \\ -\sin 0 & \cos 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}$$

Figure 6.23: The reward function will award more for minimizing the lateral error, but it is possible to achieve this even if it is not the desired effect.

If the trailer becomes perpendicular to the path, the rotation will always report a zero lateral error from the reference point. For example, if the distance from the rear axle to path is 5m, the position is actually 5m away in the global $\mathbb{X}$ coordinate. The passive rotation takes the trailer orientation, which is 0°, and reports the lateral error in the vehicle coordinate system. Since the controller is constant velocity, the longitudinal is not used as a state and thus, not in the reward function.

$$r = 1.0 - 0.5(\frac{|y_{2e}|}{y_{max}})^{0.4} - 0.5(\frac{|\psi_{2e}|}{\psi_{max}})^{0.4} - \int_0^{t_f} (\underline{s}^T Q \underline{s} + \underline{a}^T R \underline{a}) dt$$

It is true that the reward function above also takes into account the angle orientation error. However, the agent was falling into a suboptimal policy because it was finding it could obtain more reward by exploiting this flaw in the design. It discovered the lateral error could be minimized by increasing the angle orientation error, which must have been very confusing for the reinforcement learning agent. Obviously it would get more reward if both the lateral and angle orientation error of the trailer was smaller, but certain paths may have transitions that steer the reward to be higher by disregarding the importance of the angle orientation error.

Reinforcement learning is notorious for resulting in weird, unexpected, and inhuman solutions to problems. Peng's study in [84], a humanoid agent was training in simulation to pitch a baseball at a target. Peng stumbled upon an awkward, but functional policy: running towards the target with the ball. Irpan also mentions an example where an agent was trained in a boat racing environment called CoastRunners. Instead of trying to finish the race as fast as possible, it found it would earn more reward by circling in the same spot so it could repeatedly drive over three targets as they respawned. As it turned out, the reward was based on giving breadcrumb rewards of these targets along the path. The agent actually achieved a higher reward than human players even though it did not result in the desired policy. Irpan has an excellent viewpoint on this very topic in [85]:

> "*I've taken to imagining deep RL as a demon that's deliberately misinterpreting your reward and actively searching for the laziest possible local optima. It's a bit ridiculous, but I've found it's actually a productive mindset to have.*"

The fault is actually in misspecifying rewards. This goes back to the idea getting what one incentivizes, not necessarily what one wants with the Cobra Effect. It is also similar to convex optimization problems where one can get awkward results from poorly designed cost functions and constraints.

The solution to fixing this "demon bug" was to terminate the episode if the trailer lateral or orientation error exceeded a certain amount. The agent would be punished with losing a reward of 100 just as if it were to jackknife. Further investigation discovered that the LQR could really only handle 4.6m of lateral error on the baseline track before the tractor-trailer would jackknife. The largest lateral error would also change depending on the complexity of the spawned path. Thus, the maximum lateral distance the trailer was allowed before terminating the episode was set to 5m.

The angle orientation error limit needed to be set such that the agent was not incentivized to become perpendicular. The trick was to find a large enough value that would not trigger a premature termination of the episode. After rigorous studying, this was determined to be 45°. The factors in front of the terms in the reward were altered to weight the angle orientation more, however, did not appear to alleviate the problem.

$$r = \begin{cases} -100 & \text{if} \quad y_{2e} \geq 5 \\ -100 & \text{if} \quad \psi_{2e} \geq 45^\circ \end{cases} \tag{6.31}$$

After incorporating the additional negative terminal conditions and rewards to impose some desired constraints, the baseline path with minor curves and straights was solved! The agent was able to start from random initial weights, converge, and reach the goal criteria on seeds 0, 1, and 12. Figure 6.24 displays the mean specific rewards over seeds 0, 1, and 12. The shaded area represents the minimum to the maximum, displaying the variation in the progression of learning when the seed changes on the same path–the baseline path.

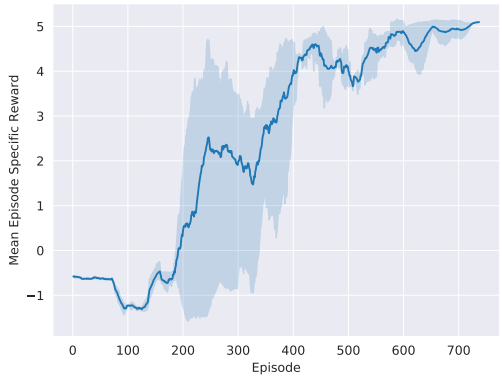


Figure 6.24: The mean episode specific return is shown over the three seeds. The shaded area represents the min to max.

Plots of the various parameters logged through *tensorboard* only on seed 0 are provided in Figures 6.25, 6.26, 6.27, 6.28, and 6.29. The data is smoothed with a factor of 0.6, which is overlayed on the raw data.

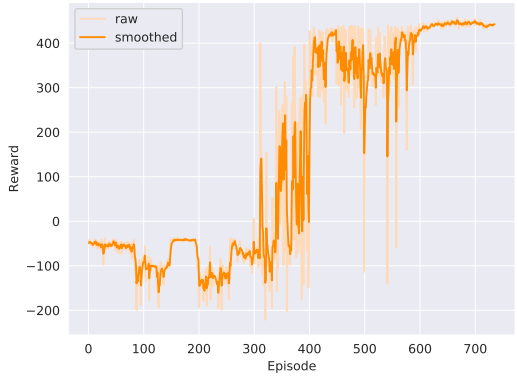


Figure 6.25: The total reward is plotted for each episode. It begins with the agent obtaining negative rewards hovering around -100 likely due to it learning that jackknifing is bad. At approximately episode 350, the agent learns to start following the path and starts to earn positive total rewards. The agent receives a total reward of 310 if it hits the loading dock, but does not meet the goal criteria. It will, however, receive a total reward of 410 if it meets the goal criteria. As the training stabilizes according to the convergence criteria, the training is stopped.

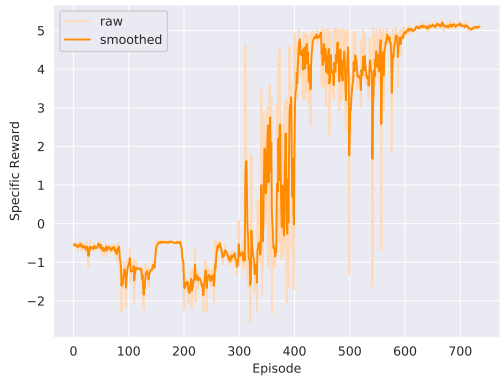


Figure 6.26: The specific reward normalized for path distance is also plotted. The trend is exactly the same as the regular reward, but the value is different. This signal is cleaner to look at once the path starts changing, but the path is consistent in each episode for this case.

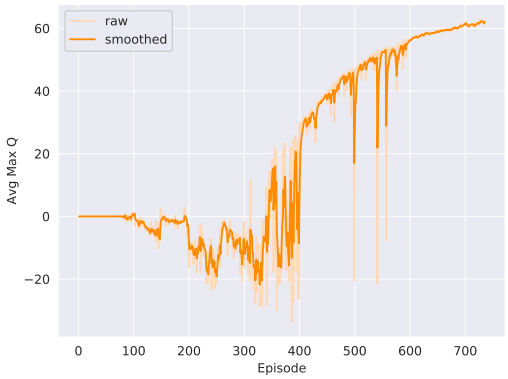


Figure 6.27: At each timestep, the critic neural network outputs a Q-value of the selected action from the actor. This resembles the maximum Q-value because at that instance, this Q-value critiques the action taken by the actor. For each episode, the average of the maximum Q-values are plotted. As the number of episodes increases, the maximum Q-values should increase to indicate that the critic is improving alongside with the reward.

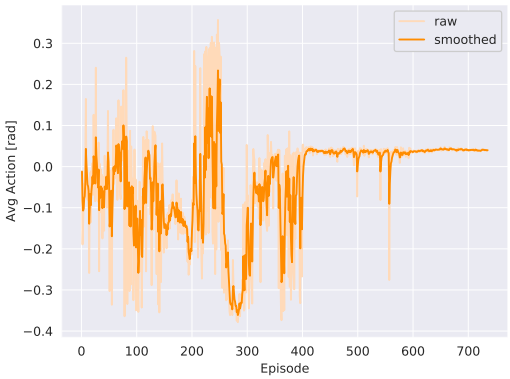


Figure 6.28: This plot of the average actions taken over each episode give a sense of how the actor policy is doing. This is the output of the actor neural network. Depending on the path, the average actions will be different along with varying noise. If the output of the actor is plateauing at a non-zero number, it is likely the agent is using the same action over and over again. This would indicate that the actor has underfitting.

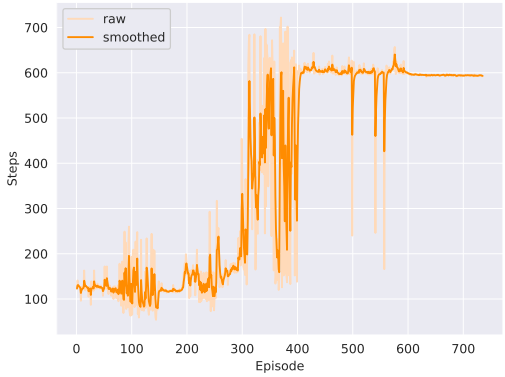


Figure 6.29: Plotting the number of steps per episode is useful to get a sense of the agent's ability to avoid terminal conditions. On the baseline track, it appears that it takes approximately 600 steps to reach the goal in an efficient manner.

The objective, however, is still to learn a policy that can extrapolate to whatever path Dubins Curves generates. Training on just one path is insufficient for extrapolating to the paths that lie ahead. Considering it took an average training time of 2 hours and 16 minutes to converge on a single path, it was decided to utilize transfer learning to increase the complexity of the paths.

Transfer learning was found beneficial as a starting place for the weights from one path to a different one, but it was theorized that it would be possible to un-learn the policy it knew before when it would continually train on only the new path. This means there is potential to learn to really only be good at the one path, but not others. In other words, this could result in a neural network

that does not generalize well. This can be mitigated by designing curriculum learning properly, another form of machine teaching.

6.3.3 Curriculum Learning

Curriculum learning consists of lessons to teach the DDPG agent in an incremental fashion when a difficult environment does not allow for convergence of learning. It utilizes transfer learning, however in this case, the agent would train on a variety of paths to promote better generalization. Due to the complexity of the task, the lessons increase in difficulty as well as the number of paths that it must converge on.

Considering there was success on the baseline path (lesson 1) which contained minor straights and curves, the weights were transferred to the next lesson plan: 1) an S-track 2) a hook and 3) a small curve leading to a straight. The paths are displayed in Figure 6.30. The paths that were selected consisted of turns in both directions, longer straights, but also one that had a quick transition from a curve to a straight.

In Table 6.12, the specific reward of the resultant neural network for each seed is displayed. The reason is because the one with the highest specific reward was chosen as the weights to proceed forward to train on three tracks.

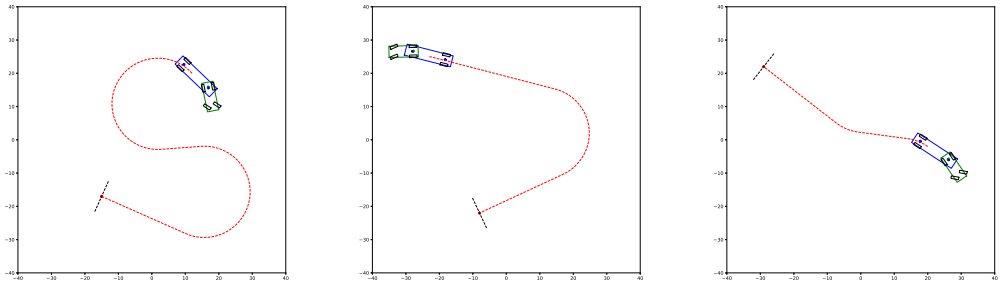


Figure 6.30: Three paths were chosen to transfer the weights to in order to increase the difficulty as well as having to converge on three paths. The starting and ending configurations will be found in the file named three_fixed.txt in appendix C.

After an average of 28 minutes on the three tracks, the agent converged on all three seeds. The agent reached the goal on all the new paths, except one shown below. Since seed 1 had the highest specific reward with 5.6466, the weights and biases were loaded into the model training on 25 tracks.

Table 6.12: Progression results of training with lesson plans over seeds 0, 1, 12.

Lesson Plan	Avg Test Reward	Avg Convergence Ep	Avg Train Time	Specific Reward	Goal
1 track	438.680±4.330	719±24	2:16:36±0:13:35	5.1039 5.0813 4.9885	1/1 1/1 1/1
3 tracks	561.309±14.849	104±2	0:28:21±0:00:31	5.6450 5.6466 5.0991	3/3 3/3 2/3
25 tracks	598.910±27.729	379±201	2:08:37±1:10:15	5.7460 5.6141 5.1497	25/25 25/25 13/25
random	N/A	N/A	N/A	N/A	N/A

The mean episode reward is plotted using the three seeds when the agent was training on three different paths. Figure 6.31 is shown to display the reward the agent received, but it is difficult to gauge convergence because the different paths can result in different maximum rewards allowable. Thus, Figure 6.32 is shown to suggest it is much clearer to determine convergence using the reward normalized by the distance of the path.

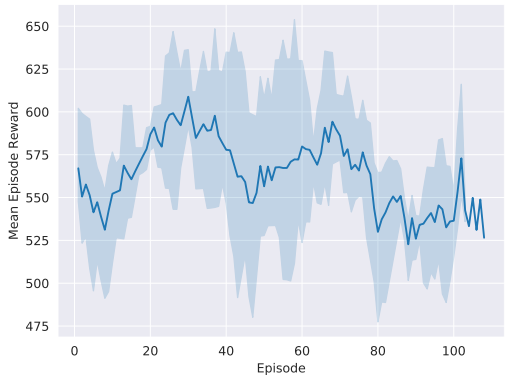


Figure 6.31: The plot of the mean reward for the three seeds does not clearly display the agent has converged when the paths continue to change.

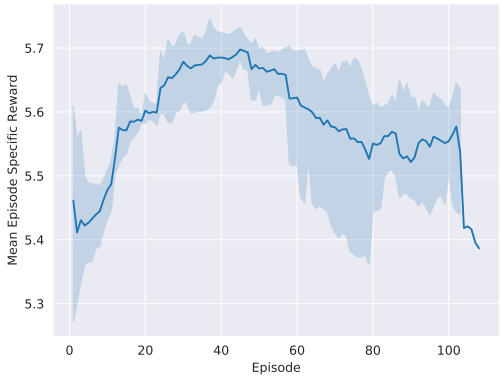


Figure 6.32: The plot of the mean specific reward for the three seeds, however, actually indicates convergence at approximately 104 episodes. The shaded area indicate the minimum and maximum from all three seeds. The reason there is no shaded region after this episode is because seed 12 actually took slightly longer to converge.

Determining what paths belonged in the lesson plan containing 25 tracks was very difficult. The process consisted of randomly spawning paths, visually checking if the expert policy (LQR) could reach the goal, then saving the starting and ending configurations into a text file. The text file is read in by the script containing the DDPG classes and randomly selects one of the paths to train on. It was imperative that these paths were more challenging than the previous lesson, but also had a diversity of paths that Dubins Curves would create. Care was taken to ensure these paths were generally longer or had sharper transitions. Figure 6.33 displays all twenty five tracks.

Transferring the weights from three tracks to 25 tracks converged after an average of 379 episodes! It took longer, but this lesson plan required it converge on 25 tracks the neural network has never seen. Figure 6.34 presents the mean specific reward over three seeds on the 25 tracks. Seed 0 actually converged after 532 episodes and seed 1 converged after 511. However, seed 12 took only 95 episodes to converge–so there is a large standard deviation on this.

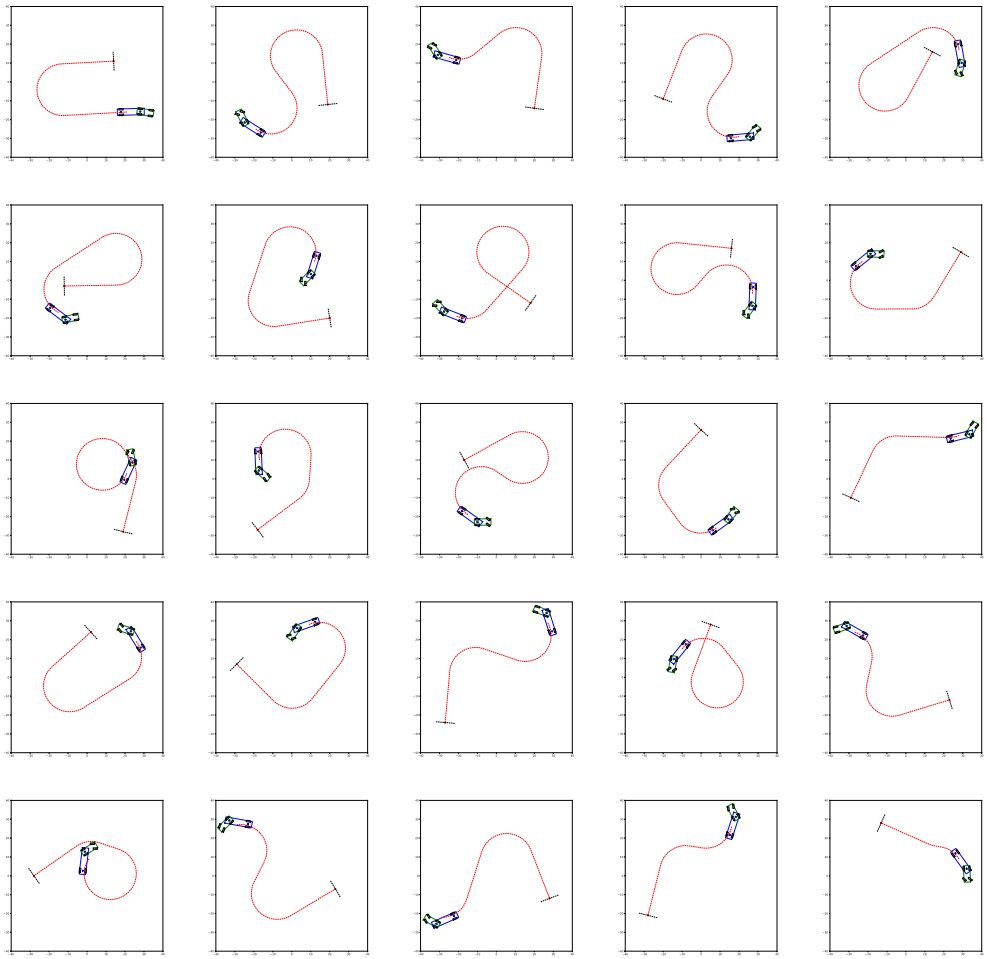


Figure 6.33: The next lesson plan consisted of 25 tracks, carefully chosen such that the expert policy would not jackknife. The starting and ending configurations will be found in the file named lesson_plan.txt in appendix C.

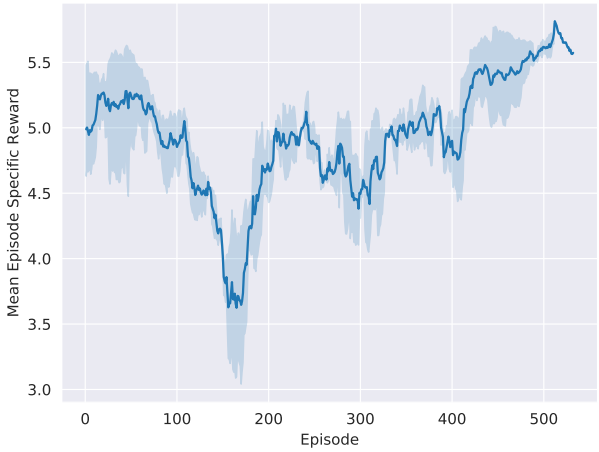


Figure 6.34: The plot of the mean specific reward for the three seeds on 25 tracks appears to be much more noisy, but this is why a convergence criteria was set up the way it was.

Seeds 0 and 1 learned a policy that was able to reach the goal criteria on all twenty five of the tracks it has seen. Seed 0 had the highest specific reward. The objective is still to create a policy that can generalize to any path that is given to it. Twenty five entirely new paths were generated to evaluate the generalization of the chosen neural network using seed 9. The DDPG agent reached the goal 22/25 times with a specific average reward of 5.5906. For a point of reference, the LQR was evaluated on those same tracks and received a specific average reward of 4.9891 and only making it to the goal 21/25 times. This was encouraging, but it was still desired to be able to train on and converge on completely random paths.

Random Tracks

The weights from the most successful seed from the 25 tracks were initialized to begin training on completely random tracks. A new track would spawn every episode to generate samples for the replay buffer. The replay buffer would likely never see the same track again and the hope was that the weights were matured enough to eventually be able to converge on a policy that could generalize to a multitude of path types. This, unfortunately, did not happen. After training for $10,000$ episodes on seed 0, the resultant specific test reward was a mere 4.8371. Using a similar set of test tracks from seed 9 as before, the agent only reached the goal nine out of 25 times.

After training on 4000 episodes, the weights were settled on seed 1 to result in an agent that reached 14 out of the 25 test tracks. Interestingly, stopping training after 1282 episodes on seed 12 resulted in an agent that reached 22 out of 25 test tracks and with the a slightly higher specific test reward of 5.4034. This trend somewhat alluded to the fact that as the agent trained on more random paths, its general policy performance degraded.

As training would progress, it would sometimes appear the agent was really close to converging. A method was implemented to manually trigger the training to begin decaying the probability of using the expert policy. This would set a flag which would begin the convergence process. The specific rewards would initially begin to decrease if tampered with at the correct time. This was used to help guide the training and trigger an artificial convergence process. This ultimately did not result in better performance when training on the random paths.

It was discovered that the agent would sometimes reach the loading dock extremely fast. It was possible for the Dubins Curves to spawn a path that overlayed on top of the loading dock as shown in Figure 6.35. Thus, the path generation was further constrained by checking if any of the relevant reference points on the path was greater than or equal to 5m from the goal point. Furthermore, the goal calculation was not performed unless the trailer orientation was within $45°$.

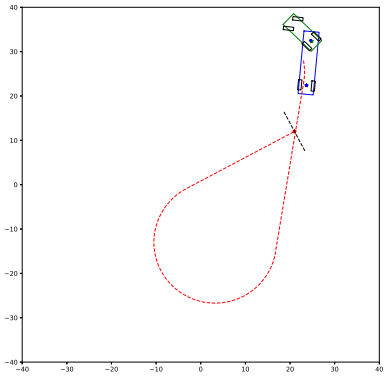


Figure 6.35: When training on random paths, it was discovered some faulty paths could be created. Therefore, a fix was implemented and training on random was continued.

Unfortunately, the overall performance of training on random paths was still not working. When further evaluating the types of random paths that could be spawned, it was discovered that the expert policy could not even achieve the goal for every path. The LQR would tend to jackknife in paths where sharp S-transitions occurred between two curves as shown in Figure 6.36

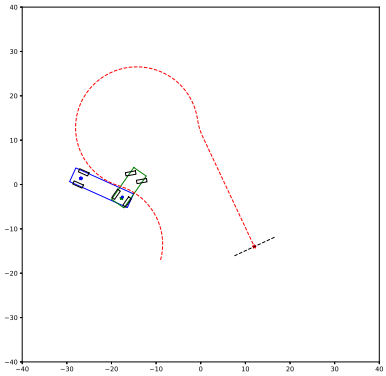


Figure 6.36: The Dubins Curve path planner could generate paths that even modern controls struggled with. The LQR struggled with sharp transitions between two curves.

It turns out that using a kinematic car driving forward to plan paths (Dubins Curves) for a trailer to reverse on may not have been the best path planner. The paths that get generated say nothing about the feasibility of the trailer being able to follow the path. This is where a planner such as Model Predictive Control can really impose constraints and prune the trajectory bundle for paths that do not jackknife. In reality, autonomous vehicles may be planning 5 seconds ahead to

achieve a more realistic trajectory based on the current pose. The intention of this thesis was to evaluate controllers on the same paths, not its ability to replan.

At this point, a substantial amount of money was spent using Google Compute. All things considered, it was realized that the achievements with DDPG using transfer learning on a total of 29 tracks was impressive ($1 + 3 + 25 = 29$). It actually resulted in a decent policy was learned from 29 tracks because it was able to generalize comparable to the LQR on 100 new paths it had never seen before. The detailed comparison results will be provided in the next chapter.

6.3.4 Summary

The DDPG algorithm was implemented on the challenging problem of backing up a tractor-trailer to a loading dock. Majority of the hyperparameters were kept similar to the vanilla DDPG aglorithm, but a few modifications were found necessary. Batch normalization and the L_2 weight decay from the original DDPG paper were not used. Since the DDPG algorithm is off-policy, an expert policy was used to guide the training. The expert policy was the LQR, which the DDPG agent only improved upon. The expert policy is what finally resulted in learning a policy which achieved the goal criteria.

As harder paths were investigated, transfer learning was found to be beneficial in progressing and generalizing on multiple path types. Determining the reward and convergence criteria really was an iterative process. Reinforcement learning is notorious for creating inhuman solutions to problems, but it is now understood this is simply because of misspecifying rewards for the desired results just like convex optimization problems. Using a more robust path planner that actually utilizes a tractor-trailer model to create driveable paths is suggested for continued work. The qualitative and quantitative analysis of the generalization of the learned DDPG algorithm is discussed next.

Chapter 7

PERFORMANCE RESULTS

This chapter presents the findings of how the DDPG policy performs on robustly backing up a trailer compared to the baseline of the LQR controller. It focuses on how well the controllers perform when the trailer changes from the model that is used to tune the LQR or train DDPG. First, the metrics for analyzing the performance are discussed. Then the generalization results are discussed when changing the trailer parameters of length, hitch length, and velocity. To further evaluate controller robustness, sensor noise and controller frequencies are altered.

7.1 Metrics

The chosen goal criteria for trailer to reach the goal are when 1) the distance from the rear of the trailer is within 15cm of the center of the loading dock and 2) the trailer angle is within 5.73° of the loading dock orientation. Many of the presentations at the Automated Vehicles Symposium in San Francisco in 2018 suggested that most of the sensor requirements for passenger vehicles were aiming for 5-10cm accuracy. The 15cm distance from the goal was selected to remain competitive, yet realistic.

Trigonometrically, a $8.192m$ trailer can be at 1.05° from the loading dock orientation if the tractor hitch is on the path and the rear of the trailer is within the 15cm region.

$$\sin \frac{0.15}{8.192} = 1.05^\circ \tag{7.1}$$

The tractor hitch may not necessarily be on the path since the controllers are focusing on the trailer. Therefore, the region of tolerance for the trailer angle was increased to 5.73° or 0.1 radians. Figure 7.1 displays a visual of the regions for the goal criteria. The figure is not to scale, but provides an understanding of what a successful backing up of the trailer is for the controllers.

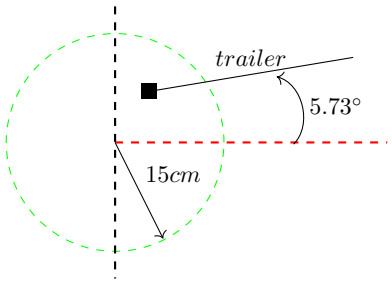


Figure 7.1: The goal criteria can be thought of as a range of tolerance for the trailer pose with respect to the loading dock. This figure is not to scale.

Reaching the goal is a requirement, however, this study focuses on the precision along the path as well. Two primary path-following metrics were chosen: the root mean squared error of the trailer lateral error (y_{2e}) and trailer angle orientation error (ψ_{2e}) are analyzed over the entire episode. Wu's study of controller performance similarly used the L_2 norm [39], however, some information can get lost with just averages. Therefore, the maximums are also displayed. The comparison of controller performance is made using the following metrics in Table 7.1:

Table 7.1: Performance Metrics

rms ψ_{1e} [rad]	rms ψ_{2e} [rad]	rms y_{2e} [m]
max ψ_{1e} [rad]	max ψ_{2e} [rad]	max y_{2e} [m]
min d_2 goal [rad]	min ψ_2 goal [rad]	boolean goal

The tractor angle orientation errors are also reported to display how excessive the controller had to reorient the tractor to move the trailer. Just because the trailer reaches the goal region does not mean the episode ends. Instead, the episode continues until the rear end of the trailer crosses the line representing the loading dock. Therefore, the metrics additionally report a minimum distance (d_2) and an angle orientation (ψ_2) that is actually less than the requirement.

A single parameter like trailer wheelbase or hitch length are changed for 100 tracks, holding all other parameters constant at the nominal values. The performance of the LQR and DDPG controllers are tested on the same path before moving onto the next track. The next value in the list of the changed parameter is altered in the environment and ran again for the same 100 tracks. The process continues until all the parameters are changed. The parameters changed are as follows in Table 7.2, where the nominal parameters the controllers are tuned for or trained on are highlighted in yellow.

Table 7.2: The changed parameters are displayed below, which are varied from the nominal case highlighted in yellow.

L_2 [m]	h [m]	v [$\frac{m}{s}$]
8.192	0.228	-2.906
9.192	0.114	-2.459
10.192	0.0	-2.012
11.192	-0.114	-1.564
12.192	-0.228	-1.118

Table 7.3 displays the control loop parameters changed to verify robustness. The sensor noise can only be increased from nominal, but the controller frequency is evaluated with decreasing and increasing it. The nominal conditions are, once again, shown in yellow.

Table 7.3: Changed Control Loop Situations

sensor noise [σ]	controller frequency [ms]
0.0	1
0.03	10
0.04	80
0.05	400
0.06	600

A summary of the terminal conditions will be reported in a table for each of the parameter changes. The number of goals will be tallied as a positive terminal condition, but so will the number of times the trailer reached the loading dock without meeting the goal region–Fin. Negative terminal conditions will also be summed, which include jackknifing, too large a distance from the path, and too large of angle from the path. Recall these are defined as $\theta > \pm 90°$, $y_{2e} \geq 5m$ and $\psi_{2e} \geq \pm 45°$.

An interesting note about reporting reinforcement learning algorithm results are that the performances vary depending on the final weights determined from the initial seed used for training. Many popular papers such as Mnih [28] and Lillicrap [67] report on only five different seeds. In fact, according to Colas [77], Mnih reports the top five seeds. Many of these papers even fail to say which seeds are used to report the results they achieved. The importance of the stochastic effect should not be underestimated.

In this work, the initial seeds are chosen here as 0, 1, 12. Three seeds are chosen due to how long and expensive computation can cost on a student's budget. Only the seed with the highest performing results will be displayed in charts, but a table will be provided for results from all three seeds to show this variation in the case with changing trailer length.

Colas suggests to use Welch's t-test for reinforcement learning algorithms when statistically comparing data with different variances. Welch's t-test assumes the scale of data measurements are continuous, representative of the population, independent from one another, and the data is normally-distributed.

$$t = \frac{x_{diff}}{\sqrt{\frac{s_1^2+s_2^2}{N}}}$$
$$v = \frac{(N-1)\cdot(s_1^2+s_2^2)^2}{s_1^4+s_2^4} \tag{7.2}$$

When using Welch's t-test, the desire is to be able to reject the null hypothesis. The null hypothesis is that the means are the same. One can reject the null hypothesis if the p-value is less than or equal to α. This just means that there is enough sample data to say the means of the samples are different, i.e. statistically significant. If $t < 0$, then the mean of reinforcement learning dataset is higher with 95% confidence. If $t \geq 0$, then the mean of the modern controls dataset is higher with 95% confidence.

As scientific as Welch's t-test would be, some of the comparison metrics required more samples to be statistically significant. Due to this, this thesis refrains from reporting these particular statistics and focuses on the results analytically.

7.2 Varying Trailer Wheelbase

The first step was to evaluate the two controllers using the trailer length the LQR gains were designed with and the neural network was trained on. This is highlighted in yellow in Table 7.4. Then the trailer lengths were varied to see how well they would perform on the same 100 tracks. The following data uses the weights saved from an initial seed of 0 because it performed the best on the 25 tracks during the evaluation phase.

Surprisingly, reinforcement learning with the nominal trailer length was able to extrapolate a decent policy to a multitude of new paths by training on just 29 tracks prior. Not only that, it appears the DDPG policy reaches the goal more than the LQR. This is even true for when the trailer lengths are varied. In fact, the DDPG never jackknifes when the trailer lengths are varied from the nominal trailer length it was trained on. The fact that it does not jackknife demonstrates reinforcement learning's ability to impose constraints on a problem such as punishing the agent for jackknifing. The Linear Quadratic Regulator does not have this ability because there is no re-planning or altering of gains as Model Predictive Control may have.

Table 7.4: Summary of the LQR and DDPG on 100 new random tracks, varying the trailer lengths.

Condition	-	8.192m	9.192m	10.192m	11.192m	12.192m
Goal	LQR	74	77	77	77	75
	DDPG	82	83	84	83	83
Fin	LQR	75	77	78	77	76
	DDPG	93	96	98	99	96
Jackknife	LQR	23	23	22	23	23
	DDPG	0	0	0	0	0
Large Distance	LQR	0	0	0	0	0
	DDPG	2	3	1	1	3
Large Angle	LQR	2	0	0	0	0
	DDPG	5	1	1	0	0
Out of Bounds	LQR	0	0	0	1	2
	DDPG	0	0	0	1	2

The problem, however, is that the rms error from the path tended to be higher than the LQR when the trailer lengths were varied. The error from the path is root mean squared over an entire episode such as shown in Figure 7.2. The desired error for each of the states is zero, which is displayed in the dashed red. The LQR and DDPG errors are similar, but slightly different resembling that DDPG, in fact, learned a different control policy.

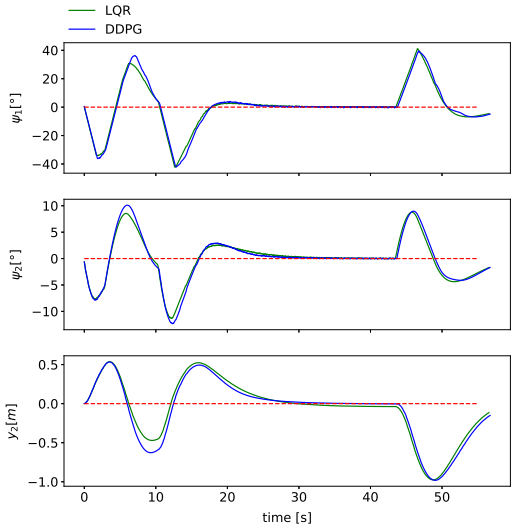


Figure 7.2: The root mean squared (rms) error over an entire track is used so the result is positive.

The resultant root mean squared error from a single path can be represented as a bar graph in Figure 7.3. The DDPG policy shows higher rms error for all the states when compared to the LQR controller for this specific track.

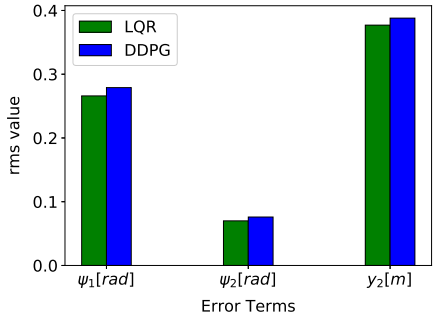


Figure 7.3: Comparison for the root mean squared error over a single path.

The rms values are averaged over the total number of paths to get a better sense of extrapolation. Figure 7.4 displays bar graphs for the averaged rms errors with standard deviation, comparing the controllers at each wheelbase. The rms error for ψ_{2e} in the nominal case is 0.069 ± 0.017 radians for the LQR, which is less than the DDPG error of 0.070 ± 0.021 radians. The rms error for y_{2e} is 0.421 ± 0.074m for the LQR, which is also less than the DDPG error of 0.423 ± 0.092m.

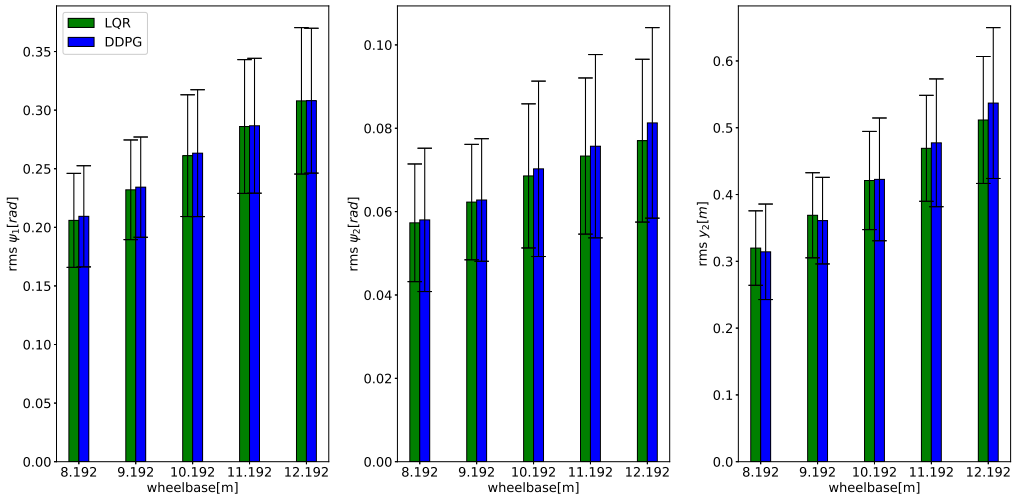


Figure 7.4: Considering only the runs where both the LQR and DDPG reach the goal, the rms errors are averaged for each wheelbase.

It is apparent that the average root mean squared errors of the states were higher for the DDPG trailer angle orientations and lateral error. This is the case for the nominal case with $10.192m$, but also increased as the wheelbase increased. As the trailer wheelbase is decreased, the DDPG policy actually performed better than the LQR controller. What is more distracting about these bar graphs

is that as the wheelbase decreases, both controller performances improve. Overall, the LQR appears to result in a better precision by majority of the states–even when the trailer wheelbase is altered.

The generalization ability of the controllers can be represented as deviation from nominal in radar graphs. The radar graphs are used because it is difficult to infer generalization from the bar graphs. The idea is that the controllers were designed with the nominal trailer in mind, so a smaller percent error from nominal for a new trailer length is better. The percent error is calculated for both controllers, each with respect to its performance on the nominal trailer length.

$$\begin{aligned} \%_{error}^{LQR} &= \left| \frac{rms_{experimental}^{LQR} - rms_{nominal}^{LQR}}{rms_{nominal}^{LQR}} \right| \\ \%_{error}^{DDPG} &= \left| \frac{rms_{experimental}^{DDPG} - rms_{nominal}^{DDPG}}{rms_{nominal}^{DDPG}} \right| \end{aligned} \tag{7.3}$$

The way to interpret the radar graph is to understand that the percent error from nominal would be zero if the nominal trailer length of $10.192m$ was plotted. Considering there are multiple trailer lengths that were evaluated, the controller that generalized better is the one with less surface area on the radar graphs. The percent error or deviation from nominal for the trailer metrics show that the LQR does, in fact, have a smaller surface area amidst the changing trailer lengths in Figure 7.5.

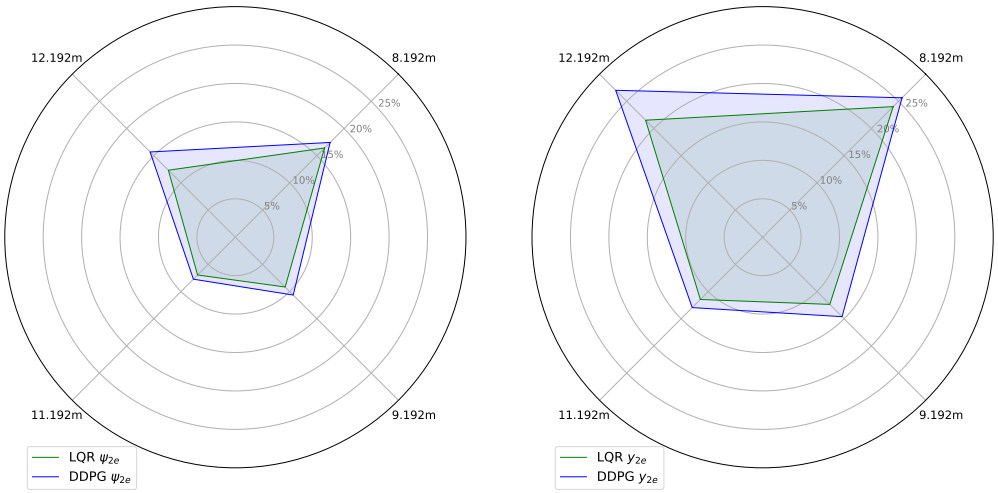


Figure 7.5: The radar graphs show the percent error from both the nominal LQR and nominal DDPG.

Evidently, there is a trade-off between precision and imposing non-linear constraints between the LQR and DDPG controllers. The LQR generalized better when looking at percent error from the nominal LQR and nominal DDPG for root mean squared error averages. This meant that the precision overall was better with the LQR. However, the DDPG policy generalized better across many trailer wheelbases, demonstrated by the fact that it never jackknifed. The LQR actually

only jackknifed more as the wheelbase was changed. The DDPG controller resulted in weights that learned to avoid jackknifing conditions even when the trailer lengths were altered. It learned a policy that more reliably follows the path to the loading dock.

The plots in Figures 7.4 and 7.5 consider only the runs where both the LQR and DDPG reach the goal because it provides a clearer picture of the performance along the path. Without it, the standard deviation is extremely large. It is also not fair to compare the controllers' rms performance when one reaches the goal and the other jackknifes. Filtering for the consistent runs in the nominal case resulted in 77 samples where both controllers reached the goal condition out of 100. Filtering for these samples improved the statistical significance of the metrics near the goal, but not for the rest of the metrics. The number of consistent goal-reaching runs varied slightly for the wheelbase scenarios, so it is yet another reason why Welch's t-test was not used.

Only evaluating the root mean squared errors can hide information, so the maximums are also plotted. Figure 7.6 displays the maximum errors for the same single path from Figure 7.2 to augment the rms information.

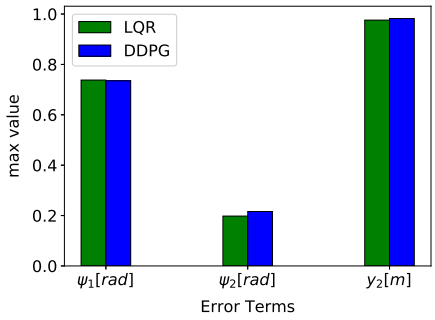


Figure 7.6: The controllers are juxtaposed for the maximum errors in a single path.

Next, the maximums are averaged over the total filtered runs where both controllers reach the goal, just like before. The bar graphs are displayed once again with standard deviations, except that this time, the average of the maximums are displayed for each wheelbase in Figure 7.7. Evidently, the bar graphs for the average maximums suggest that the DDPG controller has larger maximum errors for each of the states, for majority of the wheelbases. This trend is much clearer, so radar graphs are not utilized for the maximums.

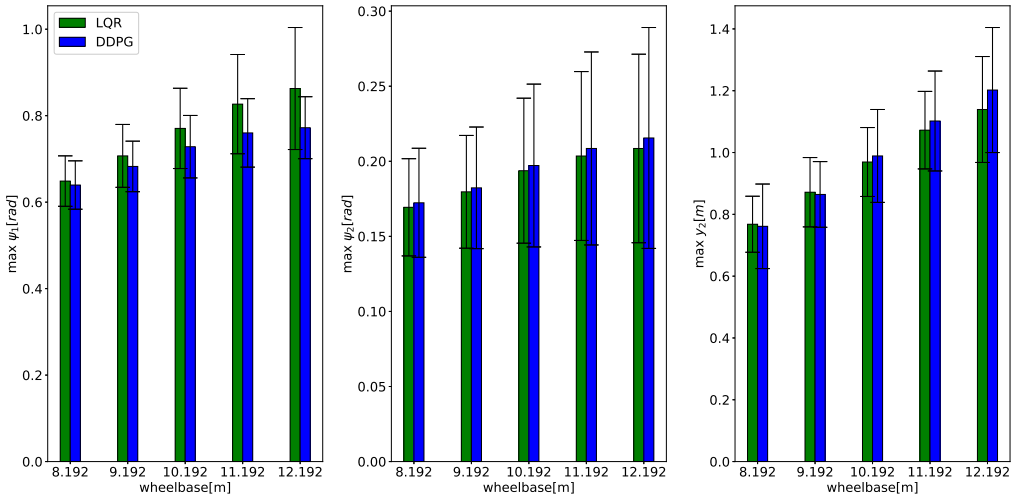


Figure 7.7: Considering only the runs where both the LQR and DDPG reach the goal, the maximum errors are averaged for each wheelbase.

When the trailer reaches the goal region, the tractor continues to reverse the trailer until the rear most point of the trailer crosses the loading dock line. The minimum distance of the trailer and the trailer angle orientation error from the loading dock can be displayed in a bar graph for a single path as in Figure 7.8.

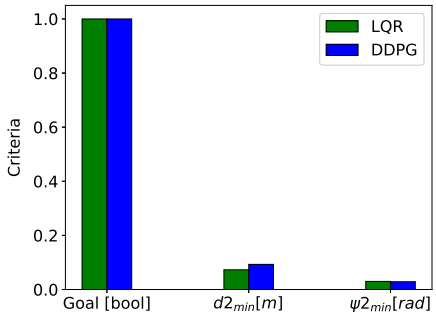


Figure 7.8: Starting from a random trailer pose, the bar graph below demonstrates the controllers' abilities to end near the loading dock.

On average and even with changing the trailer wheelbase, the DDPG policy resulted in a more accurate angle orientation of the trailer at the loading dock as shown in Figure 7.9. This is true with the exception of the $12.192m$ wheelbase. The DDPG policy for minimizing distance from the dock, however, was more prone to degradation as the wheelbase changed from nominal.

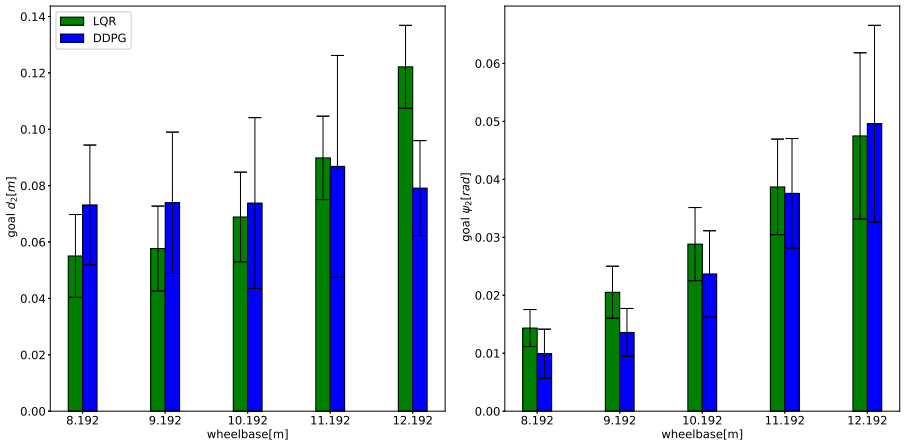


Figure 7.9: Considering only the runs where both the LQR and DDPG made it to the goal, the DDPG made it closer to the goal than the LQR.

Each of the generated reference paths end in a straight line leading up to the loading dock. The figure above suggests the DDPG controller prioritized minimizing the angle orientation to the path when on a straight. The average angle orientation to the goal was $1.38 \pm 0.4°$ for DDPG in the nominal case! The DDPG algorithm learned a policy taking into consideration that it should never jackknife, so the precision along a curved path must hinder to prevent exceeding angle orientation errors. Ultimately, imposing a non-linear constraint to not jackknife in the reward function caused DDPG to have better precision in straighter paths than in curved ones.

A 3D surface of the actions plotted over the state space of the trailer lateral error and angle orientation can shed light on the control differences between LQR and DDPG. Figures 7.10 and 7.11 display 3D surfaces generated from a single path. The DDPG surface has less defined peaks and valleys, suggesting a more complex policy is required to ensure jackknifing does not happen.

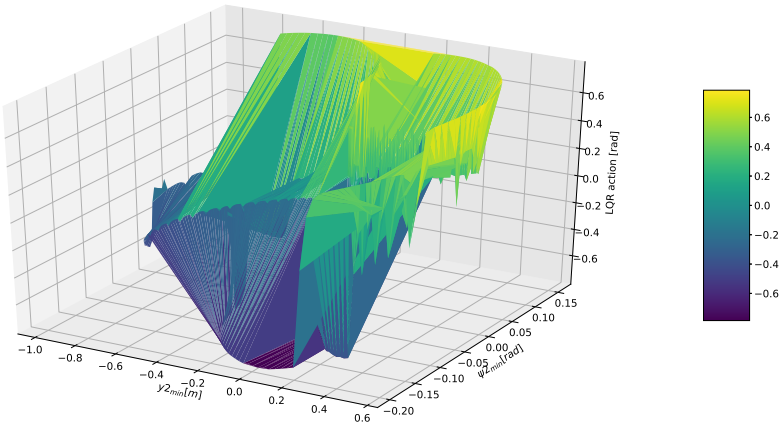


Figure 7.10: 3D plot of the LQR action surface

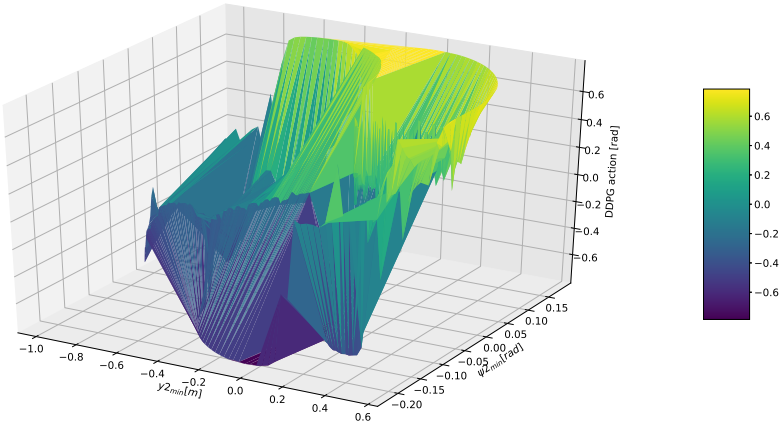


Figure 7.11: 3D plot of the DDPG action surface

Plotting the Q-value surface over the trailer states gives a sense of how the critic scores actions taken by the actor. It shows the respective cost-to-go function during one run. The critic is not necessarily used once the model is deployed, but it is sometimes useful to understand what this curve looks like. It will look different depending on the path and the action selected, however, should display a visual of the complex control problem for reversing a tractor-trailer.

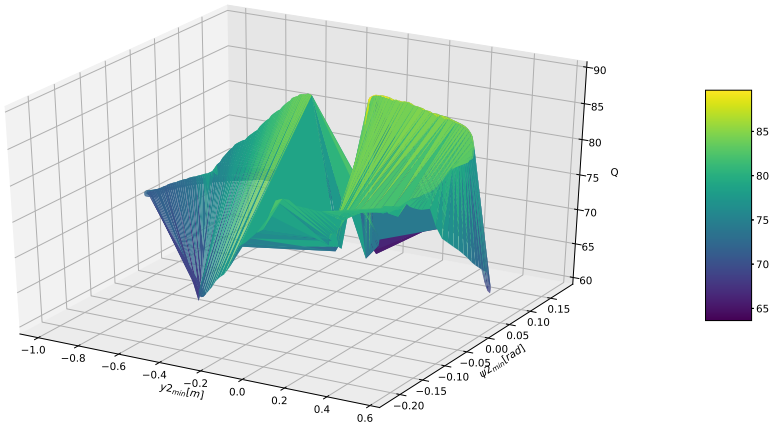


Figure 7.12: 3D plot of the Q surface

It was stated before that the results may vary depending on the saved weights due to different initial seeds. Table 7.5 below reports the stochastic variation and compares the metrics on the nominal cases. The metrics are filtered for runs where both controllers reach the goal and then averaged. The metric for goal is the sum of the number of times the goal was reached without any filtering.

Table 7.5: Stochastic Variation across 3 seeds on the nominal case over multiple paths.

Metric	LQR	RL-seed 0	RL-seed 1	RL-seed 12
rms ψ_{1e} [rad]	0.261 ± 0.052	0.263 ± 0.054	0.255 ± 0.048	0.253 ± 0.052
rms ψ_{2e} [rad]	0.069 ± 0.017	0.070 ± 0.021	0.070 ± 0.018	0.068 ± 0.017
rms y_{2e} [m]	0.421 ± 0.074	0.423 ± 0.092	0.448 ± 0.095	0.427 ± 0.094
max ψ_{1e} [rad]	0.771 ± 0.093	0.728 ± 0.072	0.698 ± 0.067	0.708 ± 0.094
max ψ_{2e} [rad]	0.194 ± 0.048	0.197 ± 0.054	0.196 ± 0.053	0.194 ± 0.049
max y_{2e} [m]	0.969 ± 0.111	0.989 ± 0.150	1.022 ± 0.140	0.919 ± 0.160
min d_2 goal [m]	0.069 ± 0.016	0.074 ± 0.030	0.085 ± 0.025	0.088 ± 0.012
min ψ_2 goal [rad]	0.029 ± 0.006	0.024 ± 0.007	0.033 ± 0.009	0.025 ± 0.008
goal	79	86	82	46

Considering this work used an initial seed of 0 for the comparison, a quick summary is discussed. Varying the trailer wheelbase from the nominal trailer the LQR was designed for and the DDPG was

trained on is meant to evaluate both controllers' ability to generalize its performance. It is known that the control performance should degrade if the system is different than what it was designed for with the Linear Quadratic Regulator. After reviewing the results, the LQR appears to generalize better when looking at percent error from the nominal. On average, the LQR follows the path closer and with better precision than DDPG does.

However, the DDPG was impressively able to extrapolate to a multitude of path types and without jackknifing after training on just 29 tracks. Analyzing the data in this section shows that an added benefit of the DDPG algorithm compared to the LQR is the ability to impose constraints such as preventing jackknifing. The DDPG controller reached the goal criteria much closer than the LQR, but it is likely due to the paths ending on a straight path. This means that the learned constraint probably affected the DDPG's performance on curves. Another reason may be due to the very nature of using a function approximator for learning this complex control problem.

7.3 Varying Hitch Length

In the trucking industry, the drivers also run into situations where the hitch needs to be moved for weight distribution or tight parking scenarios. The fifth wheel can be adjusted further or closer to the tractor center of gravity depending on the trailer cargo. With the kinematic model, the hitch length is in reference to the tractor rear axle. A hitch length of zero is on top of the rear axle. A negative hitch length moves the connection in front of the rear axle whereas a positive hitch length moves it behind. When the hitch is behind the rear axle, this can be thought of as modelling a truck with a ball hitch for passenger vehicles. Table 7.6 displays the summary of the runs on the same 100 tracks, but now varying the hitch lengths.

Table 7.6: Summary of the LQR and DDPG on 100 new random tracks, varying the hitch lengths.

Condition	-	0.228m	0.114m	0.00m	-0.114m	-0.228m
Goal	LQR	80	80	79	78	77
	DDPG	88	88	86	86	84
Fin	LQR	80	80	79	78	78
	DDPG	100	99	99	99	98
Jackknife	LQR	20	20	21	22	21
	DDPG	0	0	0	0	0
Large Distance	LQR	0	0	0	0	1
	DDPG	0	1	1	1	1
Large Angle	LQR	0	0	0	0	0
	DDPG	0	0	0	0	1
Out of Bounds	LQR	0	0	0	0	0
	DDPG	0	0	0	0	0

Once again, the DDPG policy learned to control the trailer autonomously so it does not jackknife–even when varying the hitch length! As the trailer mounting position was moved aft of the rear axle, the LQR with the fixed gains found it easier to maneuver. The same is true for the fixed weights and biases for the DDPG policy. From first glance, one might believe the opposite to be true since it is further away from the tractor center of gravity. However, when going in reverse, perhaps pushing the trailer with more leverage makes it easier. Geometrically, this means a smaller change in tractor position results in a larger change in the trailer.

The rms errors from the path are also plotted in bar charts for both controllers when varying the hitch length from the rear axle in Figure 7.13. These bar charts are created by filtering for the tracks which both controllers reach the loading dock. The trailer angle orientation error is higher for DDPG despite moving the hitch fore and aft of the rear axle. On the other hand, the lateral error of the trailer from the path appeared to decrease as the hitch point moved aft of the rear axle for DDPG.

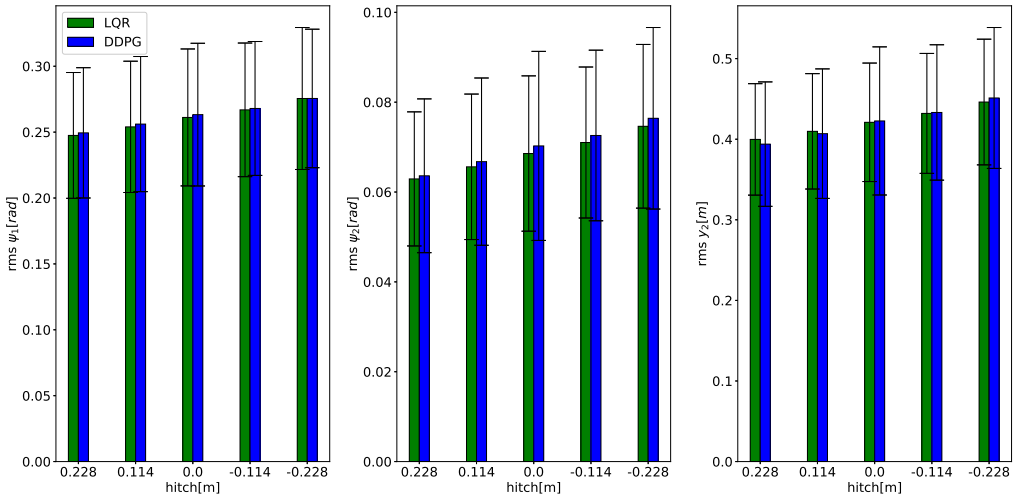


Figure 7.13: The bar charts show the controllers rms performance as the hitch length is changed.

The same rms data can again be represented as percent error from the nominal with respect to its own controller in radar graphs in Figure 7.14. Overall, the controllers are less sensitive to changes in hitch length than wheelbase. Since the LQR controller covers less surface area in the radar graphs, it was able to handle changes to the hitch length more proficiently than the DDPG policy along the path.

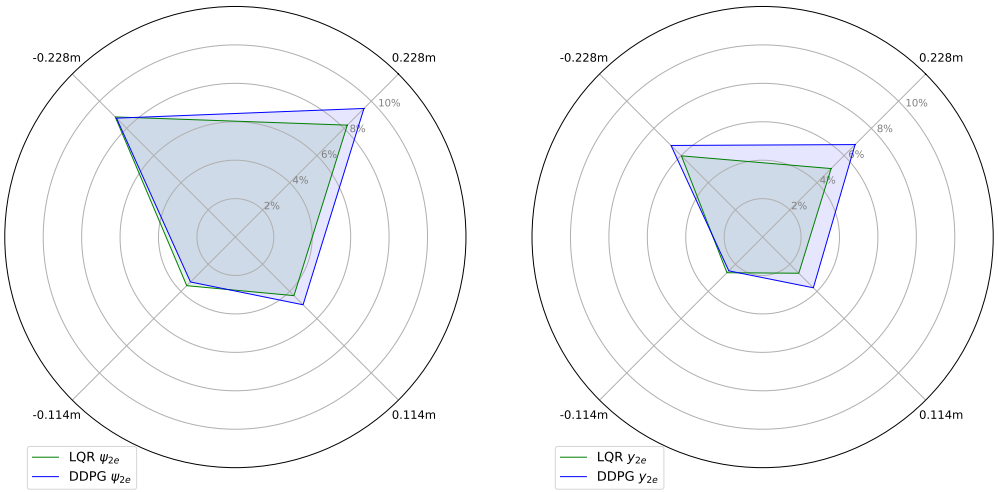


Figure 7.14: The radar graphs show the percent error from both the nominal LQR and nominal DDPG when varying hitch length.

The rms values can hide information, so the average maximum errors are plotted over the tracks when the hitch lengths are altered in figure 7.15. The LQR still has lower average maximums than DDPG even when the hitch length is changed.

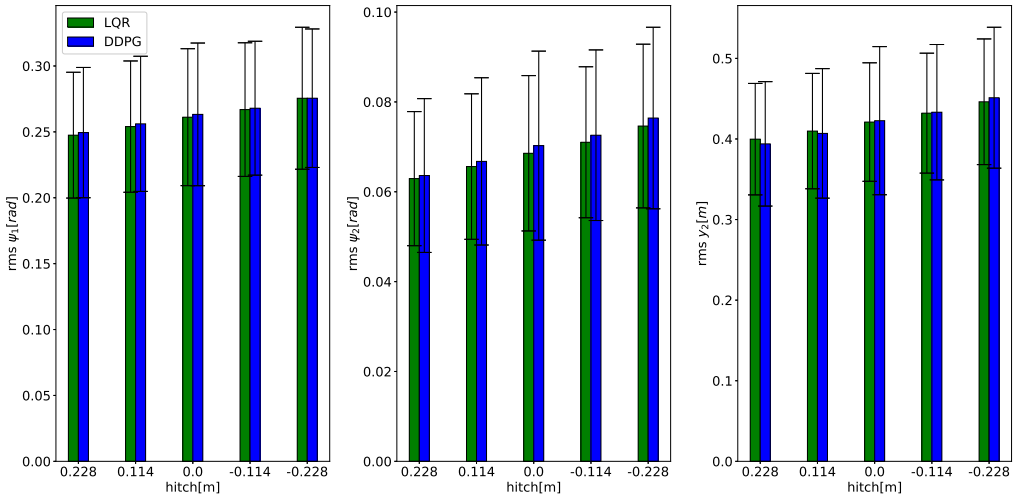


Figure 7.15: The bar charts show the controllers average maximum errors as the hitch length is changed.

Within the goal region, the controllers performance are evaluated for its placement in Figure 7.16. The DDPG policy for angle placement is superior than that of the LQR. The distance to the goal performance of the two is not clear one way or the other, but that is probably due to the fact there is a geometric optimality for moving the trailer position.

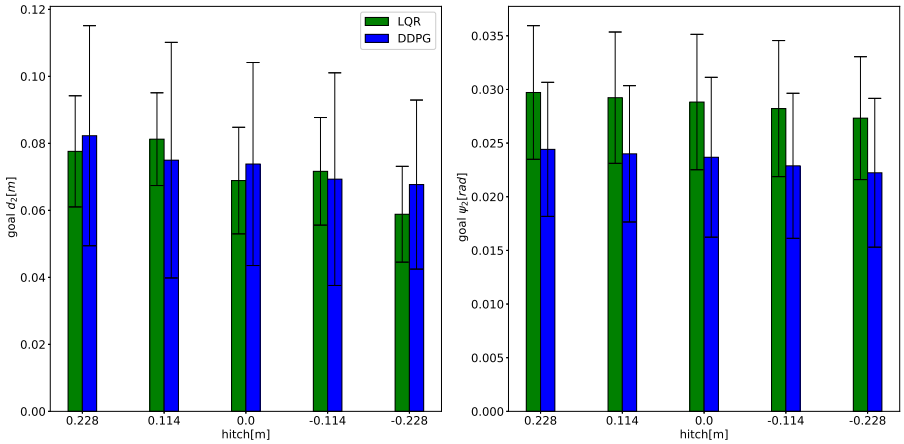


Figure 7.16: The bar charts show the controllers performance at the goal region when the hitch length is varied.

An interesting note is that it appears that as the hitch length is moved closer to the center of gravity of the tractor, the DDPG placement improves for the straight to the goal. This is probably due to the DDPG policy being more precise in a straight and more conservative in a curve because it does not have to worry about jackknifing. This appears to be opposite of the results from the rms error bar charts, however, the rms error may hide information with curves along the path. When looking at the goal region performance, the shorter leverage must allow for more precise movements in a straight.

7.4 Varying Velocity

Typically autonomous vehicles use velocity as another action, but the controller for the work in this thesis focused on the lateral position control instead of longitudinal. The kinematic model assumes constant velocity and thus, the LQR designed is for a linear time invariant problem. Considering the tractor-trailer moves so slow in reverse, it was appropriate to use a kinematic model. Initial studies found that the LQR could handle almost any velocity that was thrown at it. Further investigating found that the LQR gains hardly change when designing the controller for different assumed velocities. Faster speeds should use a kinetic model. It is to be expected that there may not be that much change in the performance summary when looking at Table 7.7 and plots ahead.

Table 7.7: Summary of the LQR and DDPG on 100 new random tracks, varying the velocity.

Condition	-	-2.906 $\frac{m}{s}$	-2.459 $\frac{m}{s}$	-2.012 $\frac{m}{s}$	-1.564 $\frac{m}{s}$	-1.118 $\frac{m}{s}$
Goal	LQR	79	78	79	79	79
	DDPG	86	87	86	87	88
Fin	LQR	79	79	79	79	80
	DDPG	99	99	99	99	98
Jackknife	LQR	21	21	21	21	20
	DDPG	0	0	0	0	0
Large Distance	LQR	0	0	0	0	0
	DDPG	1	1	1	1	1
Large Angle	LQR	0	0	0	0	0
	DDPG	0	0	0	0	0
Out of Bounds	LQR	0	0	0	0	0
	DDPG	0	0	0	0	1

The bar charts in Figures 7.17 and 7.18 for varying the velocity indicate the fact that the controllers are not very sensitive to the velocity change due to the kinematic model.

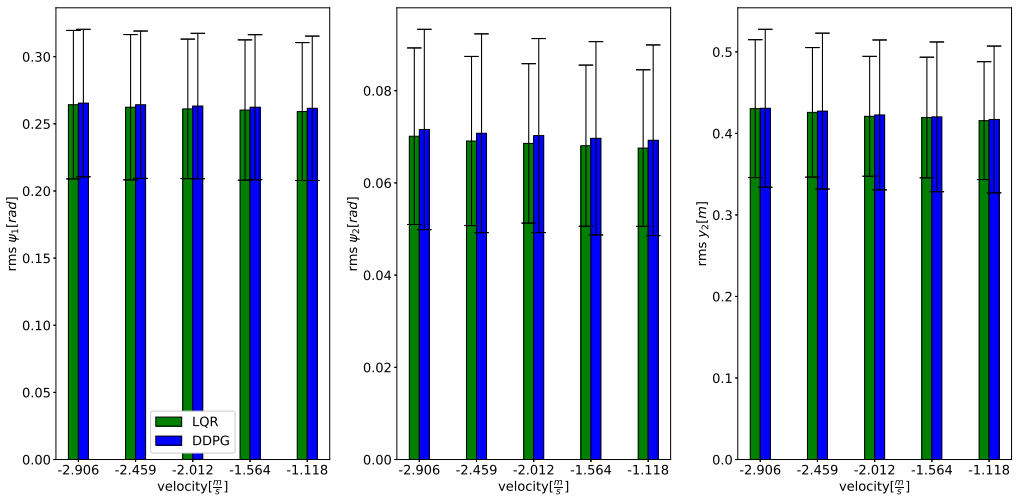


Figure 7.17: The bar charts show the rms performance as the velocity is varied.

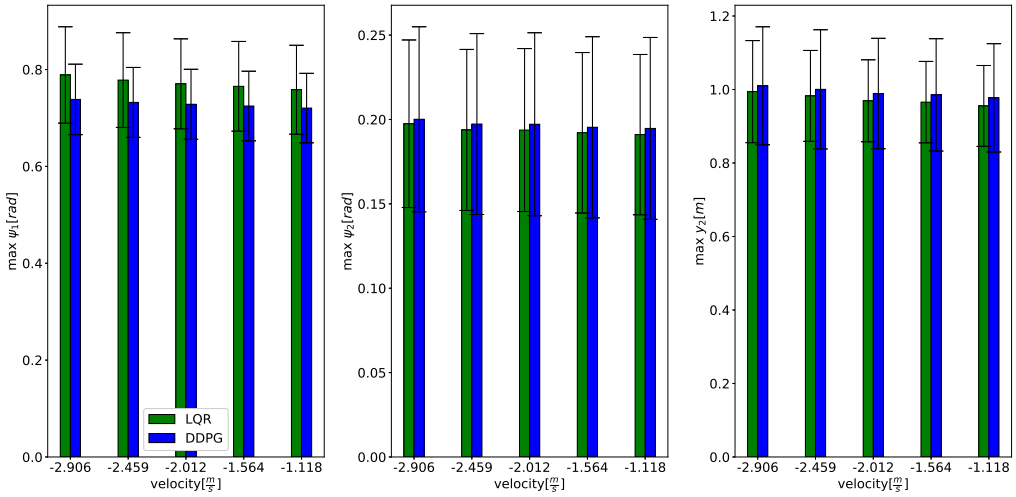


Figure 7.18: The bar charts show the average max error as the velocity is varied.

The data is more clear when looking at the radar graphs of percent error change from nominal in Figure 7.19. The maximum percent error is slightly over 2%, but there is a slight change for the small velocity change. The slower the velocity, the more precise the controllers can be. Surprisingly the LQR performance degrades the most at the fastest speed, but it is still only slightly greater than 2%.

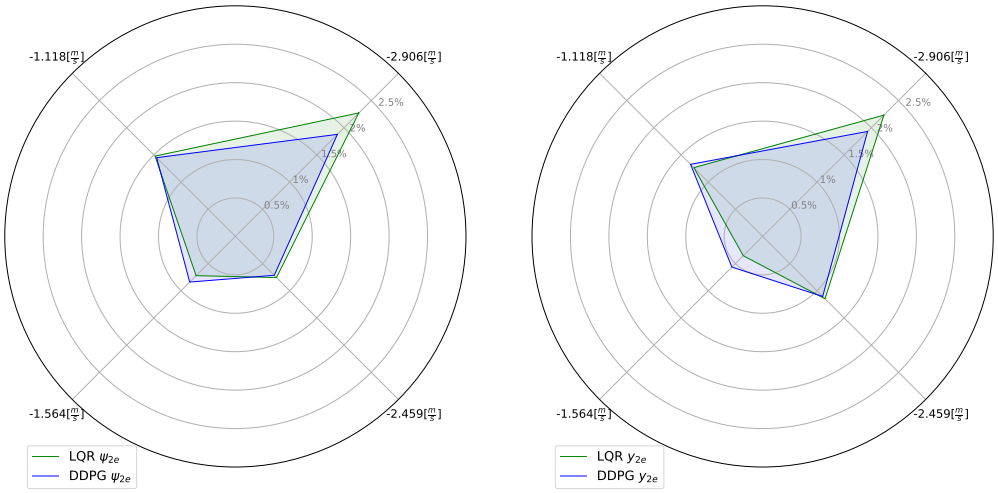


Figure 7.19: The radar graphs show the percent error from both the nominal LQR and nominal DDPG when varying velocity.

When observing the performance at the goal region in Figure 7.20, it is evident that both controllers are more precise when the velocity is slower. This makes sense because they can take more actions to correct up until the loading dock. Once again, the angle orientation seems to outperform with DDPG and the distance to the goal does better with LQR.

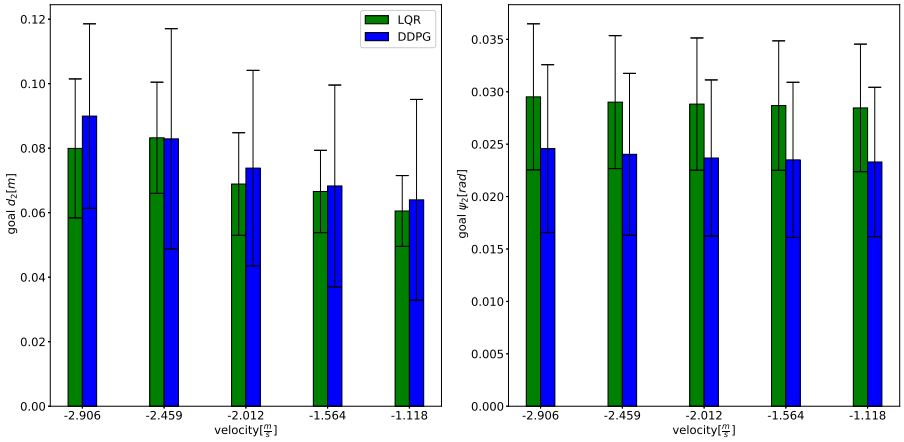


Figure 7.20: The bar charts show the goal region performance as the velocity is varied.

7.5 Varying Sensor Noise

So far this thesis has been assuming perfect sensor representations in the sense that the exact information is available at every instance. Real world applications may have noise or information being obtained at different rates. To bring this work one step closer to reality, sensor noise is modeled

by adding a Gaussian noise with a mean, $\mu = 0.0$, and various standard deviations, σ. Increasing the standard deviation will effectively increase the variance of noise added to the signal.

The signal noise is added to the position sensors of the trailer rear axle. It is added to both x and y because both values are used to calculate the lateral error using rotation matrices. The noise is clipped to have a maximum value of $[-0.3, 0.3]$ for the larger standard deviations. This clipping range is different for the trailer angle orientation. This is due to the inherent scale of realistic and expected values for trailer angles.

```
    if self.sn:
        self.x2_e[i] += np.clip(random.gauss(0.0, self.sn), -0.3, 0.3)
        self.y2_e[i] += np.clip(random.gauss(0.0, self.sn), -0.3, 0.3)
        self.psi_2_e[i] += np.clip(random.gauss(0.0, self.sn), -0.17, 0.17)
```

The controllers see the states with noise, however, the simulation uses the correct value. The idea is to simulate the autonomous vehicle's motion properly, but feed the controllers a noisy signal. Imagine being under the influence and trying to plug one's charger into their phone. The sensor is one's vision, but this does not change that physical location of the phone and the cable. To demonstrate that noise has been added properly, the path from Figure 7.2 is plotted in Figure 7.21.

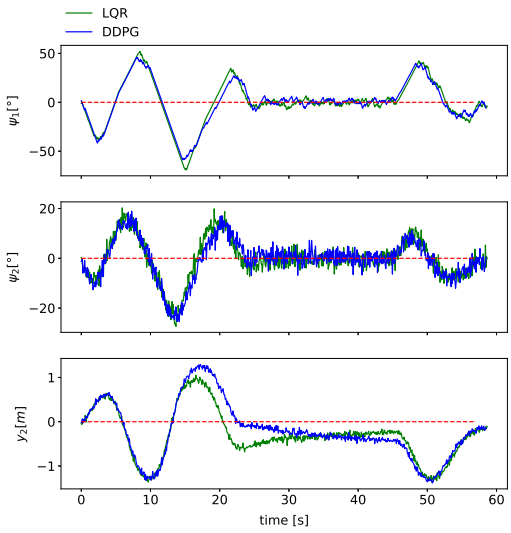


Figure 7.21: Considering the same single path as before, sensor noise with a $\sigma = 0.4$ was added.

The general trend is similar as before, however, the error signals are not as clean. In fact, the amplitudes of the trend are larger in some areas due to controller performance degradation. Both controllers reach the loading dock with $\sigma = 0.4$, however, the LQR does not with $\sigma = 0.5$. Figure

7.22 demonstrates that the LQR run ends early due the angle error becoming too large from the path. It doesn't make sense to continue running the LQR in this situation because it is beyond the point of no return. The summary of the results are found in Table 7.8

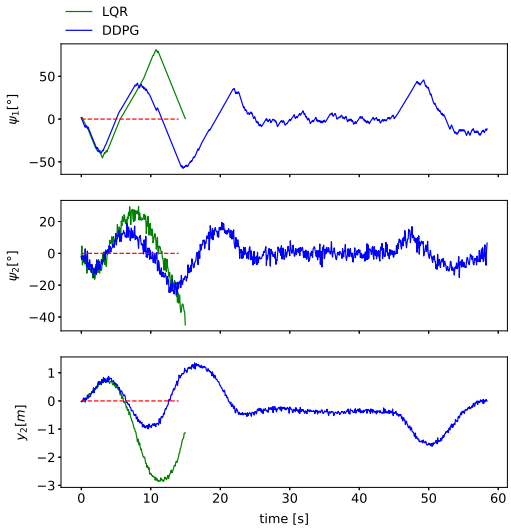


Figure 7.22: Sensor noise with a $\sigma = 05$ was added and the LQR actually incurred too large of an angle from the path. This terminated the run.

Table 7.8: Summary of the LQR and DDPG on 100 new random tracks, varying the sensor noise. The signal noise is modeled as Gaussian noise with a mean of zero and the standard deviation is varied.

Condition	-	$\sigma = 0$	$\sigma = 0.3$	$\sigma = 0.4$	$\sigma = 0.5$	$\sigma = 0.6$
Goal	LQR	79	73	68	53	40
	DDPG	86	81	79	70	48
Fin	LQR	79	74	71	67	54
	DDPG	99	97	95	94	85
Jackknife	LQR	21	25	23	27	26
	DDPG	0	0	0	0	0
Large Distance	LQR	0	0	0	0	0
	DDPG	1	1	0	0	0
Large Angle	LQR	0	1	6	6	20
	DDPG	0	2	5	6	15
Out of Bounds	LQR	0	0	0	0	0
	DDPG	0	0	0	0	0

The DDPG policy learned to never jackknife even in the face of sensor noise. The DDPG policy reached the loading dock more times than the LQR controller. Considering the runs where both controllers reach the goal, the bar graphs are plotted in Figure 7.23 to evaluate the rms errors. As the standard deviation of the Gaussian noise is increased, so are the rms errors for both controllers. The plots make it appear that sometimes the DDPG policy does better and sometimes the LQR controller handles better.

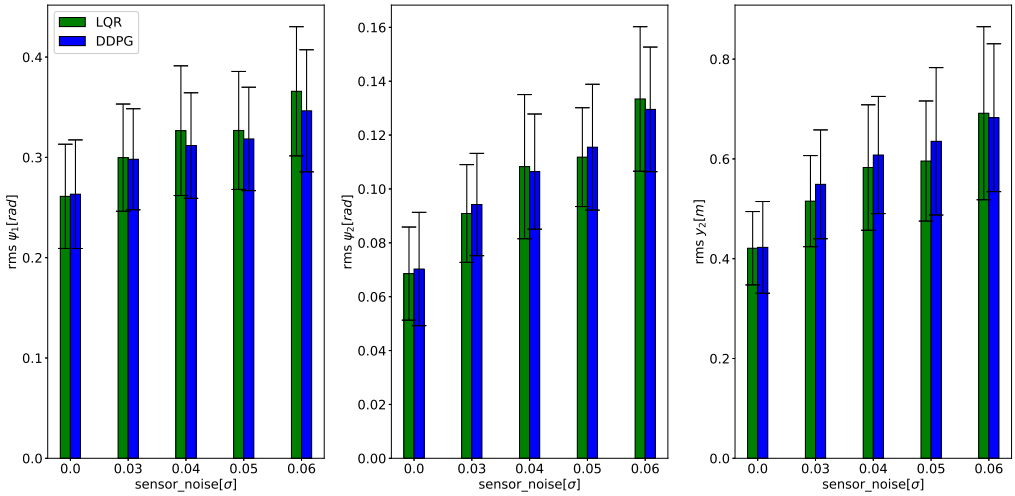


Figure 7.23: The bar charts show the rms performance as the sensor noise is varied.

To get a clearer picture, the radar graphs representing the percent error from nominal in Figure 7.24 are useful for evaluating the effect of sensor noise on the controllers. Apparently, the DDPG policy has less surface area with respect to the angle orientation and the LQR controller has less surface area for the lateral error. The DDPG policy, however, seems to perform the best with the largest amount of variation of sensor noise.

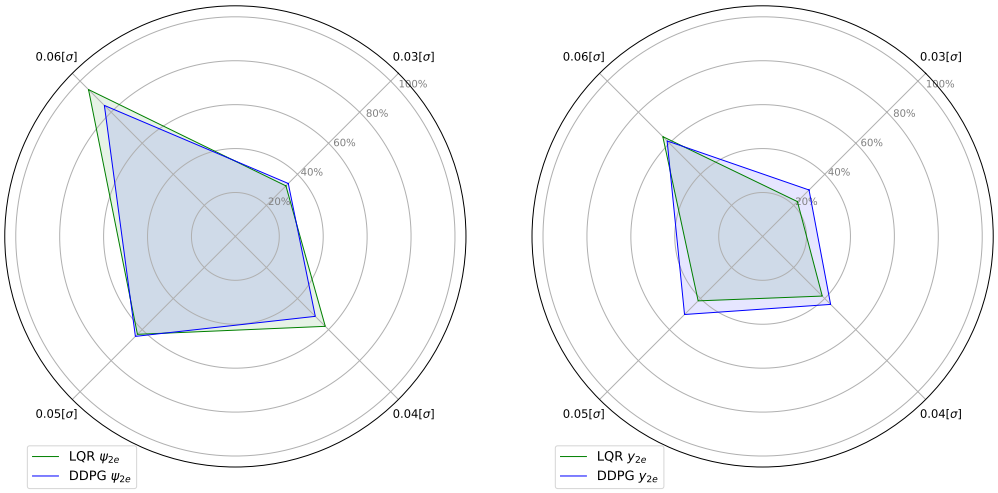


Figure 7.24: The radar graphs show the percent error from both the nominal LQR and nominal DDPG when varying sensor noise.

When interpreting the average maximum errors in the bar plot of Figure 7.25, the results are cloudy because sometimes DDPG has a higher maximum, but other times the LQR has the higher of the two. However when interpreting the performance in the goal region in Figure 7.26, the DDPG policy tended to do better.

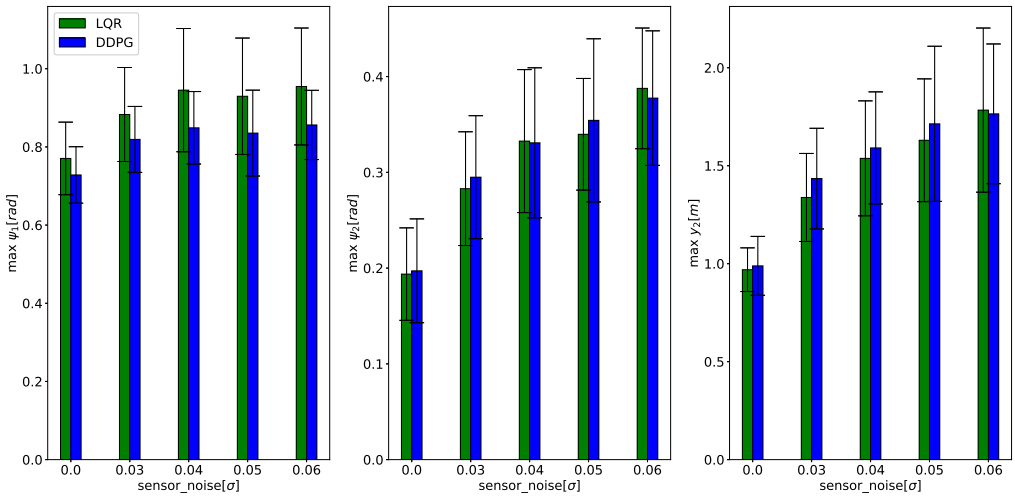


Figure 7.25: The bar charts show the average maximum errors as the sensor noise is varied.

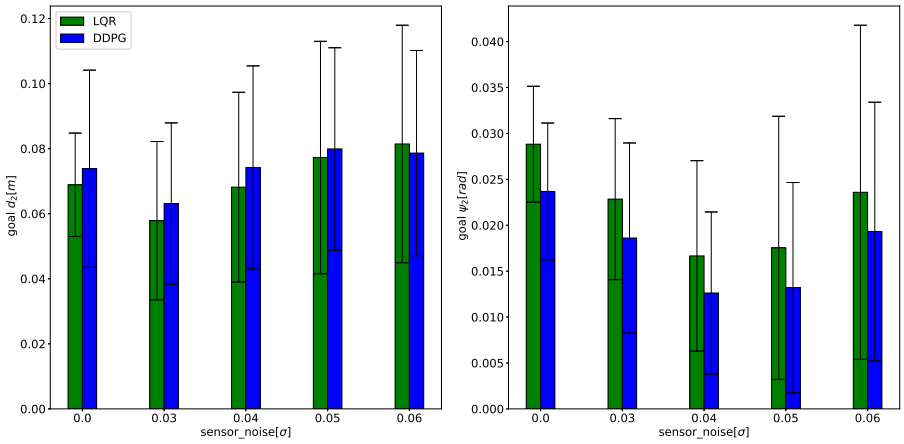


Figure 7.26: The bar charts show the performance in the goal region as the sensor noise is varied.

7.6 Varying Control Frequency

Thus far, the controller sampling timestep has been assumed to be $80ms$–which is exactly the timestep is in the simulator. In actuality, the control frequency may differ from the simulation time. For the purpose of training, it made sense to run with the largest timestep as possible that still depicted the results accurately as if it were running with a smaller timestep. The faster the simulation is, the faster training will be. In real life, however, many factors dictate the control frequency. It may be due to the logging rate of sensors or the computation time of other systems. A simple solution might be to run the simulator at the same control frequency that one expects in real life, but this would slow down getting results. If anything, one would want to train at a slower control frequency than is expected to ensure good results. This section will use control frequency and controller sampling timestep interchangeably, but care will begin when mentioning that it is increased or decreased.

The implementation for decoupling the control frequency from the simulation time is incorporated in the testing script. Basically if the modulo of the simulation time and the control frequency equals zero, then a new control action is computed. This is true for either the DDPG policy or LQR controller. The previous action is then stored for later use. Otherwise, the remaining timesteps are calculated at an extremely slow timestep of $dt = 0.001$. The previous action is used over and over again until it is the correct moment for a new controller computation.

```
    if (env.sim_i-1) % int(PARAMS[param_idx] / env.dt) == 0:
        a = np.clip(K.dot(s), env.action_space.low, env.action_space.high)
        a_last = a.copy()
    else:
        a = a_last.copy()
```

Using the same track as before, the effect of a slower control frequency can be observed in Figure 7.27. The control loop is set to 400ms and it is apparent how both controllers are much later in responding.

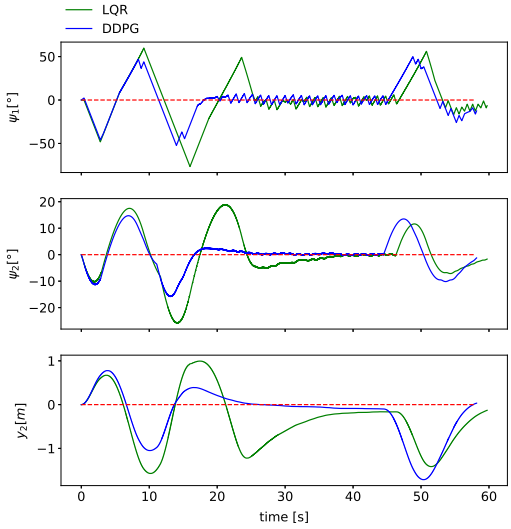


Figure 7.27: Considering the same single path as before, the control frequency was reduced to make an action every 400ms.

Increasing the interval between new actions further from 400ms results in both controllers ending the run early. The LQR jackknifed and the DDPG policy had the angle orientation error of the trailer become too large from the path. The summary of the results are found in Table 7.9.

Table 7.9: Summary of the LQR and DDPG on 100 new random tracks, varying the control frequency.

Condition	-	1ms	10ms	80ms	400ms	600ms
Goal	LQR	79	79	79	69	37
	DDPG	88	88	87	82	60
Fin	LQR	80	80	79	70	44
	DDPG	99	99	99	95	77
Jackknife	LQR	20	20	21	28	49
	DDPG	0	0	0	1	10
Large Distance	LQR	0	0	0	0	2
	DDPG	1	1	1	2	9
Large Angle	LQR	0	0	0	2	5
	DDPG	0	0	0	2	4
Out of Bounds	LQR	0	0	0	0	0
	DDPG	0	0	0	0	0

As expected, increasing the controller frequency was hardly beneficial for both controllers. What was detrimental, however, was decreasing the controller frequency. The DDPG policy still reached the goal more than the LQR, but was actually prone to jackknifing now. Figure 7.28 displays the rms errors as the control frequency is changed. As it is increased, the results are to be expected from the previous plots. As the control frequency is decreased, the DDPG policy almost always did better except for the lateral error at the smallest control frequency.

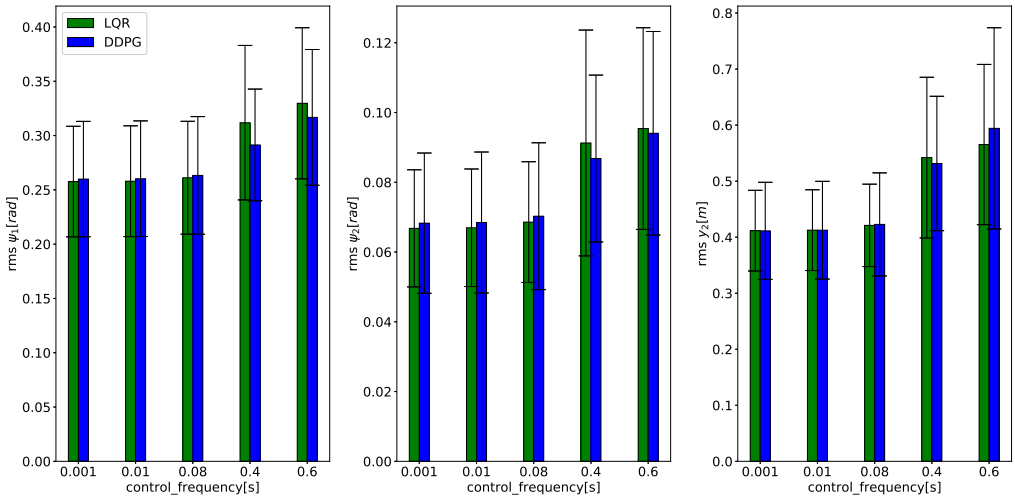


Figure 7.28: The bar charts show the rms errors as the control frequency is varied.

The results are similar for when looking at the radar graphs in Figure 7.29, i.e. the results are pretty one sided with respect to decreasing the control frequency. The percent error from nominal is much better for DDPG with the angle orientation of the trailer, but almost always better for the lateral error.

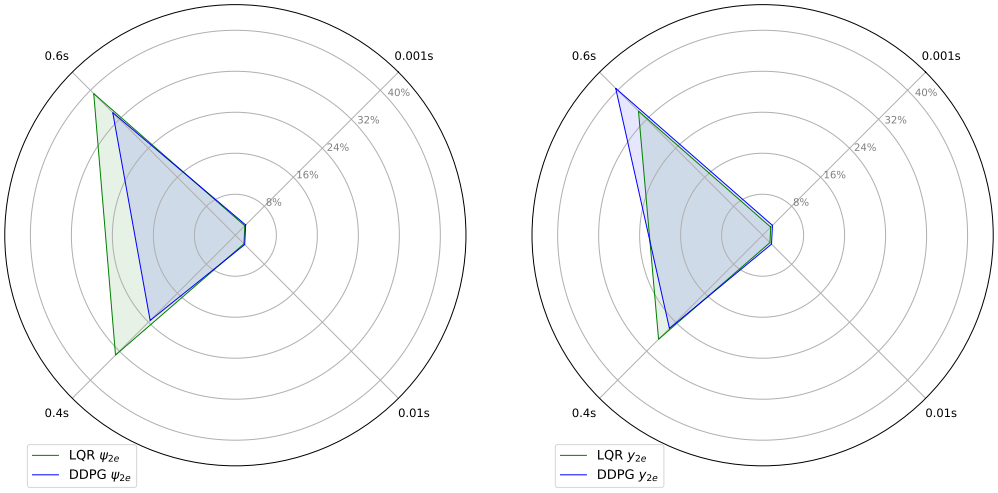


Figure 7.29: The radar graphs show the percent error from both the nominal LQR and nominal DDPG when varying control frequency.

Figure 7.30 resembles the maximum errors as the control frequency is altered. The trends for the maximums look identical between the rms and max errors. When referring to the performance in the goal region in Figure 7.31, the function approximator with the DDPG policy outperforms the LQR controller when the controller frequency is increased.

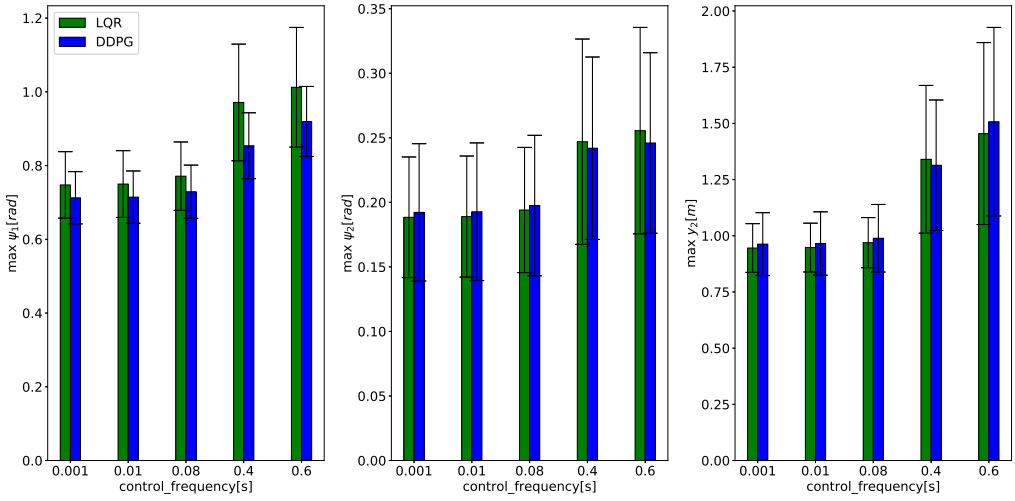


Figure 7.30: The bar charts show the average maximum errors as the control frequency is varied.

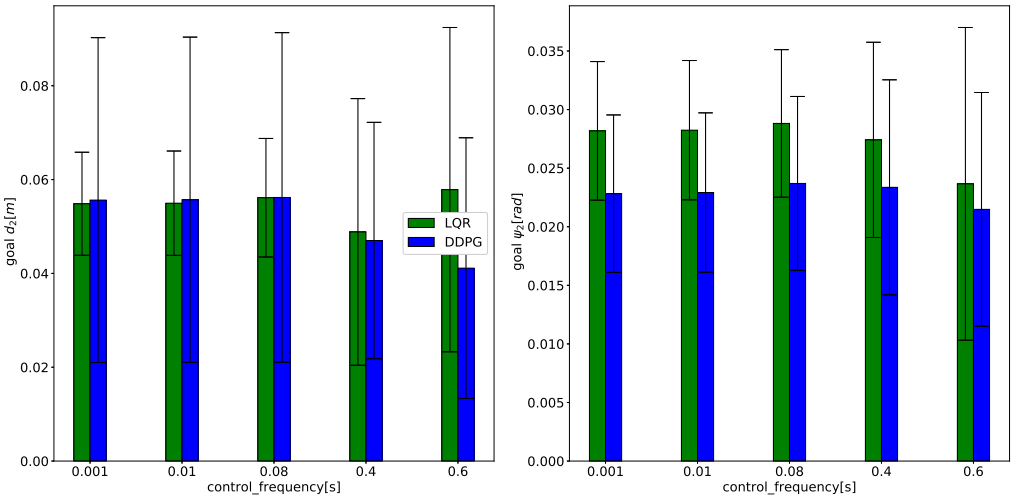


Figure 7.31: The bar charts show the performance in the goal region as the control frequency is varied.

7.7 Summary

Varying the wheelbase, hitch length, and velocity of the tractor-trailer exercised the limits of both the LQR controller and the DDPG policy because they were designed with a specific model in mind. This thesis studied the ability of both controllers being able to generalize to situations where the domain may change. Introducing sensor noise and altering the control frequency evaluated both controllers for robustness in situations closer to reality. All of the studies judged the controllers for robust control. The DDPG policy generalized to more paths and models in the sense that it never jackknifed, but the LQR tended to have better precision when it actually reached the goal.

Chapter 8

SUMMARY AND CONCLUSIONS

Steering a tractor-trailer in reverse autonomously to a loading dock requires a controller with high fidelity due to the variation of wheelbase, hitch length, and speeds. Typically modern controller gains, such as the LQR, are designed with a specific model in mind. Reinforcement learning uses similar vernacular such as states and actions to solve the same problem of optimal control, except the policy is actually learned through experience. A neural network is used as the tool for creating the function approximator representing the policy learned using the reinforcement learning algorithm called DDPG. Both controllers are used in a myriad of scenarios such as the model changing, inclusion of sensor noise, and altering the control frequency.

8.1 Contributions

Major contributions of this thesis include an objective comparison to the benchmark of the Linear Quadratic Regulator for the problem of backing up a tractor-trailer. Literature such as Nguyen [17] and Gatti [21] investigate backing up a truck and trailer using discrete actions where the reinforcement learning agent can steer fully left, fully right, or keep straight. They deserve high praise for their initial work and this thesis takes the problem one step closer to reality due to the advancements of reinforcement learning algorithms. The DDPG algorithm is used for deterministic, continuous actions.

Furthermore, the work in this thesis is novel because it is concerned with the path taken to changing loading dock locations. A planner is used to spawn paths that the controllers use to generate errors from the path as states. In other words, feedback is used to create two closed loop controllers that are evaluated against each other. Previous literature treated the loading dock always at the location $(0, 0)$, which may be not be considered as a general control scheme. Their work randomizes the initial starting positions and evaluates the reinforcement learning agent in backing up the trailer to the loading dock without jackknifing. In actuality, autonomous vehicles will plan a reference path to avoid obstacles; it matters what route the combination vehicle takes to the loading dock. This thesis addresses the autonomy gap between academia and modern solutions. A planner is used to generate a path from random starting and ending poses, localization determines where the current vehicle is with respect to the desired path, and feedback is used for optimal control to the loading dock.

8.2 Discussion

Three primary methods were used to evaluate the controllers for generalization or robust control: 1) Bar graphs with rms averages 2) Radar graphs with percent error from nominal and 3) the overall number of times the tractor-trailer reached the goal or jackknifed. Parameters or configuration variables were increased and decreased from the nominal.

The bar graphs contain a plethora of information, but truly shows the performances in each of the states for both controllers as each of the parameters are changed. The trends are not always clear or consistent, but give the reader the sense that changing parameters really do affect performances. The LQR controller tended to place the trailer on the path more precise than the DDPG policy in regards to altering the wheelbase, hitch length, and velocity. It was theorized that the function approximator in the DDPG policy would be superior than the LQR when increasing the sensor noise, but the bar charts potentially suggest otherwise. However, the bar charts do suggest that the DDPG policy was more inclined to prevail when the control frequency is slowed down.

Radar graphs are used to augment the data from the bar charts to provide a clearer picture. The rms errors for each change in configuration variable are instead compared to the nominal case, suggesting that a smaller percent error from nominal means the controller generalized better. The controllers were most sensitive to the model change of the trailer wheelbase, but least sensitive to velocity. Out of all the configuration changes, both controllers were most sensitive to sensor noise. For each of the model changes, the surface area of the LQR controller percent errors in the radar graphs tended to be smaller suggesting the LQR was more robust to parameter changes. On the other hand, the DDPG policy surprisingly had performed closer to the nominal performance when introduced to sensor noise and decreasing the controller frequency.

The third method of evaluating the performances were counting the number of times the controllers reached terminal conditions, good or bad. A major success of this work was learning a policy where the agent would never jackknife. What this means is that reinforcement learning allowed for creating a controller that could impose non-linear constraints by wrapping it into the reward function. Through experiencing good and bad situations, the learned policy knew it should not put itself into situations where it could result in jackknifing. Reinforcement learning is useful for sequential problems where it may sacrifice a higher short term reward for anticipating the longer, larger reward. In this situation, it appears that the agent may sacrifice precision along the path to ensure it can reach the goal which rewarded +100 for completion.

On the contrary, the LQR gains were designed with a model of the tractor-trailer from an equilibrium point set exactly on the path. The gains are the most valid for small perturbations from the equilibrium point. As it turns out, the precision and generalization of the LQR quantitatively exceeded the DDPG policy. However, the DDPG controller could further benefit from training on multiple trailers and tracks to learn a more general optimal policy using the function approximator. If at the end of the day the objective is to get the trailer to the loading dock without jackknifing, one should use the DDPG policy when not re-planning every few seconds. The reason for this is the ability for reinforcement learning to encode a non-linear constraint, which is not something the LQR can take into account. It is worth mentioning that the computation associated with a NN is much faster than it would be to run optimizations in the case of imposing constraints with MPC.

An interesting phenomena was found where as the wheelbase of the trailer is decreased, both controller performances generally improved. This finding was not expected because truck drivers will say that reversing a longer trailer is easier than a short one because a change in steering will equate to a smaller change in the trailer. Additionally, as the hitch length was pushed behind the rear axle, the system was easier to control for both methods. This may be due to the fact that the controllers are not imposing constraints on steering rate. In order to be more realistic, the actuators for the steering angle should not be able to move as fast as it does. If this controller were to be implemented in real life, the control effort would need to be minimized.

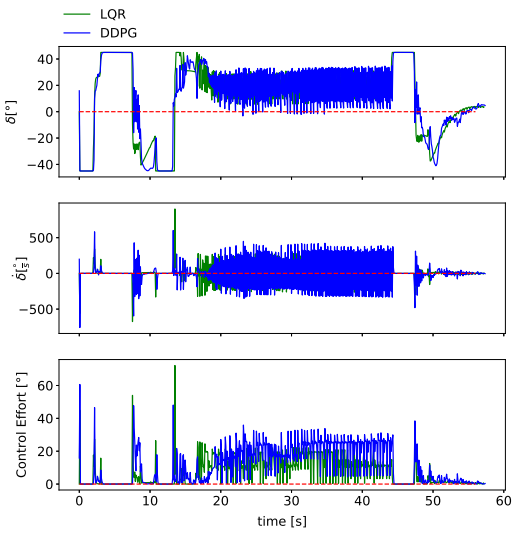


Figure 8.1: For the same single path as before, the control effort is displayed along with the unrealistic steering rate for the nominal case.

This can be accomplished by penalizing for high steering rates; in other words, this means that a cost should be added to the reward for requesting undesired steering rates. As sensor noise is increased, one would expect the actuator commands to be more noisy too. This is the case in Figure 8.2 where the sensor noise has a standard deviation of 0.04.

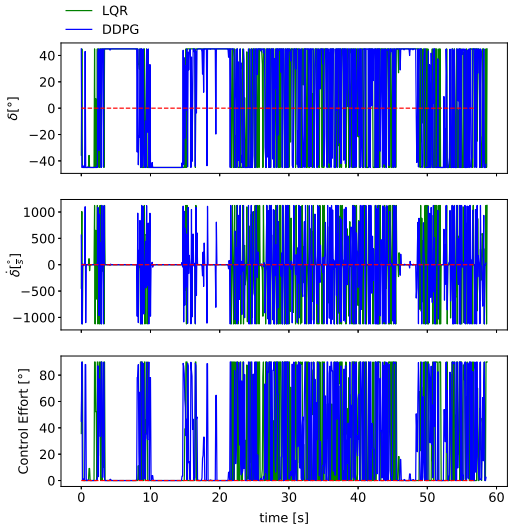


Figure 8.2: For the same single path as before, the control effort is displayed along with the unrealistic steering rate for the scenario with sensor noise.

As the control frequency is decreased, one would expect the steering commands to be much more discrete with a high control effort. This is apparent in Figure 8.3 because commands are much more on-off, bouncing off the saturations of the steering angle between $[-45°, 45°]$.

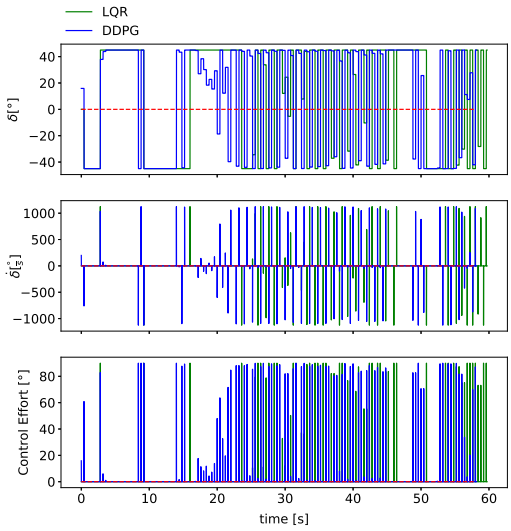


Figure 8.3: For the same single path as before, the control effort is displayed along with the unrealistic steering rate for the scenario with a control frequency of 400ms.

8.3 Future Work

As is custom, it is time to conclude this work with future improvements and items to try. At risk of sentimentality, this thesis is a snapshot in time and even though it is time to take a knee, future papers can be generated by further investigating. The first item that is suggested is to continue training with the previous weights and biases on various trailer parameters. The function approximator was trained on the same tractor-trailer on multiple paths, which was an impressive feat. If one desires a general policy for the many trailers out in existence, the next step would be to transfer learn with domain randomization. What this means is to continue training with minor changes to the vehicle parameters such as a 10% increase and decrease in trailer wheelbase.

One should expect the episodic reward to initially decrease in training, but the reinforcement learning algorithm should begin to find the weights and biases that dictate the mapping of states to actions for optimal control of different trailer types. After the reward increases again, it is recommended to stop training and increase the variance of domain randomization. Training could be continued in a process that would continually increase the lesson difficulty in a curriculum form. This can be done for as many trailer parameters as desired.

This thesis succeeded in learning a policy that could handle a substantial amount of paths it never trained on after training on 29 tracks. Training on completely random tracks was desired and attempted, however, it was discovered that the Dubins Path planner may not have always created

a path that was ideal for the LQR. It was observed that some of the paths created by the planner caused the LQR to jackknife–as is evident with the LQR, jackknifing a fair amount, on 100 new random tracks. The reason this is relevant is because the DDPG policy was trained using the LQR as an expert policy. The expert policy guided the training by providing good samples without the need for only randomly exploring. The problem is, however, training on a novice policy can produce training noise in the replay buffer. This is where methods such as prioritized experience replay [86] can be beneficial. The most important samples that are beneficial for learning are selected for the batch to train on more frequently.

More importantly, it is believed that a better path planner can be used. Dubins Curves uses a kinematic car in the forward direction and was modified to include a straight at the end. Other planners such as Model Predictive Control use an actual model of the system to run through trajectories and choose the best path using convex optimization. This was not initially chosen because it was expected that the planner performance would hinder as the model parameters were changed, but not the planner as well. The idea for the planner in this thesis was to select a path and evaluate the controller performance over a fixed reference path.

Typically autonomous vehicle planners are computationally expensive, but are necessary for planning a few seconds ahead. Re-planning allows for corrections as the controller error grows too large. Modern controllers are typically valid for small perturbations, so Model Predictive Control actually recalculates the gains and picks the next action after the optimal trajectory is chosen from a trajectory bundle. The optimal trajectory may be selected such that it would not result in a jackknife event. In order to successfully train on completely random paths, it makes sense to incorporate a planner that can look a few seconds ahead.

The issue arises then when it comes time to compare the controllers on the same paths as the parameters change. If the planners are continually looking at a finite horizon ahead, the paths may not be deterministic. Perhaps the trajectory can be stored after running with the Model Predictive Controller and is set to being a fixed reference path just as done in this thesis. There may be concern for the path to not be optimal for the reinforcement learning policy. Nonetheless, the more realistic paths it trains on, the more generalizable the policy will be.

Lastly, this thesis assumes that information about the trailer rear axle position is obtainable. Many trailers to date are not equipped with sensors for basic information, let alone position or orientation. This is because the tractor manufacturers do not own the trailers. In other words, the problem might actually be a Partially Observable Markov Decision Process. The modern controls

method for dealing with missing states involve using observers or state estimators. An interesting parallel found in reinforcement learning is to use a Recurrent Neural Network as the function approximator for dealing with missing states [76]. A history of states of maybe three timesteps could be fed to the RNN as a sequence and reinforcement learning can potentially learn an optimal control policy with some missing information. Reinforcement learning solves the optimal control problem just as modern controls, however, uses a function approximator such as a neural network for learning from experience.

BIBLIOGRAPHY

[1] Donald Michie. Experiments on the mechanization of game-learning part i. characterization of the model and its parameters. *The Computer Journal*, 6(3):232–236, 1963.

[2] Sae j1939: J1939 digital annex. Standard, Society of Automotive Engineers, Warrendale, PA, August 2018.

[3] Road vehicles - interchange of digital information on electrical connections between towing and towed vehicles. Standard, International Organization for Standardization, Geneva, CH, April 2015.

[4] Rakshith KUSUMAKAR, Karel KURAL, Abhishek Singh TOMAR, and Ben PYMAN. Autonomous parking for articulated vehicles. 2017.

[5] Tong Wu and John Y Hung. Lateral position control for a tractor-trailer system using steering rate input. In *Industrial Electronics (ISIE), 2017 IEEE 26th International Symposium on*, pages 503–507. IEEE, 2017.

[6] Amy J Rimmer and David Cebon. Theory and practice of reversing control on multiply-articulated vehicles. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 230(7):899–913, 2016.

[7] Romano M DeSantis. Path-tracking for a tractor-trailer-like robot: Communication. *The International Journal of Robotics Research*, 13(6):533–544, 1994.

[8] Hiroshi Kinjo, Bingchen Wang, and Tetsuhiko Yamamoto. Backward movement control of a trailer truck system using neuro-controllers evolved by genetic algorithm. In *Industrial Electronics Society, 2000. IECON 2000. 26th Annual Confjerence of the IEEE*, volume 1, pages 253–258. IEEE, 2000.

[9] *Articulation angle estimation and control for reversing articulated vehicles*. CRC Press, 206.

[10] Michael Hafner and Thomas Pilutti. Control for automated trailer backup. Technical report, SAE Technical Paper, 2017.

[11] Tatsuya Yoshimoto, Kosuke Kaida, Takanori Fukao, Kenji Ishiyama, Tsuyoshi Kamiya, and Noriyuki Murakami. Backward path following control of an articulated vehicle. In *System*

Integration (SII), 2013 IEEE/SICE International Symposium on, pages 48–53. IEEE, 2013.

[12] Hans Pacejka. *Tire and vehicle dynamics*. Elsevier, 2005.

[13] Nikolaj Zimic and Miha Mraz. Decomposition of a complex fuzzy controller for the truck-and-trailer reverse parking problem. *Mathematical and Computer Modelling*, 43(5-6):632–645, 2006.

[14] F Gómez-Bravo, F Cuesta, and A Ollero. Autonomous tractor-trailer back-up manoeuvering based on changing trailer orientation. *IFAC Proceedings Volumes*, 38(1):301–306, 2005.

[15] Jin Cheng, Yong Zhang, and Zhonghua Wang. Curve path tracking control for tractor-trailer mobile robot. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, volume 1, pages 502–506. IEEE, 2011.

[16] M Abroshan, M Taiebat, A Goodarzi, and A Khajepour. Automatic steering control in tractor semi-trailer vehicles for low-speed maneuverability enhancement. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, 231(1):83–102, 2017.

[17] Derrick Huy Nguyen. Applications of neural networks in adaptive control. 1991.

[18] Robert E Jenkins and Ben P Yuhas. A simplified neural network solution through problem decomposition: The case of the truck backer-upper. *IEEE transactions on neural networks*, 4(4):718–720, 1993.

[19] Marc Schoenauer and Edmund Ronald. Neuro-genetic truck backer-upper controller. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 720–723. IEEE, 1994.

[20] ML Ho, PT Chan, AB Rad, and CH Mak. Truck backing up neural network controller optimized by genetic algorithms. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 2, pages 944–950. Citeseer, 2003.

[21] Christopher Gatti. *Design of experiments for reinforcement learning*. Springer, 2014.

[22] M Khorsheed and MA Al-Sulaiman. Truck backer-upper control using adaptive critic learning. *Journal of King Saud University-Computer and Information Sciences*, 10:73–80, 1998.

[23] Hans Vollbrecht. *Hierarchical reinforcement learning in continuous state spaces.* PhD thesis, University of Ulm, Germany, 2003.

[24] Robert O Shelton and James K Peterson. Controlling a truck with an adaptive critic temporal difference cmac design. 1993.

[25] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[26] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 2005.

[27] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. *Reinforcement Learninng and Optimal Control*, volume 1. Athena scientific Belmont, MA, 2019.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[29] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[30] Andreas Bell Martinsen. End-to-end training for path following and control of marine vehicles. Master's thesis, NTNU, 2018.

[31] Sen Wang, Daoyuan Jia, and Xinshuo Weng. Deep reinforcement learning for autonomous driving. *arXiv preprint arXiv:1811.11329*, 2018.

[32] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[33] MFJ Luijten. Lateral dynamic behaviour of articulated commercial vehicles. *Eindhoven University of Technology*, 2010.

[34] Jeff Hecht. Lidar for self-driving cars. *Optics and Photonics News*, 29(1):26–33, 2018.

[35] Sergey Levine. Deep reinforcement learning.
URL: http://rail.eecs.berkeley.edu/deeprlcourse-fa17/index.html, 2017.

[36] William F Milliken, Douglas L Milliken, et al. *Race car vehicle dynamics*, volume 400. Society of Automotive Engineers Warrendale, 1995.

[37] Manoj Karkee. Modeling, identification and analysis of tractor and single axle towed implement system. 2009.

[38] Amro Elhassan. Autonomous driving system for reversing an articulated vehicle, 2015.

[39] Tong Wu. Solutions for tractor-trailer path following at low speed. 2017.

[40] Roland Siegwart, Illah Reza Nourbakhsh, Davide Scaramuzza, and Ronald C Arkin. *Introduction to autonomous mobile robots*. MIT press, 2011.

[41] Niclas Evestedt. *Sampling Based Motion Planning for Heavy Duty Autonomous Vehicles*. PhD thesis, Linköping University Electronic Press, 2016.

[42] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[43] Ali Boyali, Seichi Mita, and Vijay John. A tutorial on autonomous vehicle steering controller design, simulation and implementation. *arXiv preprint arXiv:1803.03758*, 2018.

[44] Nitin R Kapania. *Trajectory Planning and Control for an Autonomous Race Vehicle*. PhD thesis, Stanford University, 2016.

[45] Richard M Murray. Optimization-based control. *California Institute of Technology, CA*, 2009.

[46] Immanuel Baur. Entwurf einer bahnfolgeregelung fuer die querfuehrung eines formula-student rennfahrzeugs, 2018.

[47] Bryan Nagy and Alonzo Kelly. Trajectory generation for car-like robots using cubic curvature polynomials. *Field and Service Robots*, 11, 2001.

[48] Rajesh Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[49] Jarrod M Snider et al. Automatic steering methods for autonomous automobile path tracking. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08*, 2009.

[50] Fengda Lin, Zijian Lin, and Xiaohong Qiu. Lqr controller for car-like robot. In *Control Conference (CCC), 2016 35th Chinese*, pages 2515–2518. IEEE, 2016.

[51] Tomás Carricajo Martin, Marcos E Orchard, and Paul Vallejos Sánchez. Design and simulation of control strategies for trajectory tracking in an autonomous ground vehicle. *IFAC Proceedings Volumes*, 46(24):118–123, 2013.

[52] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

[53] William L Brogan. *Modern control theory*. Pearson education india, 1974.

[54] Joao P Hespanha. Topics in undergraduate control systems design, 2006.

[55] Filipe Marques Barbosa, Lucas Barbosa Marcos, Maira Martins da Silva, Marco Henrique Terra, and Valdir Grassi Jr. Robust path-following control for articulated heavy-duty vehicles. *arXiv preprint arXiv:1808.02189*, 2018.

[56] Oskar Ljungqvist, Daniel Axehill, and Anders Helmersson. Path following control for a reversing general 2-trailer system. In *Decision and Control (CDC), 2016 IEEE 55th Conference on*, pages 2455–2461. IEEE, 2016.

[57] Niclas Evestedt, Oskar Ljungqvist, and Daniel Axehill. Motion planning for a reversing general 2-trailer configuration using closed-loop rrt. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 3690–3697. IEEE, 2016.

[58] Francois Chollet. *Deep learning with python*. Manning Publications Co., 2017.

[59] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[60] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[61] Derrick H Nguyen and Bernard Widrow. Neural networks for self-learning control systems. *IEEE Control systems magazine*, 10(3):18–23, 1990.

[62] Derrick Nguyen and Bernard Widrow. The truck backer-upper: An example of self-learning in neural networks. In *Advanced neural computers*, pages 11–19. Elsevier, 1990.

[63] David Silver. Advanced topics 2015.
URL: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`, 2015.

[64] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.

[65] Christopher J Gatti and Mark J Embrechts. An application of the temporal difference algorithm to the truck backer-upper problem. In *ESANN*, 2014.

[66] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[67] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[68] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[69] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[70] Vehicle Industry News. Fifth-Wheel Travel Trailer/Coaches Length Increase, 2013.

[71] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[72] Henning Olsen. Ema truck tire metrics database. CalSpan Tire.

[73] Ning Zhang, Hong Xiao, and Hermann Winner. A parameter identification method based on time–frequency analysis for time-varying vehicle dynamics. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 230(1):3–17, 2016.

[74] City of Portland Office of Transportation. Designing for truck movements and other large vehicles in portland. `https://www.portlandoregon.gov/transportation/article/357099`, October 2008.

[75] Andrew Walker. Dubins-curves: an open implementation of shortest paths for the forward only car, 2008–.

[76] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

[77] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*, 2018.

[78] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[79] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. 2018.

[80] Mark Hammond. Deep reinforcement learning models: Tips & tricks for writing reward functions. `https://www.bons.ai/blog/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions`, November 2017.

[81] Aditya Gudimella, Ross Story, Matineh Shaker, Ruofan Kong, Matthew Brown, Victor Shnayder, and Marcos Campos. Deep reinforcement learning for dexterous manipulation with concept networks. *arXiv preprint arXiv:1709.06977*, 2017.

[82] Maximilian Jaritz, Raoul De Charette, Marin Toromanoff, Etienne Perot, and Fawzi Nashashibi. End-to-end race driving with deep reinforcement learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2070–2075. IEEE, 2018.

[83] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. *arXiv preprint arXiv:1812.02900*, 2018.

[84] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (TOG)*, 37(4):143, 2018.

[85] Alex Irpan. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[86] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[87] Plamen Petrov. Nonlinear backward tracking control of an articulated mobile robot with off-axle hitching. In *Proc. WSEAS Int. Conf. ISPRA*, pages 269–273, 2010.

[88] Karl Popp and Werner Schiehlen. *Ground vehicle dynamics*. Springer Science & Business Media, 2010.

[89] Camillo J Taylor, Jana Košecká, Robert Blasi, and Jitendra Malik. A comparative study of vision-based lateral control strategies for autonomous highway driving. *The International Journal of Robotics Research*, 18(5):442–453, 1999.

[90] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[91] LK Chen and YA Shieh. Jackknife prevention for articulated vehicles using model reference adaptive control. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 225(1):28–42, 2011.

[92] Petros A Ioannou and Jing Sun. *Robust adaptive control*. Courier Corporation, 2012.

[93] Eduardo Fernandez-Camacho and Carlos Bordons-Alba. *Model predictive control in the process industry*. Springer, 1995.

[94] Luigi Del Re, Frank Allgöwer, Luigi Glielmo, Carlos Guardiola, and Ilya Kolmanovsky. *Automotive model predictive control: models, methods and applications*, volume 402. Springer, 2010.

APPENDICES

Appendix A
KINETIC MODEL

A.1 Discussion of Lagrangian Mechanics

One can use free body diagrams to solve for the differential equations representing a tractor-trailer, however, a popularized method of using Lagrange's Equations of Motion can be found in Luijten [33] and actually Pacejka [12]. Using Lagrangian Mechanics essentially uses energy to derive the Equations of Motion (EOM); it is an alternative to Newtonian Mechanics. Dealing with scalar values instead of vectors for coupled mass systems can clean things up. Lagrangian Mechanics, however, has a requirement that the system be holonomic–otherwise one can produce erroneous EOMs.

Lagrangian Mechanics for Holonomicity

Euler-Lagrange equations:

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{q}_i} - \frac{\partial T}{\partial q_i} + \frac{\partial U}{\partial q_i} = 0 \tag{A.1}$$

Lagrange equations with Generalized Forces:

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{q}_i} - \frac{\partial T}{\partial q_i} + \frac{\partial U}{\partial q_i} = Q_i \tag{A.2}$$

Lagrangian Mechanics for Non-Holonomicity

Lagrange-d'Alembert Equations:

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{q}_i} - \frac{\partial T}{\partial q_i} + \frac{\partial U}{\partial q_i} = Q_i + \sum \lambda_i a_i \tag{A.3}$$

The term with the Lagrange mMltiplier on the right hand side of the equation can be thought of as constraint forces or reaction forces. These constraint forces do no work on the system and thus still satisfy the original Lagrange equation. This formulation is also known as the Lagrange-d'Alembert equations. In fact, if one applied this version to a holonomic system, the Lagrange Multiplier would cancel out and give the correct EOM.

When one thinks about car-like mobile robots, the term non-holonomic arises due to the rolling without slip assumption. The standard method to handle non-holonomic systems using Lagrangian Mechanics is to include Lagrange Multipliers, but Luijten and Pacejka didn't appear to do so.

The formal definitions are as follows:

- Holonomic: $f(q_1, q_2, ...q_n, t) = 0$ This constraint can be used to reduce the number of degrees of freedom (DOF) in a system. The constraints can be integrable to this form, such as velocity to constrain position.
- Non-holonomic: $f(q_1, q_2, ...q_n, \dot{q_1}, \dot{q_2}, ...\dot{q_n}, t) = 0$ This constraint cannot be used to reduce the number of degrees of freedom in the system. The velocity constraints cannot be integrated to constrain position.

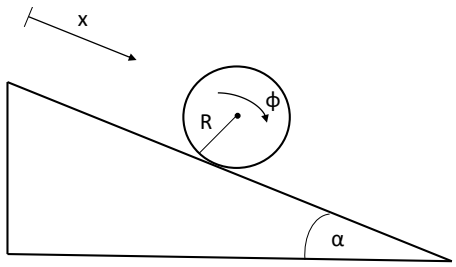


Figure A.1: The 2D disk rolling down a hill is actually holonomic.

The 2D disk in Figure A.1 has the rolling without slip assumption, which looks like $\dot{x} = R\dot{\phi}$.

$$\begin{aligned} f(q_1, q_2, ...q_n, t) &= 0 \\ R\dot{\phi} - \dot{x} &= 0 \\ R\phi - x + C &= 0 \quad \checkmark \end{aligned} \tag{A.4}$$

If the integration constant is zero, it appears the position variables can be constrained. The problem has 1 DOF because the rotation, ϕ can be written in terms of x. Thus, the 2D rolling disk problem has holonomic constraints.

Now imagine a 3D rolling disk down a hill, but it can change its yaw ψ. The rolling without slip assumption still exists, but now looks like the following:

$$\begin{aligned} v &= R\dot{\phi} \\ \dot{x} &= v\cos\psi \\ \dot{y} &= v\sin\psi \end{aligned} \tag{A.5}$$

Let's ignore $\dot{y}$ to simplify the discussion. These constraints are still non-integrable because they cannot be used to relate x with ϕ and ψ. One can integrate $\dot{x}$ and $\dot{\phi}$, but not ψ. This problem has 2 DOF and is thus non-holonomic.

$$\begin{aligned} f(q_1, q_2, ...q_n, \dot{q_1}, \dot{q_2}, ...\dot{q_n}, t) &= 0 \\ R\dot{\phi}\cos\psi - \dot{x} &= 0 \quad \square \end{aligned} \tag{A.6}$$

The following section will discuss how to model a kinetic tractor-trailer using the Lagrangian Mechanics, treating the system as a holonomic one for small angles.

A.2 Tractor Trailer Kinetic Model

The following derivation is heavily inspired by Luijten's and Pacejka's work, but includes the trailer angle orientation as a state instead of the hitch angle. The reason for this is because it made it simpler to create a controller similar to the kinematic model. The system can similarly be thought of as modeling a double pendulum, except with forces being generated on the tires on the axles. A diagram of the system focused on lateral motion is shown in figure A.2.

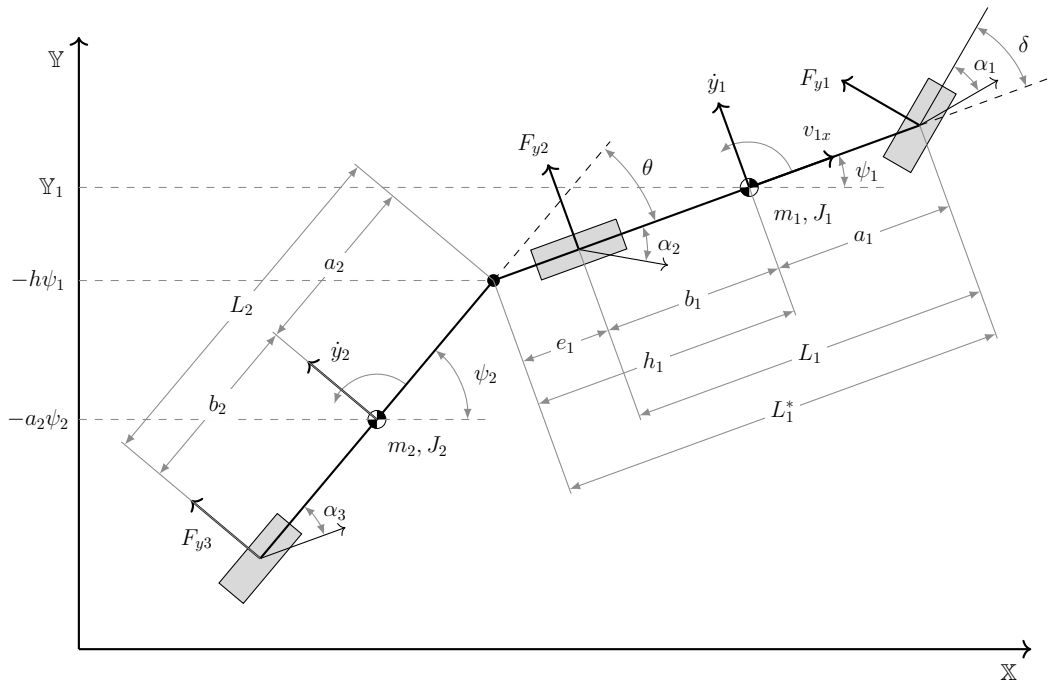


Figure A.2: The kinetic model includes masses, inertia, and forces. This model is derived with the vehicle driving forward and is valid for small angles.

The tractor wheelbase, L_1 measures the distance between the tires where lateral forces are denoted with subscripts one for the front axle, F_{y1}, and two for the rear, F_{y2}. The tires have a slip angles, α, using the same subscripts. The steering angle is δ. Longitudinal velocity is shown as v_{1x} and lateral velocity is $\dot{y_1}$. The angle orientation of the tractor is ψ_1. The mass, m_1, and mass moment of inertia, J_1, are placed at the center of gravity, which can be located from the front and rear axles using a_1 and b_1. The hitch length, h_1, is measured from the center of gravity to the connection point to the trailer. The distance from the rear axle to the hitch point is e_1.

The trailer wheelbase, L_2, is shown as the distance from the connection point to the rear axle of the trailer. The trailer tire generates a lateral force, F_{y3}, at some slip angle, α_3. The mass, m_2, and mass moment of inertia, J_2, of the trailer is placed at the center of gravity, which can be located with a_2 and b_2. The lateral velocity is denoted as $\dot{y}_2$ and the trailer angle orientation is ψ_2. The difference between the trailer angle orientation and the tractor orientation is what determines the hitch angle, θ.

A few assumptions are made to derive the equations of motion for a tractor-trailer. It assumes constant velocity, no body roll, no aerodynamic forces, small slip angles, the tires are lumped as springs in parallel, and small articulation angles. Luijten and Pacejka appear to implement holonomic constraints due to small angle assumptions for the articulation angles and the tire slip, making things linear.

$$\begin{aligned} \mathbb{Y}_2 &= \mathbb{Y}_1 - h_1 \sin\psi_1 - a_2 \sin\psi_2 \\ \mathbb{Y}_2 &\approx \mathbb{Y}_1 - h_1\psi_1 - a_2\psi_2 \end{aligned} \tag{A.7}$$

The generalized coordinates are as followed:

$$q = [\mathbb{Y}_1 \ \psi_1 \ \psi_2]^T \tag{A.8}$$

This constraint can be rewritten in velocity form:

$$\dot{\mathbb{Y}}_2 - \dot{\mathbb{Y}}_1 + h_1\dot{\psi_1} + a_2\dot{\psi_2} = 0 \tag{A.9}$$

Going back to the fundamental definition of holonomicity, some conclusions can be made. The lateral position can be rewritten to look like the requirement for holonomicity:

$$\begin{aligned} f(q_1, q_2, ...q_n, t) &= 0 \\ \mathbb{Y}_2 - \mathbb{Y}_1 + h_1\psi_1 + a_2\psi_2 + C &= 0 \ \boxtimes \end{aligned} \tag{A.10}$$

This constraint about the fifth wheel is written in terms of all the generalized coordinates, i.e. one can relate $\mathbb{Y}_1, \psi_1$, and ψ_2. The kinematic constraint can be integrated to constrain position; this reduces the DOFs. Thus, it appears the model was created with a holonomic constraint by carefully choosing generalized coordinates for lateral movement. The x coordinate could be included in the generalized coordinates and the derivation would result in the same equation of motion. This is due to the nature of the partial derivatives with respect to the generalized coordinates. Luitjen includes x in the generalized coordinates, but ignores it for the model creation because the forward velocity is constant. According to Popp [88], this quote can provide some confidence in treating this derivation as holonomic:

Real vehicle systems are subject to holonomic constraints only which may be given by geometrical or integrable kinematical conditions. However, in more simplified models, e.g. rolling of a rigid wheel or wheel set on a rigid place, nonholonomic constraints may occur.

Without further adieu, the Lagrange equations of motion used for this derivation are showed. The symbol T represents the kinetic energy, U resembles the potential energy, and Q_i are the generalized forces.

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{q_i}} - \frac{\partial T}{\partial q_i} + \frac{\partial U}{\partial q_i} = Q_i$$

The kinetic energy is expressed as the following:

$$T = \frac{1}{2}m_1(\dot{\mathbb{X}}^2 + \dot{\mathbb{Y}_1}^2) + \frac{1}{2}m_2(\dot{\mathbb{X}}^2 + \dot{\mathbb{Y}}_2^2) + \frac{1}{2}J_1\dot{\psi_1}^2 + \frac{1}{2}J_2\dot{\psi_2}^2 \tag{A.11}$$

The kinetic energy is kept in terms of θ for now, but will be substituted when forming the equations of motion for each generalized coordinate. The kinetic energy can further be expressed with substituting in $\dot{\mathbb{Y}}_2 = \dot{\mathbb{Y}}_1 - h_1\dot{\psi_1} - a_2\dot{\theta}$.

$$\begin{aligned} T = {} & \frac{1}{2}(m_1 + m_2)(\dot{\mathbb{X}}^2 + \dot{\mathbb{Y}_1}^2) + \frac{1}{2}m_2(h_1^2\dot{\psi_1}^2 + a_2\dot{\psi_2}^2 + 2h_1a_2\dot{\psi_1}\dot{\psi_2} - 2\dot{\mathbb{Y}}_1a_2\dot{\psi_2} - 2\dot{y_1}h_1\dot{\psi_1}) + \\ & \frac{1}{2}J_1\dot{\psi_1}^2 + \frac{1}{2}J_2\dot{\psi_2}^2 \end{aligned} \tag{A.12}$$

The potential energy is zero because there are no springs or gravity modeled in the lateral direction.

$$U = 0 \tag{A.13}$$

The partials of the energies will now be taken in each of the generalized coordinates. An equation of motion will be generated for each of the generalized coordinates once all the terms are plugged in to A.2. For $q_i = \mathbb{Y}_1$,

$$\begin{aligned} \frac{d}{dt}\frac{\partial T}{\partial \dot{\mathbb{Y}}_1} &= \frac{d}{dt}((m_1 + m_2)\dot{\mathbb{Y}}_1 - \frac{1}{2}m_2(2a_2\dot{\psi_2} + 2h_1\dot{\psi_1})) \\ &= (m_1 + m_2)\ddot{\mathbb{Y}}_1 - m_2a_2\ddot{\psi_2} - m_2h_1\ddot{\psi_1} \end{aligned} \tag{A.14}$$

For $q_i = \psi_1$,

$$\begin{aligned} \frac{d}{dt}\frac{\partial T}{\partial \dot{\psi_1}} &= \frac{d}{dt}(m_2h_1^2\dot{\psi_1} + m_2h_1a_2\dot{\psi_2} - m_2\dot{y_1}h_1 + J_1\dot{\psi_1} \\ &= m_2h_1^2\ddot{\psi_1} + m_2h_1a_2\ddot{\psi_2} - m_2h_1\ddot{\mathbb{Y}}_1 + J_1\ddot{\psi_1} \end{aligned} \tag{A.15}$$

For $q_i = \psi_2$,

$$\begin{aligned} \frac{d}{dt}\frac{\partial T}{\partial \dot{\psi_2}} &= \frac{d}{dt}(m_2a_2^2\dot{\psi_1} + m_2h_1a_2\dot{\psi_2} - m_2a_2\dot{\mathbb{Y}}_1 + J_2\dot{\psi_1} \\ &= m_2a_2^2\ddot{\psi_1} + m_2h_1a_2\ddot{\psi_2} - m_2a_2\ddot{\mathbb{Y}}_1 + J_2\ddot{\psi_2} \end{aligned} \tag{A.16}$$

The rest of the terms of the kinetic energy do not result in a partial derivative.

$$\begin{aligned}\frac{\partial T}{\partial \mathbb{Y}_1} &= 0\\ \frac{\partial T}{\partial \psi_1} &= 0\\ \frac{\partial T}{\partial \psi_2} &= 0\end{aligned} \tag{A.17}$$

Considering there is no potential energy, the following terms are also zero:

$$\begin{aligned}\frac{\partial U}{\partial \mathbb{Y}_1} &= 0\\ \frac{\partial U}{\partial \psi_1} &= 0\\ \frac{\partial U}{\partial \psi_2} &= 0\end{aligned} \tag{A.18}$$

The generalized forces according to virtual work are basically a scalar summation of the forces on the tires multiplied by a virtual displacement in the direction of the generalized coordinates.

$$\Delta W = F_{y1}\Delta(\mathbb{Y}_1 + a_1\psi_1) + F_2\Delta(\mathbb{Y}_1 - b_1\psi_1) + F_{y3}\Delta(y_1 - h_1\psi_1 - L_2\psi_2) \tag{A.19}$$

Therefore, the generalized forces for each of the states are:

$$\begin{aligned}Q_{y1} &= F_{y1} + F_{y2} + F_{y3}\\ Q_{\psi 1} &= a_1F_{y1} - b_1F_{y2} - h_1F_{y3}\\ Q_{\psi 2} &= -L_2F_{y3}\end{aligned} \tag{A.20}$$

The slip is then incorporated into the model through the generalized forces. For example, the force on the steering axle looks like the following with the normalized cornering stiffness, C_1:

$$\begin{aligned}F_{y1} &= C_1\alpha_1\\ &= C_1(\delta_1 - \frac{\dot{y_1} + a_1\dot{\psi_1}}{v_{1x}})\end{aligned} \tag{A.21}$$

The slip angles for the other forces are listed below:

$$\begin{aligned}\alpha_2 &= -\frac{\dot{y_1} - b_1\dot{\psi_1}}{v_{1x}}\\ \alpha_3 &= \theta - \frac{\dot{y_1} - h_1\dot{\psi_1} - L_2\dot{\psi_2}}{v_{1x}}\end{aligned} \tag{A.22}$$

Now that all the terms have been calculated, they can be plugged in the Lagrange Equation to formulate the equations of motion for each of the generalized coordinates. For $q_i = y_1$,

$$(m_1 + m_2)\ddot{\mathbb{Y}}_1 - m_2a_2\ddot{\psi_2} - m_2h_1\ddot{\psi_1} = F_{y1} + F_{y2} + F_{y3} \tag{A.23}$$

This equation of motion can be expressed in local vehicle coordinates by substituting the relationship $\dot{\mathbb{Y}} = \dot{y_1} + v_{1x}\psi_1$ resembling an active rotation, just linearized. Furthermore, the hitch angle is substituted for the following relationship: $\theta = \psi_2 - \psi_1$.

$$(m_1+m_2)(\ddot{y_1}+v_{1x}\dot{\psi_1}) - m_2a_2\ddot{\psi_2} - m_2h_1\ddot{\psi_1} = C_1\delta_1 - C_1(\frac{\dot{y_1}+a_1\dot{\psi_1}}{v_{1x}}) - C_2(\frac{\dot{y_1}-b_1\dot{\psi_1}}{v_{1x}}) + C_3(\psi_2-\psi_1) - C_3(\frac{\dot{y_1}-h_1\dot{\psi_1}-L_2\dot{\psi_2}}{v_{1x}}) \tag{A.24}$$

This equation of motion can be rewritten such that it has the mass and accelerations on the left and the forces on the right.

$$(m_1+m_2)\ddot{y_1} - m_2h_1\ddot{\psi_1} - m_2a_2\ddot{\psi_2} = -\frac{1}{v_{1x}}[(C_1+C_2+C_3)\dot{y_1} + (C_1a_1 - C_2b_1 - C_3h_1 - (m_1+m_2)v_{1x}^2)\dot{\psi_1} - C_3L_2\dot{\psi_2} + C_3v_{1x}\psi_1 - C_3v_{1x}\psi_2] + C_1\delta_1 \tag{A.25}$$

Next, the equation of motion for $q_i = \psi_1$ can be formulated by plugging the relevant terms into the Lagrange equation. Similarly, the relationships $\dot{\mathbb{Y}} = \dot{y_1} + v_{1x}\psi_1$ and $\theta = \psi_2 - \psi_1$. are substituted in.

$$m_2h_1^2\ddot{\psi_1} + m_2h_1a_2\ddot{\psi_2} - m_2h_1\ddot{\mathbb{Y}}_1 + J_1\ddot{\psi_1} = a_1F_{y1} - b_1F_{y2} - h_1F_{y3} \tag{A.26}$$

$$-m_2h_1^2(\ddot{y_1}+v_{1x}\dot{\psi_1}) + m_2h_1v_{1x}^2\ddot{\psi_1} + J_1\ddot{\psi_1} + m_2h_1a_2\ddot{\psi_2} = a_1C_1(\delta_1 - \frac{\dot{y_1}+a_1\dot{\psi_1}}{v_{1x}}) - b_1(C_2(\frac{\dot{y_1}-b_1\dot{\psi_1}}{v_{1x}})) - h_1(C_3\psi_2 - C_3\psi_1 - C_3(\frac{\dot{y_1}-h_1\dot{\psi_1}-L_2\dot{\psi_2}}{v_{1x}})) \tag{A.27}$$

The equation of motion in the direction of the generalized coordinate ψ_1 is now rearranged such that the masses and accelerations are on the left and the forces are on the right.

$$-m_2h_1^2\ddot{y_1} + (J_1+m_2h_1^2)\ddot{\psi_1} + m_2h_1a_2\ddot{\psi_2} = -\frac{1}{v_{1x}}[(C_1a_1 - C_2b_1 - C_3h_1)\dot{y_1} + (C_1a_1^2 + C_2b_1^2 + C_3h_1^2 - m_2h_1v_{1x}^2)\dot{\psi_1} + C_3L_2h_1\dot{\psi_2} - C_3h_1v_{1x}\psi_1 + C_3h_1v_{1x}\psi_2] + C_1a_1\delta_1 \tag{A.28}$$

The final equation of motion is derived in the direction of $q_i = \psi_2$. Once again, the relationships $\dot{\mathbb{Y}} = \dot{y_1} + v_{1x}\psi_1$ and $\theta = \psi_2 - \psi_1$. are substituted in.

$$m_2a_2^2\ddot{\psi_1} + m_2h_1a_2\ddot{\psi_2} - m_2a_2\ddot{\mathbb{Y}}_1 + J_2\ddot{\psi_2} = -L_2F_{y3} \tag{A.29}$$

$$-m_2a_2(\ddot{y_1}+v_{1x}\dot{\psi_1})+m_2h_1a_2\ddot{\psi_1}+m_2a_2^2\ddot{\psi_2}+J_2\ddot{\psi_2}=-L_2(C_3\psi_2-C_3\psi_1- C_3(\frac{\dot{y_1}-h_1\dot{\psi_1}-L_2\dot{\psi_2}}{v_{1x}})) \tag{A.30}$$

This equation of motion is rearranged so the forces are on the right and the masses and accelerations are on the left.

$$-m_2a_2\ddot{y_1}+m_2h_1a_2\ddot{\psi_1}+(J_2+m_2a_2^2)\ddot{\psi_2}=-\frac{1}{v_{1x}}[(-C_3L_2\dot{y_1}+(C_3L_2h_1-m_2a_2v_{1x}^2)\dot{\psi_1}+ C_3L_2^2\dot{\psi_2}+C_3L_2v_{1x}\psi_2-C_3L_2v_{1x}\psi_1] \tag{A.31}$$

The equations of motion are equations A.25, A.28, A.31. They are currently in a form where the mass matrix is coupled. One can view this as the following where M is the inertia matrix, S is the centrifugal and Coriolis matrix, and E is the input transformation matrix.

$$M\dot{\underline{x}}=-S\underline{x}+E\underline{u} \tag{A.32}$$

The derived equations of motion for an articulated tractor-trailer are formalized in matrix form as the following:

$$\begin{pmatrix} m_1+m_2 & -h_1\,m_2 & -a_2\,m_2 & 0 & 0 \\ -h_1\,m_2 & m_2\,{h_1}^2+J_1 & a_2\,h_1\,m_2 & 0 & 0 \\ -a_2\,m_2 & a_2\,h_1\,m_2 & m_2\,{a_2}^2+J_2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} \ddot{y_1} \\ \ddot{\psi_1} \\ \ddot{\psi_2} \\ \dot{\psi_1} \\ \dot{\psi_2} \end{bmatrix} = -\frac{1}{\mathrm{v_{1x}}}$$

$$\begin{pmatrix} C_1+C_2+C_3 & (m_1+m_2)\,{\mathrm{v_{1x}}}^2+C_1\,a_1-C_2\,b_1-C_3\,h_1 & -C_3\,L_2 & C_3\,\mathrm{v_{1x}} & -C_3\,\mathrm{v_{1x}} \\ C_1\,a_1-C_2\,b_1-C_3\,h_1 & C_1\,{a_1}^2+C_2\,{b_1}^2+C_3\,{h_1}^2-m_2\,h_1\,{\mathrm{v_{1x}}}^2 & C_3\,L_2\,h_1 & -C_3\,h_1\,\mathrm{v_{1x}} & C_3\,h_1\,\mathrm{v_{1x}} \\ -C_3\,L_2 & C_3\,L_2\,h_1-a_2\,m_2\,{\mathrm{v_{1x}}}^2 & C_3\,{L_2}^2 & -C_3\,L_2\,\mathrm{v_{1x}} & C_3\,L_2\,\mathrm{v_{1x}} \\ 0 & -\mathrm{v_{1x}} & 0 & 0 & 0 \\ 0 & 0 & -\mathrm{v_{1x}} & 0 & 0 \end{pmatrix} \begin{bmatrix} y_1 \\ \dot{\psi_1} \\ \dot{\psi_2} \\ \psi_1 \\ \psi_2 \end{bmatrix} +$$

$$\begin{pmatrix} C_1 \\ C_1\,a_1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \delta_1 \tag{A.33}$$

Luijten has an excellent overview on different types of trailers such as semi-trailers, full-trailers, and truck-center axle trailers. A major metric for looking at performance of lateral dynamics of tractor trailers is rearward amplification. In order to reduce the rearward amplification value for

combination vehicles with one articulation, there are a number of things that one can design. 1) make distance between the center of gravity and the coupling point small. 2) the trailer wheelbase should be large. 3) make the tractor more understeered. The kinematic model covers all of these except for the understeer. This can be achieved by modifying the tire cornering stiffness. Luijten states that for semi-trailers, modeling the multiple rear axles did not make a significant difference as it does in truck-center axle trailers.

A.3 State Space Representation

The derived equations of motion are not in state space representation, $\dot{\underline{x}} = A\underline{x} + B\underline{u}$. In order to use modern controls, it is beneficial to modify the system to state space form. Hence, the standard A and B matrices can be found by inverting the mass matrix:

$$\begin{aligned} A &= M^{-1}\, S \\ B &= M^{-1}\, E \end{aligned} \tag{A.34}$$

The A and B Matrices were evaluated using MATLAB's symbolic toolbox and exported directly to LaTeX. The terms are quite lengthy, so the individual elements of the matrices are thus shown separately on horizontal paper at the end of this appendix.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \tag{A.35}$$

$$B = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \\ b_{51} \end{bmatrix} \tag{A.36}$$

A.4 Nominal Parameters

Nominal parameters are provided for a realistic tractor-trailer. The cornering stiffnesses are multiplied by the number of tires modeled to be lumped into a single axes. It is important to note

that the h_1 term differs from the hitch length in the kinematic model. Now this distance is measured from the center of gravity to the mounting point of the trailer. The equivalent value to the hitch length of the kinematic model is $e_1 = a_1 + h_1 - L_1$.

$$
\begin{aligned}
m_1 &= 8994.4\,kg \\
J_1 &= 53314.6\,kg - m^2 \\
a_1 &= 2.5359\,m \\
b_1 &= 3.1977\,m \\
L_1 &= 5.7336\,m \\
h_1 &= 2.9691\,m \\
C_1 &= 2 \times 199505\,\frac{N}{rad} \\
C_2 &= 2 \times 515568\,\frac{N}{rad}
\end{aligned}
\qquad
\begin{aligned}
m_2 &= 27256.8\,kg \\
J_2 &= 729225\,kg - m^2 \\
a_2 &= 6.2941\,m \\
L_2 &= 12.192\,m \\
b_2 &= 5.8979\,m \\
C_3 &= 4 \times 264311\,\frac{N}{rad}
\end{aligned}
$$

$$
A = \begin{bmatrix}
-33.7271 & -30.3989 & -27.5536 & 4.5471 & -4.5471 \\
-1.3857 & -44.7355 & 13.8016 & -2.2776 & 2.2776 \\
0.7331 & -0.6106 & -49.6762 & 8.1979 & -8.1979 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0
\end{bmatrix} \tag{A.37}
$$

$$
B = \begin{bmatrix}
47.9884 \\
17.1624 \\
-0.2815 \\
0 \\
0
\end{bmatrix} \tag{A.38}
$$

A.5 Augmenting the Equations of Motion

The problem with the state space representation of the kinetic tractor-trailer derived thus far is that it is not using the lateral position of the tractor or the trailer as a state. Barbosa in [55] augmented the system matrix with an equation describing the lateral error of the tractor in the forward direction, then calculated the state space representation again with inverting the M and the E matrices.

$$
\begin{aligned}
\dot{d_1} &= \dot{y_1} \cos\psi_1 + v_{1x} \sin\psi_1 \\
\dot{d_1} &\approx \dot{y_1} + v_{1x}\psi_1
\end{aligned} \tag{A.39}
$$

A similar conversion can be done with $\dot{y_2}$, the lateral error of the trailer. One should take care to account for the longitudinal velocity plus the angular velocity in the kinematics of the dynamic

system. However, this is where it was discovered the kinetic model broke down when driving in reverse.

A.6 Model Divergence

With the kinematic model, the velocity simply needed to be negative and the model would drive backwards. The kinetic model, however, would diverge when this was attempted. The equations could simulate in any direction in reverse as long as the steering, δ_1 was zero. To explain the magnitude of this, even a steering angle value of MATLAB's machine tolerance would cause the equations to diverge and the tractor-trailer would immediately jackknife. MATLAB's smallest number used was a value of $2.2204e-16$.

The directions of forces and accelerations were modeled for the tractor-trailer going in the forward direction. This probably means that when driving in reverse, forces were erroneously added to the incorrect direction. Early renditions of using Lagrangian Mechanics to derive the Equations of Motion for the tractor-trailer going in reverse proved to not be successful within the time allotted. Future work can and should be done in deriving the correct model, but this thesis could still be answered with the kinematic model due to the low speeds in reverse.

A.7 Modern Controls with the Kinetic Model

Controlling the kinetic model is different than the kinematic model. The kinematic model used an LQR in which all three states were desired to be controlled. With the kinetic model, there are potentially six states if including the lateral position of the trailer. This means that one would have to generate desired values for each of the states. Planning using Dubins Curves does not provide these reference values. The obvious way to generate reference values are to use a planner like Model Predictive Control where a trajectory can be selected from a bundle of trajectories after using the model to plan with. The trajectory would likely consist of (x, y, t), but each of the states can be determined because of the desired time to reach the next reference point using the model.

Technically, one could use Output Feedback to control variables that are not in the state. This essentially means that one could create a linear combination in the output matrix, $\underline{y} = C\underline{x}$. An observer may be needed, however, because arbitrary pole placement may not work with $A_c = A - BKC$. This means performance may not be guaranteed.

If one does not want to provide reference values for each of the states to generate errors from, one can use set point control [54]. Set point control provides the ability to select three states to

control instead of the full state of six, i.e. controlling a subset of the states. The F matrix in set point control are used as a scaling ratio of input to output of one over the steady state error. The N matrix in set point control is an offset scaled for the desired values. This can be thought of as a bias like gravity, it is always there–so one must account for it. Although this would allow for control of the system similar to the kinematic model using the Dubins planner, set point control was abandoned due to the inability of the current kinetic model to describe the motion of a tractor-trailer in reverse.

$$a_{11} = -\frac{C_1\,J_1\,J_2 + C_2\,J_1\,J_2 + C_3\,J_1\,J_2 + C_1\,J_1\,{a_2}^2\,m_2 + C_2\,J_1\,{a_2}^2\,m_2 + C_3\,J_1\,{a_2}^2\,m_2 + C_1\,J_2\,{h_1}^2\,m_2 + C_2\,J_2\,{h_1}^2\,m_2 - C_3\,J_1\,L_2\,a_2\,m_2 + C_1\,J_2\,a_1\,h_1\,m_2 - C_2\,J_2\,b_1\,h_1\,m_2}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{12} = -\frac{J_1\,J_2\,m_1\,\mathrm{v1x}^2 + J_1\,J_2\,m_2\,\mathrm{v1x}^2 + C_1\,J_1\,J_2\,a_1 - C_2\,J_1\,J_2\,b_1 - C_3\,J_1\,J_2\,h_1 + J_1\,{a_2}^2\,m_1\,m_2\,\mathrm{v1x}^2 + J_2\,{h_1}^2\,m_1\,m_2\,\mathrm{v1x}^2 + C_1\,J_1\,a_1\,{a_2}^2\,m_2 - C_2\,J_1\,{a_2}^2\,b_1\,m_2 + C_1\,J_2\,a_1\,{h_1}^2\,m_2 + C_1\,J_2\,{a_1}^2\,h_1\,m_2 - C_3\,J_1\,{a_2}^2\,h_1\,m_2 - C_2\,J_2\,b_1\,{h_1}^2\,m_2 + C_2\,J_2\,{b_1}^2\,h_1\,m_2 + C_3\,J_1\,L_2\,a_2\,h_1\,m_2}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{13} = \frac{-C_3\,J_1\,m_2\,{L_2}^2\,a_2 + C_3\,J_1\,m_2\,L_2\,{a_2}^2 + C_3\,J_1\,J_2\,L_2}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{14} = -\frac{C_3\,J_1\,m_2\,{a_2}^2 - C_3\,J_1\,L_2\,m_2\,a_2 + C_3\,J_1\,J_2}{J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2}$$

$$a_{15} = \frac{C_3\,J_1\,m_2\,{a_2}^2 - C_3\,J_1\,L_2\,m_2\,a_2 + C_3\,J_1\,J_2}{J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2}$$

$$a_{21} = -\frac{C_1\,J_2\,a_1\,m_1 + C_1\,J_2\,a_1\,m_2 - C_2\,J_2\,b_1\,m_1 - C_2\,J_2\,b_1\,m_2 + C_1\,J_2\,h_1\,m_2 + C_2\,J_2\,h_1\,m_2 - C_3\,J_2\,h_1\,m_1 + C_1\,a_1\,{a_2}^2\,m_1\,m_2 - C_2\,{a_2}^2\,b_1\,m_1\,m_2 - C_3\,{a_2}^2\,h_1\,m_1\,m_2 + C_3\,L_2\,a_2\,h_1\,m_1\,m_2}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{22} = -\frac{C_1\,J_2\,{a_1}^2\,m_1 + C_1\,J_2\,{a_1}^2\,m_2 + C_2\,J_2\,{b_1}^2\,m_1 + C_2\,J_2\,{b_1}^2\,m_2 + C_3\,J_2\,{h_1}^2\,m_1 + C_3\,{a_2}^2\,{h_1}^2\,m_1\,m_2 + C_1\,J_2\,a_1\,h_1\,m_2 - C_2\,J_2\,b_1\,h_1\,m_2 + C_1\,{a_1}^2\,{a_2}^2\,m_1\,m_2 + C_2\,{a_2}^2\,{b_1}^2\,m_1\,m_2 - C_3\,L_2\,a_2\,{h_1}^2\,m_1\,m_2}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{23} = -\frac{C_3\,L_2\,h_1\,m_1\,\left(m_2\,{a_2}^2 - L_2\,m_2\,a_2 + J_2\right)}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{24} = \frac{C_3\,h_1\,m_1\,\left(m_2\,{a_2}^2 - L_2\,m_2\,a_2 + J_2\right)}{J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2}$$

$$a_{25} = -\frac{C_3\,h_1\,m_1\,\left(m_2\,{a_2}^2 - L_2\,m_2\,a_2 + J_2\right)}{J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2}$$

$$a_{31} = -\frac{C_1\,J_1\,a_2\,m_2 - C_3\,J_1\,L_2\,m_2 - C_3\,J_1\,L_2\,m_1 + C_2\,J_1\,a_2\,m_2 + C_3\,J_1\,a_2\,m_2 - C_3\,L_2\,{h_1}^2\,m_1\,m_2 + C_3\,a_2\,{h_1}^2\,m_1\,m_2 - C_1\,a_1\,a_2\,h_1\,m_1\,m_2 + C_2\,a_2\,b_1\,h_1\,m_1\,m_2}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{32} = \frac{C_2\,J_1\,a_2\,b_1\,m_2 - C_3\,J_1\,L_2\,h_1\,m_2 - C_1\,J_1\,a_1\,a_2\,m_2 - C_3\,J_1\,L_2\,h_1\,m_1 + C_3\,J_1\,a_2\,h_1\,m_2 - C_3\,L_2\,{h_1}^3\,m_1\,m_2 + C_3\,a_2\,{h_1}^3\,m_1\,m_2 + C_1\,{a_1}^2\,a_2\,h_1\,m_1\,m_2 + C_2\,a_2\,{b_1}^2\,h_1\,m_1\,m_2}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{33} = -\frac{C_3\,L_2\,\left(J_1\,L_2\,m_1 + J_1\,L_2\,m_2 - J_1\,a_2\,m_2 + L_2\,{h_1}^2\,m_1\,m_2 - a_2\,{h_1}^2\,m_1\,m_2\right)}{\mathrm{v1x}\,\left(J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2\right)}$$

$$a_{34} = \frac{C_3\,\left(J_1\,L_2\,m_1 + J_1\,L_2\,m_2 - J_1\,a_2\,m_2 + L_2\,{h_1}^2\,m_1\,m_2 - a_2\,{h_1}^2\,m_1\,m_2\right)}{J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2}$$

$$a_{35} = -\frac{C_3\,\left(J_1\,L_2\,m_1 + J_1\,L_2\,m_2 - J_1\,a_2\,m_2 + L_2\,{h_1}^2\,m_1\,m_2 - a_2\,{h_1}^2\,m_1\,m_2\right)}{J_1\,m_1\,m_2\,{a_2}^2 + J_2\,m_1\,m_2\,{h_1}^2 + J_1\,J_2\,m_1 + J_1\,J_2\,m_2}$$

$$a_{41} = 0$$

$$a_{42} = 1$$

$$a_{43} = 0$$

$$a_{44} = 0$$

$$a_{45} = 0$$

$$a_{51} = 0$$

$$a_{52} = 0$$

$$a_{53} = 1$$

$$a_{54} = 0$$

$$a_{55} = 0$$

$$b_{11} = \frac{C_1\, J_1\, m_2\, {a_2}^2 + C_1\, J_2\, m_2\, {h_1}^2 + C_1\, J_2\, a_1\, m_2\, h_1 + C_1\, J_1\, J_2}{J_1\, m_1\, m_2\, {a_2}^2 + J_2\, m_1\, m_2\, {h_1}^2 + J_1\, J_2\, m_1 + J_1\, J_2\, m_2}$$

$$b_{21} = \frac{C_1\, \left(a_1\, m_1\, m_2\, {a_2}^2 + J_2\, a_1\, m_1 + J_2\, a_1\, m_2 + J_2\, h_1\, m_2\right)}{J_1\, m_1\, m_2\, {a_2}^2 + J_2\, m_1\, m_2\, {h_1}^2 + J_1\, J_2\, m_1 + J_1\, J_2\, m_2}$$

$$b_{31} = \frac{C_1\, a_2\, m_2\, \left(J_1 - a_1\, h_1\, m_1\right)}{J_1\, m_1\, m_2\, {a_2}^2 + J_2\, m_1\, m_2\, {h_1}^2 + J_1\, J_2\, m_1 + J_1\, J_2\, m_2}$$

$$b_{41} = 0$$

$$b_{51} = 0$$

Appendix B

MOUNTAIN CAR TUTORIAL

It is often suggested to first attempt reinforcement learning algorithms with benchmark problems such as an inverted pendulum, Atari games, and the mountain car. The mountain car domain is a problem where the engine does not have enough torque to get out of the valley; so it must driving back and forth to build up momentum. This is a challenging controls problem because the agent must actually move away from the target in order to reach the objective. OpenAI gym [71] provides the mountain car gym environment shown in Figure B.1.

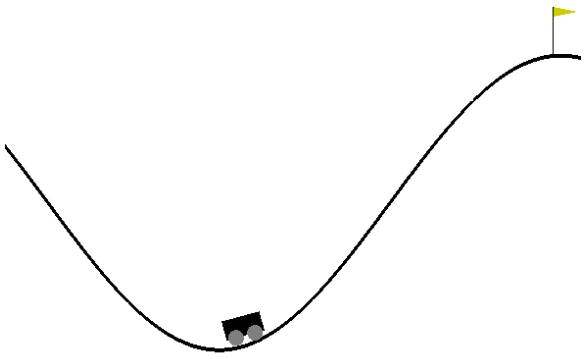


Figure B.1: Mountain Car is a typical benchmark problem to get algorithms working initially.

The state of the vehicle includes position between [-1.2, 0.6] and velocity between [-0.7, 0.7]. The goal is located at x = 0.5. The actions of the agent are discrete, where a = [-1, 0, 1] for full throttle backward, coast, and full throttle forward.

The initial states of the vehicle are randomly selected from a position of [-0.6, -0.4] with no velocity. The environment lasts 200 steps maximum per episode and the rewards are simply -1 for each timestep. The rewards are sparse, so it is an interesting problem for exploration. OpenAI gym environments typically provide reward functions, but when solving a new problem in the real world–reward functions require subject matter experts with an understanding of the optimization or optimal control problem.

Discrete action algorithms such as tabular Sarsa, tabular Q-learning, Sarsa, and DQN were actually investigated on this environment to understand value based methods of reinforcement learning. Investigating Sarsa and DQN was especially helpful in understanding the difference between on-policy and off-policy. Furthermore, REINFORCE and REINFORCE with a baseline were also used to grasp the idea of policy based methods–but the implementation was actually using discrete ac-

tions. Using REINFORCE assisted in the understanding of temporal difference and monte carlo ideas from dynamic programming.

The reason for investigating other algorithms is because DDPG is an actor-critic method, which used components from both value based methods and policy based methods. It was thought that understanding the pitfalls and the implementations would benefit the comprehension when it came time to program the algorithm of DDPG. Recall that DDPG was selected because it provided deterministic, continuous actions.

OpenAI gym also provides a continuous action version called MountainCarContinuous-v0. The reward function is also altered to incentivize less fuel used by subtracting the squared sum of actions. A reward of 100 is awarded for reaching the goal. This environment is used to validate the implementation of the DDPG algorithm used to solve the truck reversal problem in this thesis.

Lillicrap's DDPG algorithm had the following unique advances which made learning more effective. It used soft target updates for both the actor and the critic. It used Uhlenbech & Ornstein to add stochastic noise to the policy for exploration. It included the action to the critic in the second layer. DDPG used batch normalization to help with mixture of state values and units. Finally, it also used L_2 regularization only for the critic. Lillicrap provided suggested values for the hyperparameters, but arguably these are items that may require tuning. In addition, not all of these tricks were found to be beneficial in the mountain car toy problem, nor in reversing a truck and trailer.

The following tables report a progression of the investigation in writing the DDPG algorithm in Python and $tensorflow$ from the pseudo algorithm found in Lillicrap's paper. One can visually monitor the performance of the agent learning using *tensorboard*, however, a convergence criteria established set to automate the early stopping of training. The problem was considered solved if the last 100 episodes of training had a mean reward greater than 90. Due to the stochastic nature of training, three starting seeds were used to evaluate the results. The seeds used were 0, 1, and 12. The training consisted of a maximum of 300 episodes. The results consisted of the average test reward with the saved weights, number of episodes it took to converge, and the overall training time. The idea is to select the best performing DDPG algorithm with certain hyperparameters in a simple grid search.

Table B.1: DDPG training results with no L_2 regularization nor batch normalization.

Avg Test Reward	Reward std	Avg Convergence Ep	Convergence std	Convergence	Avg Train Time [h:min:s]	Time std
92.724	1.158	236	44	2/3	0:32:32	0:20:20

Table B.2: DDPG training results with adding L_2 regularization to the loss, effectively punishing for large weights and biases.

Avg Test Reward	Reward std	Avg Convergence Ep	Convergence std	Convergence	Avg Train Time [h:min:s]	Time std
-67.301	45.818	300	0	0/3	1:45:05	0:02:30

It would appear so far that the implementation without L_2 regularization resulted in better performance, but it still only converged two out of three times. Thus, batch normalization was investigated. In theory, one should set the *is_training* argument to False with the *tensorflow contrib* API when making predictions with the target neural networks for interacting with the environment. This makes use of the stored mean and variance for prediction, which are updated during training. It should be set to True when actually making adjustments to the weights and biases in both the actor and critic. Lillicrap's paper was not extremely clear whether or not to batch normalize the inputs as well as the layers, so it was initially tested without it.

Table B.3: DDPG training results with including batch normalization. The *is_training* argument is set to False when making predictions with the target neural networks during interaction with the environment. The action gradient operation is also set to False. Not batch normalizing the inputs to the critic.

Avg Test Reward	Reward std	Avg Convergence Ep	Convergence std	Convergence	Avg Train Time [h:min:s]	Time std
-34.478	46.255	300	0	0/3	1:37:56	0:22:01

The theoretical batch normalization implementation did not do so well, but what was found to be interesting was that setting the *is_training* argument to True for the target neural networks and the action gradient calculation actually resulted in converging for all three seeds.

Table B.4: DDPG training results with including batch normalization. The *is_training* argument is set to True when making predictions with the target neural networks during interaction with the environment. The action gradient operation is also set to True. Not batch normalizing the inputs to the critic.

Avg Test Reward	Reward std	Avg Convergence Ep	Convergence std	Convergence	Avg Train Time [h:min:s]	Time std
65.419	39.619	186	63	3/3	0:22:02	0:11:32

The test reward was not as high, however, it may be due to the action gradient operation actually needing to be set to False. It should be set to $is\_training = False$ during the operation

because the computational graph uses the online actor network for batch prediction, then updates the weights and biases. The next table actually sets the action gradient $is\_training = False$.

Table B.5: DDPG training results with including batch normalization. The $is\_training$ argument is set to True when making predictions with the target neural networks during interaction with the environment. The action gradient operation is set to False. Not batch normalizing the inputs to the critic.

Avg Test Reward	Reward std	Avg Convergence Ep	Convergence std	Convergence	Avg Train Time [h:min:s]	Time std
91.219	1.655	205	71	2/3	0:26:44	0:15:48

The results improved slightly with the previous table because it received a higher average reward and took fewer episodes to converge, however, it still only converged two out of three times. Due to the uncertainty of whether or not to batch normalize the inputs to the critic, this is evaluated next.

Table B.6: DDPG training results with including batch normalization. The inputs to the critic are now batch normalized.

Avg Test Reward	Reward std	Avg Convergence Ep	Convergence std	Convergence	Avg Train Time [h:min:s]	Time std
60.316	43.375	299	1	1/3	1:00:48	0:28:12

Batch normalizing the inputs to the critic did not appear to improve convergence, nor test results. Now that the best implementation of batch normalization has been selected, it is combined with the L_2 regularization to see if they compliment each other.

Table B.7: DDPG training results with including L_2 regularization and batch normalization. Not batch normalizing the inputs to the critic.

Avg Test Reward	Reward std	Avg Convergence Ep	Convergence std	Convergence	Avg Train Time [h:min:s]	Time std
56.805	47.604	254	65	1/3	0:52:42	0:27:34

As it turns out, Lillicrap's suggestions for using L_2 regularization and batch normalization was not as beneficial for the mountain car problem when using Tensorflow 1.12 and the *contrib* API. The implementation of these tricks utilized open source tools, so it is worth noting it is challenging to reproduce results from research papers.

The computations from batch normalization, despite ironically setting an argument incorrectly, proved to be slightly beneficial in terms of performance and training speed. Typically, batch nor-

malization should speed up training convergence, but additional computations are incurred due to using it. Initial work on the tractor-trailer problem utilized the methods from Table B.5 where batch normalization is used.

It is true that these tricks may vary from problem to problem, however, Lillicrap suggested they used the same algorithm on over 20 continuous action problems. When starting to train on the tractor-trailer problem, it was discovered that training was taking a significant amount of time due to not knowing the ideal reward function. Due to the computational load added from the batch normalization calculations, it was decided to later remove it until minimal performance of reaching the goal was achieved. Due to doing a small study on mountain car, it was apparent that batch normalization offered only a slight speed increase of convergence. It also resulted in a policy that resulted in a slightly lower reward than without it. After finding sufficient results with the tractor-trailer thanks to transfer learning and using an expert policy, batch normalization was attempted again. It provided minimal returns, but that is likely due to the value of the states being relatively similar in range.

Appendix C

CODE

This appendix points to the relevant repositories for code. The files at the current state of this document are zipped and uploaded to Digital Commons, however, code collaboration is encouraged through git. Please fork, add issues, or write pull requests!

C.1 Repositories

controlTrailerKinematics

This repository includes the MATLAB/Simulink files for the initial modern controls investigations in this work. It calculates the LQR gains using the linearized system for the tractor-trailer. The paths are imported as text files and the simulation is run in Simulink for the number of tracks defined. It renders the tractor-trailer and will report the average results to the console. Instructions for usage can be found in the README. Note: the comparison done in this thesis was accomplished solely in python; this is provided as supplementary code.

```
>>> git clone https://github.com/journeyman9/controlTrailerKinematics.git
```

gym-truck-backerupper

This repository contains the OpenAI gym environment for the tractor-trailer simulation built in Python. Follow the README to install the dependencies and get working on your system. One thing to note is that the version included as a zip file to the Digital Commons will NOT work for Windows OS, however, Linux Ubuntu 16.04 and 18.04 are supported. This is simply a simulator, so the control method is left up to the user to define.

```
>>> git clone https://github.com/journeyman9/gym-truck-backerupper.git
```

DDPG

The repository below is for the Deep Deterministic Policy Gradient, which is tailored to the problem of the gym-truck-backerupper. It is written using tensorflow 1.12. Refer to the README

for usage instructions. This code base has complex machine teaching capabilities such as easy ways to use curriculum learning and usage of an expert policy.

```
>>> git clone https://github.com/journeyman9/DDPG.git
```

ModernControl_v_ReinforcementLearning

This repository contains the scripts used for comparing the controllers, creating logs, and generating the figures in this thesis. Refer to the README for usage.

```
>>> git clone https://github.com/journeyman9/ModernControls_v_ReinforcementLearning.git
```