Software-in-the-loop Testing for Autonomous Vehicles With Docker

Sarah Bhaskaran

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-135
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-135.html

May 17, 2022

Acknowledgement

I could not have completed this project without the technical assistance of Dr. Rahul Bhadani and the supervision of Dr. Jonathan Sprinkle. Fangyu Wu, George Gunter, and Dr. Eugene Vinitsky also provided important assistance and contributions. Thank you to Dr. Saleh Albeaik and Dr. Alexandre Bayen for bringing me onboard the research team.

Software-in-the-loop Testing for Autonomous Vehicles With Docker

by Sarah Bhaskaran

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Alexandre Bayen
Research Advisor

5/17/22

(Date)

* * * * * * *

Professor Jonathan Sprinkle
Second Reader

5/16/22

(Date)

Software-in-the-loop Testing for Autonomous Vehicles With Docker

Sarah Bhaskaran
UC Berkeley

Abstract

When developing autonomous vehicle controllers that will act directly on the physical environment, software-in-the-loop (SWIL) testing is important to ensure reliability of the codebase and system. Existing programs to do this job are often difficult to install, clunky, and may lack the granularity or complexity needed for certain experiments. Here, an approach is implemented that uses Docker's containerization to facilitate running and maintaining the simulation. The simulation is comprised of a Python training and evaluation environment, modified so that it can interface with the same ROS components that will be running on a hardware-in-the-loop (HWIL) testbed. This containerized simulation can be easily run by new researchers and with new versions of the controller, making it convenient and useful for controller designers to run SWIL regression testing before deployment.

Plan II Masters Report, UC Berkeley, Spring 2022

1 Introduction

In autonomous vehicle development, many levels of testing are needed to give us confidence that the system is ready to interact with the physical environment on public roads. From the ad-hoc and unit testing performed by controller designers to road tests where humans can take control of the vehicle if needed, each successive stage demands more stringent safety precautions. Large-scale (or even small-scale) testing of vehicles on public roads is extremely expensive and high-stakes, so simulation-based testing does the majority of the work between design and deployment. Simulations facilitate a cyclical design process in which developers iterate on their controllers and quickly test the results of each set of changes. Developing, maintaining, and evaluating the simulations themselves now becomes part of the work of the project collaborators. The practices involved in this process are part of the broader topic of DevOps, which asks how to efficiently cycle between making changes to a project and deploying these changes in a working system (John et al. [2021]). As the field of AV research and technology development grows, those involved in the field will increasingly borrow tools from DevOps, but identify what is particularly appropriate and necessary for the case of autonomous vehicles.

The intermediate stage of testing that is the focus of this work is referred to as software-in-the-loop (SWIL or SIL) testing. Specifically, at this stage, code is intended to be production-ready, but this readiness is confirmed by testing in simulation software, rather than on the hardware system that will be used in deployment (Erkkinen and Conrad [2008]). The space of possible simulations is quite large, as there is no standard for the granularity or the complexity of the simulation. The real world is infinitely complex and intractable to specify, so the simulator occupies a somewhat arbitrary place on this spectrum. Choosing how to test at this stage therefore involves some judgment calls as to how confident one must be before moving to the next stage, processor- or hardware-in-the-loop testing. However, considering the controller that will be tested narrows the scope of the problem. The controller only takes a clearly defined set of inputs and produces defined and constrained outputs. I argue that a customized simulation focused on this state space best enables the developers to evaluate their product.

A key factor for the practical value of any simulator used for SWIL testing existing simulators is the ease of use. If the software is difficult to install and run on standard computing resources, this reduces the likelihood that engineers will thoroughly test their products. And, as mentioned above, bringing testing closer to the design space is helpful to ensure the suitability of SWIL testing. Running and installing programs is a significant barrier given today's proliferation of operating systems and software packages. I examine the conditions that facilitate and hinder software use, and find them to be another compelling reason to build a new simulator.

Once this simulator is built, containerization is a useful tool to borrow from DevOps. Running a program using a containerization platform like Docker is intended to be agnostic to the operating system of the user's local machine or any other software they might have installed or run. In addition to the desire for reproducibility that first motivated this adoption, containerization contributes to the maintainability of the simulation, at least for the duration of the project if privacy concerns bar sharing the codebase or container publicly. Docker is a highly beneficial agent in facilitating the use of DevOps-style SWIL testing towards the deployment of autonomous vehicles.

In the context of the ongoing CIRCLES Consortium project focused on designing and testing autonomous vehicles, this paper traces the modification of a custom simulation for software-in-the-loop testing. I examine the factors that motivated the development of this simulation, and evaluate the extent to which it fulfills its promise of being both valuable and easy to use. The SWIL simulator developed here largely fulfills the need for accessible, targeted testing that enables a DevOps mentality for verifying and validating autonomous vehicle controllers.

2 Related Work

Erkkinen and Conrad [2008] lists software-in-the-loop testing as a crucial step before processor- and hardware-in-the-loop testing of models. SWIL must run on production code and verify that the code performs according to its requirements. In the context of Erkkinen and Conrad [2008], requirements are strictly quantifiable and can be specified in Matlab, permitting automated validation programs to run. The authors also mention code coverage as an important metric appropriate to the SWIL testing phase.

However, beyond these mechanisms which apply to models in general, there are other ways in which SWIL must reassure designers that certain types of models are performing desirably. Pei et al. [2017] warns that when neural networks are involved, code coverage can be 100% even when neuron coverage is only 10%. The authors develop DeepXplore to measure neuron coverage, perturb inputs, and automatically discover problematic corner cases using gradient methods. This process of "white-boxing" machine learning models counteracts some of the uncertainty surrounding how they will perform in conditions on the edge of or outside of the distribution of training data.

The previous works apply to testing robotics software in many domains; AV controller software demands particular care in ensuring that the car can respond safely to the many scenarios that arise on public roads. Rajabli et al. [2020] systematically reviews the approaches to validation and verification of AV software, covering methods of automatically generating test cases, quantifying confidence in ML algorithms, and formally proving safety of algorithms. For example, Schultz et al. [1992] proposes an approach in which faults are gradually added to the simulation in order to test the fault-tolerance of the system. Borg et al. [2018] was an earlier literature survey focusing on machine learning network safety in AVs, with similar considerations to Rajabli et al. [2020].

A similar, somewhat parallel study to this one is AbdelHamed et al. [2020], in which the AV software communicates via a Robot Operating System (ROS) network, and Gazebo is used as the platform for the simulation, as it has been used in my group. The authors develop and evaluate a framework for software-in-the-loop testing of autonomous vehicle software. In addition to the steering, acceleration, and other standard car controls, they primarily focus on testing camera and LiDAR sensors.

In Bhadani et al. [2019], which is also part of the background of this work, both software- and hardware-in-the-loop testing contribute to verification and validation of AV systems. Both the SWIL and HWIL stages involve integrating production-ready code modules into an environment (ROS) that allows them to communicate in the manner that they would in a real operational AV. The testing process is model-based: in SWIL, parts of the system that involve complex real-world inputs (such as sensors) are modeled mathematically and then simulated. This simulated data feeds into the code to be tested. Moving to HWIL, sensor models are replaced with actual sensors. Bhadani et al. [2019] justifies this process and provides a case study with autonomous vehicles, ROS, and Gazebo. Automatic code generation in Simulink is also used to facilitate the translation of models into simulation components.

Containerization is an increasingly common solution for projects relating to both autonomous vehicles and SWIL testing. Wang and Bao [2020] uses several containers for the distributed programs needed to run an actual AV, not simply for testing or simulation. These containers are handled with the open-source Kubernetes, and the setup adds benefits of modularization, versioning, without incurring too much cost in overhead on the vehicle computing resources. Likewise, Berger et al. [2017] advocates for packaging different algorithms on the autonomous vehicle as microservices offered through their own Docker containers for increased security and flexibility of development. White and Christensen [2017] provides a tutorial on using OSRF's Docker images of ROS distributions.

3 Project Background

The approach to simulation development proposed by this paper involves catering to the scope of the AV controller and the structure of the project, so here is an overview of the project so far.

The Congestion Impacts Reduction via CAV-in-the-loop Lagrangian Energy Smoothing (CIRCLES) project runs from January 2020 to December 2022, a collaboration between five universities and additional industry partners (Consortium [2020], Bayen [2020]). As the name suggests, the research primarily focuses on the effects of AVs on the flow of traffic networks. The premise of the project is that if connected autonomous vehicles (CAVs) are present in the traffic network at a certain penetration rate, all of the vehicles in the traffic network can experience energy savings of at least 10%. It turns out that having these controllers in the loop can also improve overall throughput of the network by smoothing standing waves generated by normal human driving behavior.

Towards this goal, the project consists of showing the impacts of CAVs in simulation and then validating these results in field tests with real cars on a public highway. As of writing in May 2022, one medium-scale car test, nicknamed the Vandertest, was already carried out in summer 2020, demonstrating the project's ability to run control algorithms on vehicles on the I-24 highway. The cars ran by themselves for the majority of the drive, but human drivers were at the wheel and occasionally took control when they felt that the autonomous controller was insufficient, such as when a vehicle cut in ahead. Data from the Vandertest is used to investigate weaknesses in the controllers; validate models of car energy consumption; and contribute to training data for machine learning-based controllers.

A new 100 CAV demonstration for fall 2022 is scheduled, where the recent iteration of controllers will run on 100 vehicles in multiple scenarios (Bayen [2021]). This time, the experiment intends to demonstrate the energy reduction and traffic smoothing that is the objective of this project.

3.1 Simulation

As in similar projects, CIRCLES researchers have explored the space of possible simulators for training and testing autonomous vehicles. Different simulators were used for energy modeling, but here we highlight those concerning the flow of traffic, the most macro-scale focus of the project. Earlier research by the same lab group used the Simulation of Urban Mobility (SUMO), the open-source large-scale traffic simulator, with many successful publications and results (Wu et al. [2022]). However, SUMO tries to simulate such a wide and complex variety of traffic-related phenomenon, from signalized intersections to demand routing to multi-modal traffic, that it has become quite bulky and has a steep learning curve. Furthermore, due to the challenges in modeling such complexity, some of the implementation is brittle and exhibits unrealistic behavior. For example, it is unable to represent traffic waves realistically (S. Shanto [2021]), which is a problem for this research project which is centered around damping traffic waves. Therefore, the researchers had already implemented a new simulation, called trajectory_training, on which to train and evaluate controllers. Because it is small (limited in the number of traffic features it simulates), transparent (written in easy-to-parse Python), and self-built, trajectory_training successfully exhibits the behavior that is the main focus of the study. It is continually being extended and thus vulnerable to requirement creep that may eventually overwhelm its useful simplicity, but currently still straightforward to use. The transition from SUMO to trajectory_training is relevant because it demonstrates the value of building a simulator from scratch; this value, in fact, led to trajectory_training becoming the basis of the SWIL simulator that is the focus of this paper.

3.2 Controllers from simulation to real

In terms of CAV controllers, most of the focus is on Multi-Agent Reinforcement Learning-based controllers. These RL controllers learned in simulation to reduce energy consumption across the whole network of AVs. For a controller that is not connected across vehicles, an imitation learning algorithm was trained on a FollowerStopper controller, enabling slight improvement over the baseline of human driving behavior. Some algorithms based on classical control methods are also implemented in simulation for comparison to the RL controller. For the deployment to actual vehicles, safety controllers which are not based on deep neural methods wrap around the main controller and restrict the range of control signals that are sent to the car (Bayen [2021]).

Any controller that was trained on simulation must undergo some changes before being deployed for live testing. Table 1 compares the different simulations to how the AVs will run in the real-life 100-car test. On the vehicle, controller modules pass messages between each other over the ROS (Robot Operating System) network as real-time signals. It is essential to minimize latency when executing algorithms in real time so that controllers operate based on up-to-date data. For lowest latency, code must be compiled from C++, whereas most of the control algorithms are implemented in Python for simulation training and testing. This necessitates manual translation of the code into C++. Some parts of the controller can also be translated into Simulink models, which can be converted into C++ code automatically. However, machine learning models present a special challenge when being incorporated into a ROS module. The weights can be saved as an ONNX model, and Simulink can automatically convert standard ML models from ONNX to C++. But when custom models

Table 1: Simulations in the research group[1]

Aspect of the simulation	Real life	Unmodified trajectory_training	Gazebo SWIL	trajectory_training SWIL
How is the controller implemented?	C++ ROS nodes	Python PyTorch model	C++ ROS nodes	C++ ROS nodes
How is data exchanged between the controller and simulated vehicle?	ROS	Python	ROS	ROS
How is the vehicle simulated?	N/A	Euler updates	Rigid-body kinematics	Euler updates and dynamics model
How is lead distance calculated?	Radar	Python calculation	Ray tracing from simulated sensors	Python calculation
How can lead vehicle trajectory be ingested?	Bagfiles or real-life vehicle	CSVs of existing drives	Bagfiles, simulated or of existing drives	Bagfiles or CSVs of existing drives
Who runs the simulation and reports the results?	100 drivers	Anyone who can clone the repo and run	Controller designers and Rahul Bhadani	Anyone who can pull the docker image and run

are used, this conversion does not work. For this conversion, a new onnx2ros package was developed by the research group.

3.3 Previous SWIL testing

SWIL testing procedures were already developed to validate the safety of controllers for the summer 2021 "Vandertest". Gazebo was used as the simulation platform for software-in-the-loop testing (column "Gazebo SWIL" of table 1. This high-fidelity simulation ran on production code and checked that it behaved safely in several scenarios. The work was based on Bhadani et al. [2019], and as discussed above, different components of the cyberphysical system were modeled in Simulink, converted to C++ by code generation, and included in the Gazebo simulation.

This testing was sufficient to confirm the quality of the software before deployment. However, the process was bottlenecked by the fact that only a couple of researchers had ROS installed on their computers and were familiar enough with Gazebo to run tests. Understanding ROS already involves going through as many as thirty tutorials, and to install it, one must choose between 13 distributions which are not all supported by all operating systems. On top of that, Gazebo is even more finicky during installation than basic ROS because Gazebo includes rendering of images and video. Therefore, it is not surprising that the project's controller designers, who are mostly interested in the mathematics behind reinforcement learning or the simulation of large traffic networks, mostly eschewed the lengthy ROS installation and learning process. The translation and testing of their controllers were left more to the part of the team focused on hardware and safety.

This was not a seamless integration of SWIL testing into the operation of the research team. The inefficiency resulted in limited use of SWIL testing during the development process. In practice, newly written code often goes straight to testing on a car, in which the controller is run but the outputs are not immmediately used to control the car. Researchers can verify that the outputs are reasonable before doing some live hardware-in-the-loop testing. This process is expensive and jumps straight to HWIL without SWIL confirmation.

Gazebo-based simulation is also limited by its inability to scale. Simulating multiple AVs makes it too slow and requires larger computing resources. Therefore, it is unable to check whether production code is similar enough to the code used in controller design and training to exhibit the same benefits for traffic flow and fuel consumption.

[1]Information in the table presented by Dr. Jonathan Sprinkle at 05/05/2022 All-Hands meeting.

4 Problem

The goal of this work is to create a tool for validating the behavior of autonomous vehicle controllers, in a way that conveniently enables setup of the software, running of the simulation, visibility and informativeness of the results, and maintenance under updates. The desired characteristics of this simulation are explored in more detail below.

4.1 Validation of controllers

True validation of ML controllers is extremely challenging due to their inherently black-box nature. The neural networks are not linear or convex in the input features, so there is no guarantee that in input somewhere between two validated inputs will also behave normally. That is only interpolation; extrapolation to inputs out of distribution of the training data is even more unspecified. This leads researchers to argue that machine learning code for autonomous vehicles is fundamentally incompatible with current safety standards for vehicle software, such as ISO 26262 (Borg et al. [2018]).

This work does not pretend to comprehensively validate all possible outputs of a machine learning algorithm, or even to provide the software infrastructure necessary for doing so. Researchers may want to try measuring neural coverage (Pei et al. [2017]) or tolerance to system faults (Schultz et al. [1992]) for these stronger guarantees; many of these tests would require digging further past the abstraction barrier in ways that would further complicate the testing software environment. However, I claim that my framework provides a good entrypoint for running a range of tests on the state space. These tests can be expected to confirm that behavior seen in the ROS-ified, safety-bounded, dynamics-burdened version of controllers matches closely enough the performance of controllers as bare PyTorch models.

4.2 Portability of the simulation

The aim is to substantially limit the amount of time that developers must spend installing software in order to run a simulation of an arbitrary ROS controller. Of course there will always be installation overhead, as the initial version of the simulation must be containerized. Also, Docker occasionally fails to live up to its promise of full portability, as we will see. However, this paper shows how the process of installation requires less careful parsing of documentation, and instead a more direct following of guidelines, as is part of a DevOps mindset (Boettiger [2015]).

4.3 Maintainability of the simulation

This goal is similar to the above in that we would like to minimize time dealing with software compatibility and usage learning curves. But in fact, it is often traded off against the initial portability of the simulation. Building in more flexibility from the beginning means that each user has more parameters to specify even when running something simple. Conversely, the entire simulation can be tightly packaged in a black box, but this makes digging into the black box to extend its capabilities or fix its bugs more difficult. I identify these trade-offs in this simulation and point out opportunities for maintenance as well as potential future struggles.

5 Approach

Rather than further exploring the capabilities of Gazebo, the open-source robotics simulator that had previously been used to test controllers, I adapted the custom trajectory_training simulation developed for the CIRCLES project to interface with the controllers through ROS. I packaged this in a Docker container to increase the simulation's portability and hopefully facilitate future iterations on the simulation and testing framework itself. The structure of the project is shown in Figure 1.

Table 2 lists the input parameters that can most easily be changed, and table 3 highlights some outputs automatically generated by running the simulation.

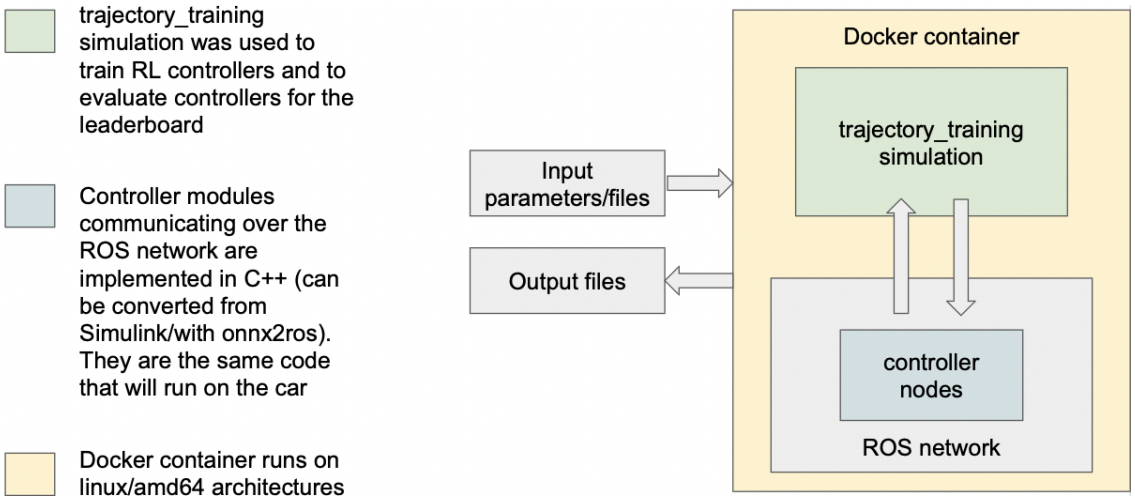


Figure 1

Table 2: Inputs

Lead vehicle
AV controller
Platoon order and composition
Lane changing (enabled or disabled)
Acceleration dynamics model

5.1 trajectory_training environment

This environment allows a platoon of vehicles, composed partly of autonomous vehicles and partly of simulated human-driven vehicles, to follow a given leader trajectory. As a relatively simple and transparent simulation, it has endless opportunities for modification, but here we treat most of the environment as static to preserve consistency with the main branch of the repository. The modifications center around (1) creating an interface between the trajectory_training Python environment and the ROS topics informing and controlling the autonomous vehicles, and (2) passing arguments from the command line to those controller interfaces inside the environment.

An additional simulation feature I added was a module that made acceleration dynamics more realistic. For safety and comfort, acceleration is already cut off between -3 and 1.5m/s^2, but vehicle controllers, especially those tuned in reinforcement learning, may request changes in acceleration within the space of .05 seconds. Engines are physically incapable of providing this jerk instantaneously, and the friction on the road also prevents the requested acceleration from taking effect instantly. When testing code for cyberphysical systems in simulation, mathematical models approximate the way physical phenomena interact with a robotic component (Bhadani et al. [2019]). Accordingly, system identification was used to produce both a first-order and a fourth-order model of acceleration, incorporating a PID controller to simulate the engine's response and a transfer function to simulate other sources of delay (Wu [2022]). I converted these modules into ROS nodes in Matlab Simulink and incorporated them into the trajectory_training simulation. Users can choose which acceleration module to include. This comprises another step towards evaluating controllers under more realistic conditions than those in which they were trained.

Table 3: Outputs

Bagfile recording ROS messages
Time-space diagram
Energy use metrics
Trajectory plot

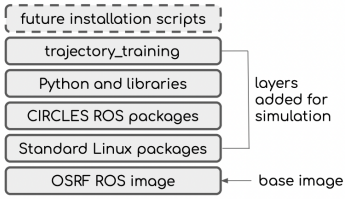


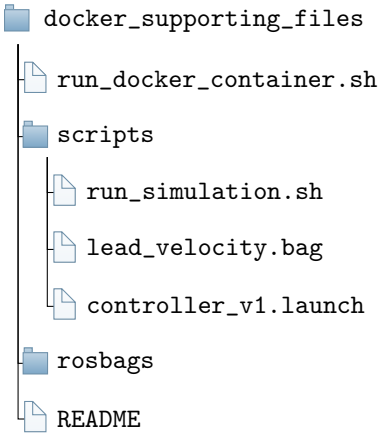
Figure 2: Docker layers

Finally, I added a script that converts a bagfile into a csv that can be used as a lead vehicle trajectory in the trajectory_training simulation with the help of the bagpy Python package (Bhadani). Bagfiles are generated by the ROS networks on this project's test vehicles during test runs. This script enables researchers to conveniently pass in a new real-life driving trajectory behind which to test their controllers. trajectory_training already provides 60 real driving trajectories (Nice et al. [2021]), but new driving tests are run regularly and provide a rich source of possible new test cases for simulation.

5.2 Containerizing with Docker

The Open Source Robotics Foundation facilitates the process of containerization by providing a pre-built Docker image for each ROS distribution (dockerhub [2022]). I started with this image and added more layers to install the necessary Python packages and repositories for the project and build ROS dependencies (2). As a best practice for using Docker for research reproducibility, Boettiger [2015] recommends distributing to other researchers, in addition to a pre-built image, a Dockerfile, which contains line-by-line installation commands. Since the Dockerfile is used to automatically build the image, it serves as a better form of reproducible "recipe" for recreating an environment than text documentation. My Dockerfile is approximately 20 Docker commands and is included in Appendix A.

5.2.1 Supporting file structure



While it would be nice for the Dockerfile and Docker image to be self-contained, a small tree of supporting files and folders was necessary to enable both the installation and the running of the simulation. First of all, ROS handles path resolution by providing two different setup files that must be sourced in order for the `ros*` commands to be found. So, `catkin_make`, `rosrun`, and any initialization of ros nodes, among other commands, must occur inside a bash script.

The most questionable aspect of my design was the inclusion of GitHub private SSH keys in the Docker images. To enable git repositories to be cloned and updated during the initial build, subsequent builds for extension/maintenance, and runs, I created private SSH keys linked to my account, located them in the build folder, and copied them into the image via the Dockerfile. This violates the best practices prescribed for private SSH keys: never copy, never share. Additionally, it limits the scope of where this Docker image can be safely

shared, as it essentially provides access to the project's proprietary information. Alternative solutions proposed on StackOverflow include:

1. SSH keys can be mounted as a volume at build time or run time, using a Buildkit option specifically for SSH so that they do not get copied into the repository.
2. Private Git repositories can be mounted as a volume on the image at runtime (mounting is a means of allowing Docker access to files on the host computer).
3. Private git repositories can be copied into the image by additional layers in a Dockerfile that pulls from the base image.
4. Deployment keys can be issued for each individual Git repository that only provide access to that repository.

For most or all of these options, the additional steps required to add this privacy benefit would decrease the likelihood of people trying the software tool. Due to this concern, I have so far chosen not to follow any of these suggestions. However, (1) appears to be the best option and I am investigating making this change.

For running the simulation, I wanted to make the actual command as simple as possible, but in order to provide the user with flexibility certain options must be specified. The most common modifications can be made to the variables in a bash script (5.2.1). A launch file and bagfile may be placed in the `scripts` folder in the provided file structure; this folder is accessible from inside the Docker container as the mounted `/docker_script` folder. The launch file specifies which controller is used, and the bagfile provides a trajectory that will lead the platoon in the simulation. If users wish to make finer-grained modifications to the simulation, such as allowing lane changes, running multiple AVs with different controllers in the same platoon, or shortening the length of the simulation, they will need to modify the command running `simulate.py`, the entrypoint of trajectory_training that is useful to us. And the actual command to run the docker container hard-codes the arguments that mount the appropriate supporting folders on the container, allowing the user to ignore that file as long as they use the provided supporting file structure.

Finally, in order to view outputs of the simulation, the `/rosbags` folder is mounted to a location inside the Docker container at which some of the ROS outputs are saved. Additionally, all of the outputs generated automatically by the trajectory_training simulation are copied into the `/scripts` folder after the simulation is run. I added one more plot, an emissions graph, to add clarity on the output of the simulation.

5.2.2 Extensions

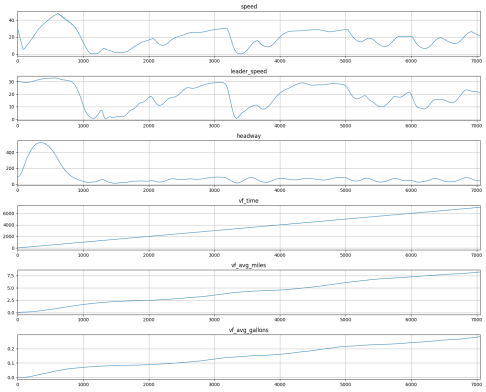
For new controllers that are developed, it is usually necessary to install some additional programs or at least update the repositories. The proposed mechanism for this is via a new Dockerfile that pulls from the base image before running additional installation steps. Usually one or more new supporting files would also be necessary for running this script because, as mentioned, ROS commands need to be run preceded by `source` commands in a shell. Even after pulling from git repositories, `catkin_make` would be necessary at minimum to re-build the ROS packages. For supporting files for build time, users can include them in whatever folder from which they run `docker build`. Supporting files necessary for runtime may be placed in the `scripts` folder that is mounted as `/docker_script` folder.

6 Results

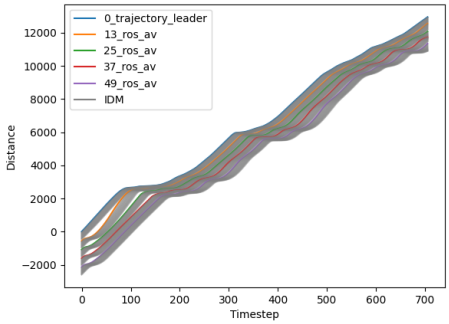
The Appendices provide further specifics on developing, installing, running, and interpreting the software. Appendix A explains the construction of the Docker image; 5.2.1 contains the scripts that run the container and specify the parameters; C shows what to run on the command line for installation and running; D gives an example of how to extend the image to work for a new controller; and E dissects the terminal output for insight on understanding and debugging the simulation.

[2]Trajectory: Westbound I-24 trajectory 2021-04-22-12-47-13; Controller: 08/05 Vandertest, Platoon: Leader Human AV Human AV; Command: `python simulate.py -platoon "av human av human" -traj_path dataset/data_v2_preprocessed_west/2021-04-22-12-47-13_2T3MWRFVXLW056972_masterArray_0_7050/trajectory.csv -av_controller ros -no_lc`

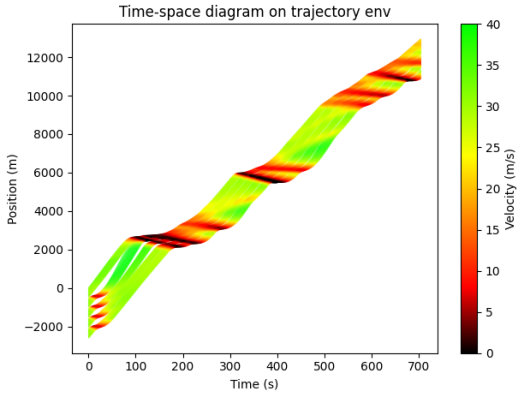
Figure 3: Available output graphs.[2]



(a) Default outputs for simulation as a whole.



(b) Graphed emissions highlighting AV trajectories.



(c) Time-space diagram showing platoon performance.

The simulation runs in real time; most ROS topics are published to at 10Hz or 20Hz, and the trajectories usually last several thousand .1s timesteps. Unfortunately, the way Docker handles the terminal output, no new output shows on the screen for most of the simulation running time. For this reason, I recommend starting with a short horizon (100 timesteps) when first running it.

6.1 Simulation examples

Figure 3 displays several plots that are generated during each run of the simulation. Visual inspection can allow researchers to judge whether the behavior looks smooth, safe, and normal. Figure 4 graphs the topics recorded in a bagfile for each AV that is controlled through ROS. These messages would be published on the car hardware. Seeing the different messages provides debugging guidance for situations when the outputs are not as expected, as well as assurance that the different modules of the safety controller are passing information from node to node as expected.

5a displays a classic trajectory_training scenario that was used throughout RL training. In this scenario, 4 AVs lead platoons of 6 "human" vehicles each, where each "human" vehicle is controlled by the Intelligent Driver Model (IDM). 5b and 5c both illustrate running the RL controller from the Vandertest in the same scenario, the former without acceleration dynamics and the latter with fourth-order acceleration dynamics applied. In both cases, the platoon vehicles keep up with their leaders and do not cause any crashes, but when the controllers communicate through ROS control modules and the acceleration dynamics are applied, the AVs become slightly less effective at damping waves, creating a bit more congestion for the vehicles behind.

Figure 4: Bagfile data graphed

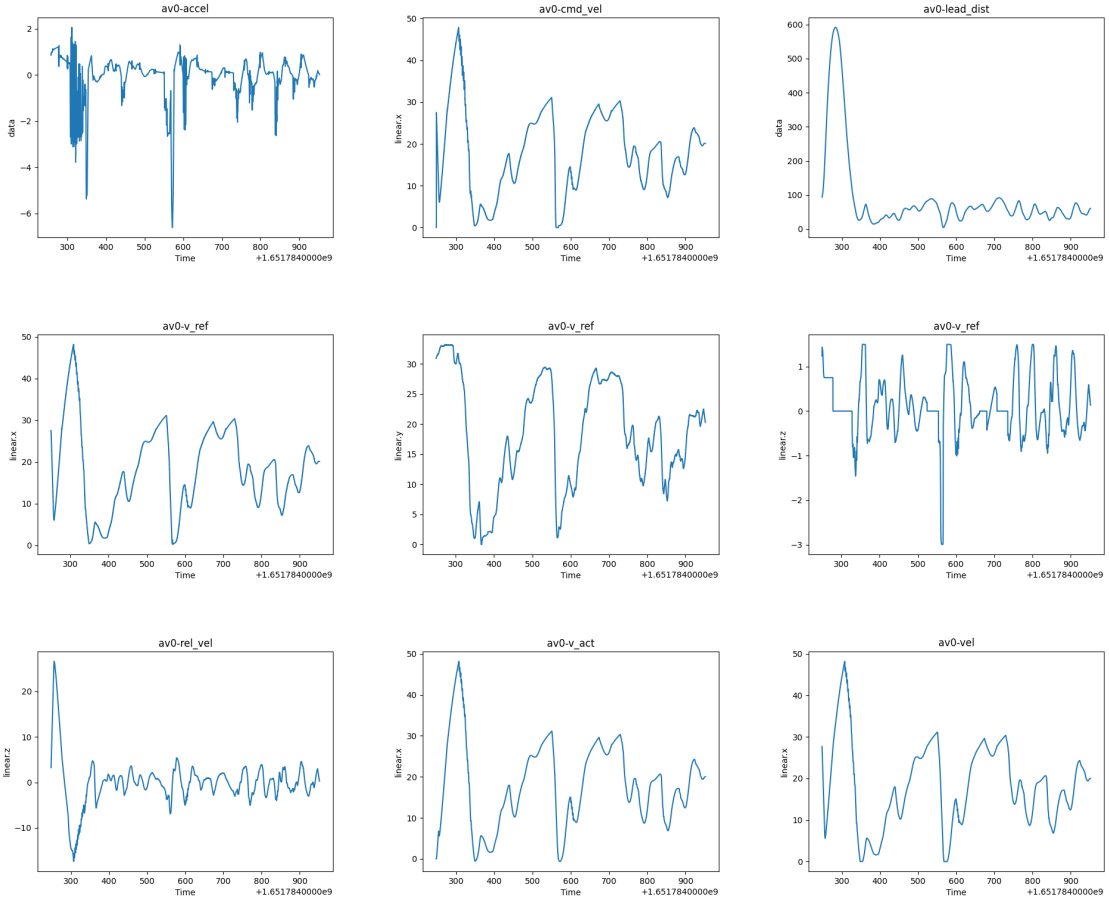
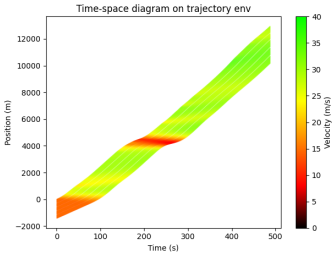
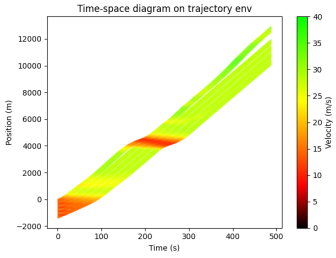


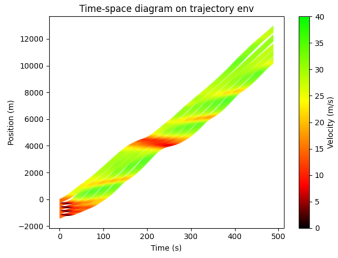
Figure 5: Platoon scenario



(a) Python RL model.



(b) Vandertest RL controller using ROS without acceleration dynamics.



(c) Vandertest RL controller using ROS with acceleration dynamics.

Figure 6: Closer look at a cut-in event

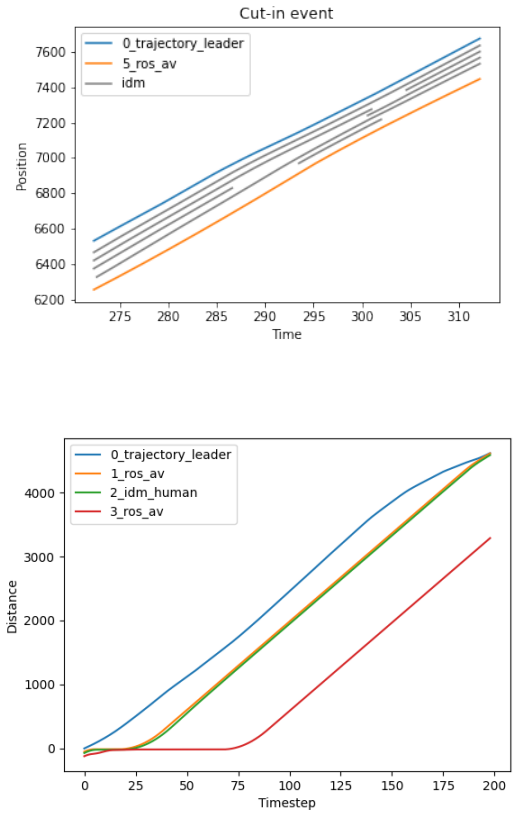


Figure 7: New safety controller running behind simulated eastbound I24 vehicle.

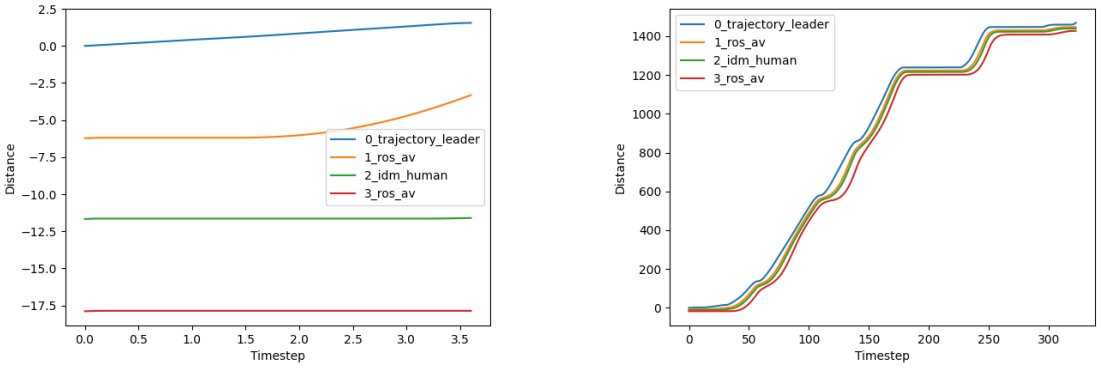
6 demonstrates how one might more closely examine the data from a simulation in which lane changing was enabled. The logged metrics state that 33 cut-in or cut-out events occurred. By doing some exploratory data analysis on the emissions data, we can pinpoint the cut-in events and graph them to better visualize the controller's behavior. We can see that when the human vehicle ahead of the AV speeds up, the AV does not immediately close the gap, and that leads to a car entering the lane in front of it.

7 and 8a are examples of using Docker for SWIL testing. This safety controller is in the process of being developed. For the development process, the safety module is run with a controller that just outputs a constant velocity. In 7, we can tell at a glance that the controller successfully avoids a crash, but behaves in a manner that would be uncomfortable for human passengers. Meanwhile, in 8a, a crash occurred within the first few seconds. It could be that the safety controller does not behave properly for inputs at this slow speed. In comparison, the AVs that were developed for the Vandertest (8b) were able to avoid crashing when presented with the same scenario in simulation.

Controller	Acceleration Dynamics	Crashed on easy trajectory	Crashed on hard trajectory
0805	4th order	No	No
CBF v1	4th order	Yes	No
CBF v1	1st order	No	No
CBF v2	1st order	Yes	No
CBF v2	None	No	No
CBFe	None	Yes	No

Table 4: Inputs resulting in crashes during design iterations on CBF controller.

Figure 8: Using minitest bagfile as lead trajectory



(a) New safety model crashes.

(b) RL controller from Vandertest does not crash.

As part of development of a new version of the CBF (control boundary function) safety controller, we tested this controller against an easy lead trajectory and a harder one. A crash was observed in the harder case, so we tested multiple controllers on these trajectories and varied the acceleration dynamics in case the 4th order model was causing the problem. Results are shown in table 4. Removing the dynamics did allow the new CBF controller (CBF v2) to succeed on the harder trajectory. In fact, the old version of CBF (CBF v1) performed satisfactorily when the first order dynamics model was used, raising the question whether the 4th order dynamics are an important test to check to pass before testing on the road.

Meanwhile, as a control, a version of CBF with a known error (CBFe) failed on the harder trajectory regardless of the dynamics, and the Vandertest controller (0805) passed even with fourth-order dynamics. This shows the harder trajectory with fourth order dynamics is a viable baseline test. However, the easy trajectory was not sufficient to catch any errors in the example controllers. This also suggests that a wider range of tests is important to increase the likelihood of forcing a crash in flawed controllers.

6.2 Ease of setup and use

A researcher unfamiliar with Docker was able to pull the image and run it locally without errors in less than 20 minutes. It remains to be seen whether the tool will be adopted consistently across the research group, and whether users will be able to comfortably tailor tests to their needs. However, the installation process is orders of magnitude faster than it would be to install ROS, trajectory_training, and their assorted dependencies and get them running properly.

Yet Docker's promise of portability did not fully hold up. On a MacOS Monterey containing the Apple M1 chip, ROS processes segfaulted and died, preventing the controllers from running. This bug affected running ROS nodes on the image pulled from osrf/ros. Based on GitHub posts, it seems to be a known effect of faulty qemu emulation when running x64 images on an ARM-based architecture. The only available strategy may be to wait for the OSRF to publish an image for arm64.

Lastly, it should be noted that the size of the image is 10.8 GB. Docker's ability to share layers between images means that one could maintain several different versions stemming from the same image, or run many containers of the same image simultaneously Boettiger [2015], but this size could be prohibitive for researchers who are already using their personal computers intensively for other tasks.

6.3 Design for maintainability

To test the ease of updating the image as software changes, I enabled the running of a newer safety controller, which involved running an installation script. I began the new Dockerfile by pulling `FROM` the image I had developed to work with the Vandertest controller. From there, only four more Dockerfile commands, which achieved the purpose of running the installation script, were needed to set up the image (see Appendix D

for the Dockerfile and script). A similar process was followed for the CBF controllers.[3]This suggests that future installs will also be relatively low-effort. Experimenting with this installation also motivated some other changes to the original image, such as installing most software as a user rather than as root.

Also related to maintainability, some issues with the existing base Docker image became apparent as the trajectory_training repository evolved over the course of the project. Boettiger [2015] cautions about code-rot, a phenomenon in which package dependencies and relationships start to break as packages iterate through versions over time. For example, if a 'git pull' is part of running a container, the run may change as new commits are added to the git repository by external collaborators. Accordingly, Boettiger [2015] advises saving occasional tarballs of the image, as a snapshot to confirm how the code originally worked. However, if one wishes to work with up-to-date software, some maintenance of the software environment is inevitable.

Finally, a challenge with maintainability is that development and debugging are slightly harder in Docker containers than in environments that are not containerized. When running a Docker machine interactively, by default one can only access it through the command line, and there is only one entrypoint (no running multiple shells simultaneously accessing the same Docker container in different working directories or running different programs). This presents a challenge for debugging ROS-networked applications, which require notoriously many terminal windows to run ad-hoc testing. It is possible to use GUI tools, such as `rqt_graph` that is often very helpful for debugging faulty ROS node connections. But viewing a GUI served on a Docker container requires some volume-mounting, networking, and display setup that demands a greater familiarity with Docker and with these software concepts. The process would also be different on different people's local computers where they are running Docker. This is the cost of using containerization rather than full virtualization.

7 Future Work

As development of controllers continues, future work will naturally involve installing additional requirements via new layers of the Docker image. For example, some controllers under development take information about downstream congestion as input. This is not currently one of the ROS messages read by the ROSController in trajectory_training, so `vehicles.py` and `accel_controllers.py` will need to be updated, and new ROS packages will likely be necessary. All of these updates could either be made in `trajectory_script.sh` or in a Dockerfile that pulls from the base Docker image, and it is up to the discretion of the researchers whether the update time is short enough to be endured every time the simulation runs. Either way, these updates will prove the extent to which the existing software provides a robust and flexible foundation.

A large missing piece of the SWIL testing setup remains: generating a set of test cases that can be used as a standard regression test between different controllers and iterations. The state space of the controllers is (currently) limited to the inputs of ego velocity, leader velocity, and space headway, and test cases should cover common traffic phenomena as well as corner cases. See Section 2 for suggestions for testing the state space of deep neural models, but even a simple set of tests for sanity checks would be useful to standardize.

Gazebo simply does not scale for simulating a platoon, but it is built on a more realistic physics engine that may uncover safety risks that trajectory_training would miss. dockerhub [2022] provides a docker image for Gazebo. Future work could explore how to facilitate more widespread use of Gazebo via containerization. These results could be compared to the trajectory_training results to better understand what a high-fidelity simulator adds to the SWIL testing process.

8 Conclusion

This simulation addresses a need for convenient software-in-the-loop testing within the process of AV research and development. More exhaustive and sophisticated simulators are available, but their usefulness depends on many researchers being able to install, run, debug, and comprehend the outputs of a simulator. Previous DevOps processes in the lab group suggest that this tool will enable better integration of SWIL testing into

[3]In fact, for the CBF controllers, since I was working with someone who lacked comfort with writing Dockerfiles, we eased into the process by calling the installation script directly from `trajectory_script.sh`. This has the same effect but the installation script is then run every time the simulation is run.

controller development. Towards this end, containerization encourages researchers to use the simulation by minimizing installation time and easing maintenance.

Some difficulties still exist, notably ROS's failure to run in Docker on ARM architectures. More work is needed to improve the security practices around Git private keys. Maintentance of the packages can still require effort as software evolves.

At writing, this software-in-the-loop testing tool is becoming useful in practice as controller development moves towards the deployment stage for a large field test. It is used by controller designers as they generate production-ready versions of existing algorithms. It serves to debug the functioning of the ROS network; if the vehicle does not run or its behavior is clearly wrong, there may be a gap in the modules' communication. This tool fulfills this iterative-testing need better than previous tools because of its ease of use and portability. Before the field test, it will also be used to refine our expectations of how well the controllers will live up to their original promise of reducing congestion, even when acceleration dynamics are introduced in the model. This tool's flexibility and scalability lets us answer this question with more certainty than previous methods.

Despite its challenges and limitations, this SWIL simulation tool contributes to a streamlined development process well-suited to the demands of this vehicle autonomy research project.

9 Acknowledgments

I could not have completed this project without the technical assistance of Dr. Rahul Bhadani and the supervision of Dr. Jonathan Sprinkle. Fangyu Wu, George Gunter, and Dr. Eugene Vinitsky also provided important assistance and contributions. Thank you to Dr. Saleh Albeaik and Dr. Alexandre Bayen for bringing me onboard the research team.

References

Ahmed AbdelHamed, Girma Tewolde, and Jaerock Kwon. Simulation framework for development and testing of autonomous vehicles. In *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, pages 1–6, 2020. doi: 10.1109/IEMTRONICS51293.2020.9216334.

Alexandre M. Bayen. Circles: Congestion impacts reduction via cav-in-the-loop lagrangian energy, 6 2020. URL `https://www.energy.gov/sites/default/files/2020/06/f75/eems083_bayen_2020_o_5.12.20_1247PM_LR.pdf`.

Alexandre M. Bayen. Circles: Congestion impacts reduction via cav-in-the-loop lagrangian energy, 6 2021. URL `https://www.energy.gov/sites/default/files/2020/06/f75/eems083_bayen_2020_o_5.12.20_1247PM_LR.pdf`.

Christian Berger, Björnborg Nguyen, and Ola Benderius. Containerized development and microservices for self-driving vehicles: Experiences amp; best practices. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 7–12, 2017. doi: 10.1109/ICSAW.2017.56.

Rahul Bhadani. jmscslgroup/bagpy. URL `https://github.com/jmscslgroup/bagpy`. Accessed on 04-30-2022.

Rahul Bhadani, Matt Bunting, and Jonathan Sprinkle. *Model-Based Engineering with Application to Autonomy*, pages 255–285. 12 2019. ISBN 9781119552390. doi: 10.1002/9781119552482.ch10.

Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, jan 2015. ISSN 0163-5980. doi: 10.1145/2723872.2723882. URL `https://doi.org/10.1145/2723872.2723882`.

Markus Borg, Cristofer Englund, Krzysztof Wnuk, Boris Duran, Christoffer Levandowski, Shenjian Gao, Yanwen Tan, Henrik Kaijser, Henrik Lönn, and Jonas Törnqvist. Safely entering the deep: A review of verification and validation for machine learning and a challenge elicitation in the automotive industry, 12 2018.

CIRCLES Consortium. Circles: Using deep reinforcement learning and self-driving cars to improve traffic flow and reduce energy consumption, 2020. URL https://circles-consortium.github.io. Accessed on 2022-04-27.

dockerhub. osrf/ros, 2022. URL https://hub.docker.com/r/osrf/ros/. Accessed on 2022-04-28.

Tom Erkkinen and Mirko Conrad. Verification, validation, and test with model-based design. 10 2008. doi: 10.4271/2008-01-2709.

GitHub. qemu: uncaught target signal 11 (segmentation fault) - core dumped when running docker-compose up on apple silicon 5123. URL https://github.com/docker/for-mac/issues/5123. Accessed on 2022-04-28.

Meenu Mary John, Helena Holmström Olsson, and Jan Bosch. Towards mlops: A framework and maturity model. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 1–8, 2021. doi: 10.1109/SEAA53835.2021.00050.

Matthew Nice, Nathan Lichtle, Gracie Gumm, Michael Roman, Eugene Vinitsky, Safwan Elmadani, Matt Bunting, Rahul Bhadani, Kathy Jang, George Gunter, Maya Kumar, Sean McQuade, Chris Denaro, Ryan Delorenzo, Benedetto Piccoli, Daniel Work, Alex Bayen, Jonathan Lee, Jonathan Sprinkle, and Benjamin Seibold, September 2021. URL https://zenodo.org/record/6456348#.Ym3Dxy-B1O0.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, oct 2017. doi: 10.1145/3132747.3132785. URL https://doi.org/10.1145%2F3132747.3132785.

Nijat Rajabli, Francesco Flammini, Roberto Nardone, and Valeria Vittorini. Software verification and validation of safe autonomous cars: A systematic literature review. *IEEE Access*, PP, 12 2020. doi: 10.1109/ACCESS.2020.3048047.

R. Ramadan B. Seibold D. Work S. Shanto, G. Gunter. Challenges of microsimulation calibration with traffic waves using aggregate measurements, Jan 2021. Accepted for presentation at the Transportation Research Board Annual Meeting.

A.C. Schultz, J.J. Grefenstette, and K.A. De Jong. Adaptive testing of controllers for autonomous vehicles. In *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology*, pages 158–164, 1992. doi: 10.1109/AUV.1992.225178.

Yujing Wang and Qinyang Bao. Adapting a container infrastructure for autonomous vehicle development. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0182–0187, 2020. doi: 10.1109/CCWC47524.2020.9031129.

Ruffin White and Henrik Christensen. *ROS and Docker*, pages 285–307. Springer International Publishing, Cham, 2017. ISBN 978-3-319-54927-9. doi: 10.1007/978-3-319-54927-9_9. URL https://doi.org/10.1007/978-3-319-54927-9_9.

Cathy Wu, Abdul Rahman Kreidieh, Kanaad Parvate, Eugene Vinitsky, and Alexandre M. Bayen. Flow: A modular learning framework for mixed autonomy traffic. *IEEE Transactions on Robotics*, 38(2):1270–1286, apr 2022. doi: 10.1109/tro.2021.3087314. URL https://doi.org/10.1109%2Ftro.2021.3087314.

Fangyu Wu. Vehicle systems identification, 2022. URL https://cocalc.com/fangyuwu/workspaces/sysid. Accessed on 2022-04-28.

A Dockerfile

Below is included the Dockerfile that can be built to generate the base image.

Listing 1: Base image Dockerfile to build jmscslgroup/trajectory_training_swil:v0

```
######################
# ROS and libpanda
######################

FROM osrf/ros:noetic-desktop-full

RUN sudo apt-get -y update && \
  sudo apt-get install -y libusb-1.0-0-dev && \
  sudo apt-get install -y libncurses5-dev && \
  sudo apt-get -y install git-all && \
  sudo apt-get -y install dialog apt-utils && \
  # sudo apt-get -y install openssh-server && \
  sudo apt-get install -y build-essential && \
  sudo apt-get install -y wget && \
  sudo apt-get clean && \
  sudo rm -rf /var/lib/apt/lists/* && \
  adduser --disabled-password --gecos 'docker machine for swil' user1 && adduser user1 sudo && \
  echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers

USER user1

WORKDIR /opt/
RUN sudo git clone https://github.com/jmscslgroup/libpanda.git && \
  cd /opt/libpanda && \
  sudo mkdir build
WORKDIR /opt/libpanda/build
RUN sudo cmake .. && sudo make && \
  git config --global --add safe.directory /opt/libpanda && sudo mkdir /var/panda && sudo chown user1:user1 /var/panda

############################
# Conda
############################

# Install miniconda
ENV CONDA_DIR /home/user1/conda
RUN wget --quiet https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O /home/user1/miniconda.sh && \
/bin/bash /home/user1/miniconda.sh -b -p /home/user1/conda

# Put conda in path so we can use conda activate
ENV PATH=$CONDA_DIR/bin:$PATH

RUN conda init

######################################


##########
# Git ssh
##########

WORKDIR /home/user1
RUN sudo apt-get update && sudo apt-get -y install openssh-server && \
mkdir /home/user1/.ssh
COPY id_ed25519 /home/user1/.ssh/id_ed25519
COPY id_ed25519.pub /home/user1/.ssh/id_ed25519.pub
RUN ssh-keyscan github.com >> /home/user1/.ssh/known_hosts && eval "$(ssh-agent -s)" && \
sudo chown -R user1 /home/user1/.ssh; sudo chmod -R go-rwx /home/user1/.ssh && \
ssh-add /home/user1/.ssh/id_ed25519

##########################
# CIRCLES ROS dependencies
##########################

RUN mkdir /home/user1/catkin_ws && mkdir /home/user1/catkin_ws/src && \
cd /home/user1/catkin_ws/src && \
git clone git@github.com:jmscslgroup/hoffmansubsystem && \
git clone git@github.com:jmscslgroup/followerstoppermax4rl && \
git clone git@github.com:jmscslgroup/followerstoppermax && \
git clone git@github.com:jmscslgroup/followerstopperth && \
git clone git@github.com:jmscslgroup/followerstopperth4rl && \
git clone git@github.com:jmscslgroup/micromodel && \
git clone git@github.com:jmscslgroup/trajectory_07_05_2021_real && \
git clone git@github.com:jmscslgroup/velocity_controller && \
git clone git@github.com:jmscslgroup/integrator && \
git clone git@github.com:jmscslgroup/margin && \
git clone git@github.com:jmscslgroup/can_to_ros && \
git clone git@github.com:jmscslgroup/transfer_pkg && \
git clone https://github.com/sarahbhaskaran/accel_4th_order.git && \
git clone git@github.com:sarahbhaskaran/accel_to_vel.git && \
git clone git@github.com:sarahbhaskaran/accel.git && \
git clone git@github.com:CIRCLES-consortium/algos-stack.git && \
cd algos-stack && git checkout setpoint_rahul && cd .. && \
git clone git@github.com:nathanlct/trajectory_training.git && \
cd trajectory_training && git checkout ros-controller
WORKDIR /home/user1/catkin_ws/src/trajectory_training
RUN conda install -y python==3.8
RUN pip install update && git pull origin ros-controller
RUN pip install -r requirements.txt

######################
```

```
# Build and run trial
#####################

COPY catkin_make_script.sh /home/user1/catkin_make_script.sh
COPY trajectory_script.sh /home/user1/catkin_ws/src/trajectory_training/trajectory_script.sh
SHELL ["/bin/bash", "-c"]
RUN sudo chmod +x /home/user1/catkin_make_script.sh && sudo /home/user1/catkin_make_script.sh && \
  sudo chmod +x /home/user1/catkin_ws/src/trajectory_training/trajectory_script.sh && \
  /home/user1/catkin_ws/src/trajectory_training/trajectory_script.sh
```

B Key supporting files

As described in section 5.2.1, additional supporting files are necessary in order to run the simulation. Script 2 runs the docker container.

Listing 2: Script starting the Docker container

```
# --label doesn't matter but it could help identify the Docker containers to delete them later
# The first volume is our local scripts/ mounted inside the container as docker_script
# Second volume mounts the file where rosbags will be generated
# Currently using jmscslgroup/trajectory_training_swil:v0 but it will be different if you use a different dockerhub repo or tag.
# You have to have that docker image pulled or at least do docker login, otherwise authentication will fail.
# The trajectory_script.sh that it runs is the one in scripts/ on your local computer
# (not the one pulled in the docker container's trajectory_training folder)
docker run --label run_trajectory_script \
-v $(pwd)/scripts:/home/user1/catkin_ws/src/trajectory_training/docker_script \
-v $(pwd)/rosbags:/home/.ros/latest \
jmscslgroup/trajectory_training_swil:v0 docker_script/trajectory_script.sh
```

The above script mounts a folder in the Docker container containing the `trajectory_script.sh` (Listing 3), and then runs this `trajectory_script.sh` inside the container. `trajectory_script.sh` specifies the simulation parameters, runs the simulation, and gathers the outputs.

Listing 3: Script running the simulation

```
# Source the ROS environment
source /opt/ros/noetic/setup.bash
source /home/user1/catkin_ws/devel/setup.bash
# Start the ros master in the background
roscore >/dev/null &
# Update the ROS repository
cd /home/user1/catkin_ws/src/trajectory_training
git pull origin ros-controller

# Input parameters

# Leader trajectory
# Suggestion: dataset/data_v2_preprocessed_west/2021-03-22-22-23-58_2T3MWRFVXLW056972_masterArray_0_4223/trajectory.csv
# or easier: dataset/data_v2_preprocessed_east/2021-04-07-21-22-07_2T3MWRFVXLW056972_masterArray_0_4882/trajectory.csv
# If using a bagfile in scripts/, do docker_script/<bagfile name>.bag
LEADER_TRAJ=dataset/data_v2_preprocessed_west/2021-03-22-22-23-58_2T3MWRFVXLW056972_masterArray_0_4223/trajectory.csv
LEADER_TRAJ_IS_BAG=false
# accel_launch.launch is committed in trajectory_training and currently uses a fourth order model.
# It gets passed in the value of ros_accel as an argument.
# Only relevant when using a single AV. If multiple, just put the acceleration node in the launch file.
ACCEL_MODULE=accel_launch.launch
# If ros_accel is --ros_accel, cmd_accel will be the topic passed through the dynamics model and into the simulation.
# If the variable is empty, cmd_vel will be the topic passed through the dynamics model into the simulation.
# Watch out! Even if this is set wrong, the vehicle might still run. Check the bagfile generated from the run
# to make sure the right topics are being published with the right values.
# ROS_ACCEL=--ros_accel
# Horizon limits the length of the simulation which is good to do when making sure it runs successfully.
# For no horizon set the variable to ""
HORIZON="--horizon 100"
# HORIZON=""

# Convert the bagfile into a csv and put it in an accessible folder, if leader traj is a bagfile
if [[ $LEADER_TRAJ_IS_BAG = true ]]
then
    python bagfile_to_csv.py $LEADER_TRAJ
    LEADER_TRAJ=dataset/data_v2_preprocessed_west/bagfile/trajectory.csv
fi

# Use is_single_av unless you have specifically written a launch file to work with multiple avs, as described in
# https://docs.google.com/document/d/1wXCnonVP3yk4MDscO53HOcJYcZkWwEUDfQi2akWMSzc/edit?usp=sharing
IS_SINGLE_AV=true

# Get the right launch file parameters and run the simulation
if [[ $IS_SINGLE_AV = true ]]
then
  # If the launch file is placed in scripts/, the path to it is docker_script/<launch file name>.launch
  # If the launch file can be accessed from a ros package, you can enclose it in quotes like "transfer_pkg rl0719_readonly.launch"
  # --platoon takes "av" and "human"; make sure that if using --launch_file option there is only one av
```

```
  python simulate.py --platoon "av human" --traj_path $LEADER_TRAJ --no_lc --av_controller ros $HORIZON $ROS_ACCEL \
  --launch_file "transfer_pkg rl0805_readonly.launch"
else
  # If no launch_file_list is specified it will use specific.launch which is committed in trajectory_training.
  # Make sure the length of the launch file list equals the number of AVs in the platoon.
  # (for this example platoon could be "av human av")
  # --platoon takes "av" and "human"
  LAUNCH_FILE_LIST="[specific.launch]*2"
  python simulate.py --platoon "av human av" --traj_path $LEADER_TRAJ --no_lc --av_controller ros $HORIZON $ROS_ACCEL \
  --launch_file_list $LAUNCH_FILE_LIST
fi

# Generate the emissions graph
python emissions_graph.py
# Copy the output data into scripts/
cp -r data/simulate/ docker_script/
```

C Installation and running process

In order to install the software, it suffices to install Docker, and on the command line,

```
docker login
docker pull jmscslgroup/trajectory_training_swil:v0
```

One must obtain the supporting folder structure, and place any bagfiles or launch files that will be used into `scripts` in this supporting folder structure. Then, adjust parameters in `trajectory_script.sh` to the desired ones. Finally, run the software:

```
./docker_run.sh
```

If one wanted to build the base SWIL simulator image from a Dockerfile, they must have an SSH key in the current working directory and run

```
docker build -t jmscslgroup/trajectory_training_swil:<tag of choice> -f Dockerfile .
```

It can be saved on the DockerHub cloud storage by

```
docker push jmscslgroup/trajectory_training_swil:<tag of choice>
```

If one wanted to run the Docker image interactively:

```
docker run --label run_trajectory_script \
-v $(pwd)/scripts:/home/user1/catkin_ws/src/trajectory_training/docker_script \
-v $(pwd)/rosbags:/home/.ros/latest -it  jmscslgroup/trajectory_training_swil:v0
```

D Installing new controllers

To install a new safety controller, a new Dockerfile was written that pulled from the base image and built on it.

```
# Installation and build for new controller, as an additional layer
FROM jmscslgroup/trajectory_training_swil:v0

SHELL ["/bin/bash", "-c"]
COPY install_script.sh /home/user1/catkin_ws/src/trajectory_training/install_script.sh
RUN /home/user1/catkin_ws/src/trajectory_training/install_script.sh
```

The install_script.sh is required as a supporting file:

```
source /opt/ros/noetic/setup.bash
source /home/user1/catkin_ws/devel/setup.bash
cd /home/user1/catkin_ws/src
sudo apt-get update
sudo apt-get install -y ros-noetic-robot-upstart
git clone git@github.com:jmscslgroup/time_to_collision.git
sudo chown -R user1:user1 /home/user1/catkin_ws

cd /opt/libpanda
git status
sudo git pull origin master
pip install empy
sudo chmod +x scripts/install_time_to_collision_Packages.sh
./scripts/install_time_to_collision_Packages.sh
```

Alternatively, this code can be included in the trajectory_script.sh and run in the same container. This makes the container take longer to run, but it saves the user the trouble of learning how to write a Dockerfile and correcting the various issues that arise when porting command line code into that format.

E Interpreting terminal output from the simulation

There are many possible ways to debug problems with the simulation, and the command line output is a good place to start. 4 is an example of terminal output from a working run of the simulator. 5 explains what these outputs refer to and highlights some that are often useful for debugging.

Listing 4: Terminal outputs from a successful run

```
From github.com:nathanlct/trajectory_training
 * branch            ros-controller -> FETCH_HEAD
   46517b9..f165c42  ros-controller -> origin/ros-controller
Updating 46517b9..f165c42
Fast-forward
 README.md                          |  7 ++-
 docker/Dockerfiles/Dockerfile4     | 86 ------------------------------------
 echo_vel.py                        | 30 ++++++++++++
 emissions_graph.py                 |  8 +++-
 simulate.py                        | 14 +++++-
 trajectory/env/accel_controllers.py |  4 ++
 trajectory/env/trajectory_env.py   |  3 +-
 7 files changed, 61 insertions(+), 91 deletions(-)
 delete mode 100644 docker/Dockerfiles/Dockerfile4
 create mode 100644 echo_vel.py
rosbag_record_swil is now alive!
cat: /etc/libpanda.d/vin: No such file or directory
In Save param
/var/panda/CyverseData/JmscslgroupData/bagfiles/2022_05_10/2022_05_10_23_23_41_following_real_vehicle_rl0719_enable_true.csv
[ INFO] [1652225020.806226559]: ** Starting the model "velocity_controller" **

[ INFO] [1652225020.781186997]: Forwarding lead distance events...
[ INFO] [1652225021.054020387]: Recording to '/home//.ros/latest/2022_05_10_23_23_40_NO_VINfs_enable_true.bag'.
[ INFO] [1652225021.055035260]: Subscribing to /rosout_agg
[ INFO] [1652225021.056569497]: Subscribing to /vel
[ INFO] [1652225021.058000794]: Subscribing to /lead_dist
[ INFO] [1652225021.059445271]: Subscribing to /rel_vel
[ INFO] [1652225021.060895914]: Subscribing to /msg_467
[ INFO] [1652225021.062640685]: Subscribing to /accel
[ INFO] [1652225021.064230641]: Subscribing to /rosout
[ INFO] [1652225021.066240643]: Subscribing to /v_act
[ INFO] [1652225021.067661226]: Subscribing to /lead_dist_869
[ INFO] [1652225021.069179317]: Subscribing to /rel_vel_869
[ INFO] [1652225021.070530585]: Subscribing to /steering_angle
[ INFO] [1652225021.071905643]: Subscribing to /msg_921
[ INFO] [1652225021.073264617]: Subscribing to /track_a0
[ INFO] [1652225021.074617328]: Subscribing to /track_a1
[ INFO] [1652225021.075995796]: Subscribing to /rel_vel_old
[ INFO] [1652225021.077356289]: Subscribing to /track_a2
[ INFO] [1652225021.080211992]: Subscribing to /track_a3
[ INFO] [1652225021.082266134]: Subscribing to /track_a4
[ INFO] [1652225021.083774809]: Subscribing to /track_a5
[ INFO] [1652225021.085252842]: Subscribing to /track_a6
[ INFO] [1652225021.086653349]: Subscribing to /track_a7
[ INFO] [1652225021.088134270]: Subscribing to /car/hud/mini_car_enable
[ INFO] [1652225021.089990722]: Subscribing to /track_a8
[ INFO] [1652225021.091402195]: Subscribing to /track_a9
[ INFO] [1652225021.092800195]: Subscribing to /track_a10
[ INFO] [1652225021.094221637]: Subscribing to /track_a11
[ INFO] [1652225021.095612733]: Subscribing to /track_a12
[ INFO] [1652225021.096976505]: Subscribing to /track_a13
[ INFO] [1652225021.098232792]: Subscribing to /track_a14
[ INFO] [1652225021.099624441]: Subscribing to /track_a15
[ INFO] [1652225021.100962636]: Subscribing to /highbeams
[ INFO] [1652225021.102268507]: Subscribing to /region
[ INFO] [1652225021.103617707]: Subscribing to /cmd_vel
[ INFO] [1652225021.105184774]: Subscribing to /timheadway1
[ INFO] [1652225021.107099614]: Subscribing to /cmd_accel_null
[ INFO] [1652225021.108876019]: Subscribing to /v_ref
[ INFO] [1652225020.836418805]: ego_vel_topic/vel
[ INFO] [1652225020.837391751]: relative_vel_topic/rel_vel
[ INFO] [1652225020.837406100]: ego_odom_topic/ego_odom
[ INFO] [1652225020.837417009]: leader_odom_topic/leader_odom
[ INFO] [1652225020.837430784]: headway_topic/lead_dist
[ INFO] [1652225020.837445873]: use_lead_vel0
[ INFO] [1652225020.837458421]: use_odom0
[ INFO] [1652225020.837471681]: use_accel_predict1
[ INFO] [1652225020.855344307]: ego vel topic: /vel
[ INFO] [1652225020.855368205]: relative vel topic: /rel_vel
[ INFO] [1652225020.855382646]: headway topic: /lead_dist
[ INFO] [1652225020.855393524]: ego odom topic: /ego_odom
[ INFO] [1652225020.855403531]: leader odom topic: /leader_odom
[ INFO] [1652225020.855418036]: use lead vel: 0
```

```
[ INFO] [1652225020.855429896]: use  odom: 0
[ INFO] [1652225020.855447333]: headyway scale: 1
[ INFO] [1652225020.855457898]: speed scale: 1
[ INFO] [1652225020.855467943]: We will predict acceleration first. Acceleration will be on linear.z component
[ INFO] [1652225020.855478777]: T Parameter is :0.6
[ INFO] [1652225020.797183776]: ** Starting the model "followerstopperth4rl" **

... logging to /home/user1/.ros/log/367d3342-d0b8-11ec-bd95-0242ac110002/roslaunch-0b97d65786ee-114.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://0b97d65786ee:37199/

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.14

NODES
  /
    bashscript2 (trajectory_training/rosbag_record_swil.sh)

ROS_MASTER_URI=http://localhost:11311

process[bashscript2-1]: started with pid [152]
[bashscript2-1] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
... logging to /home/user1/.ros/log/367d3342-d0b8-11ec-bd95-0242ac110002/roslaunch-0b97d65786ee-113.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://0b97d65786ee:36589/

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.14

NODES
  /
    accel_4th_order (accel_4th_order/accel_4th_order)

ROS_MASTER_URI=http://localhost:11311

process[accel_4th_order-1]: started with pid [147]
[accel_4th_order-1] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
... logging to /home/user1/.ros/log/367d3342-d0b8-11ec-bd95-0242ac110002/roslaunch-0b97d65786ee-112.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://0b97d65786ee:39871/

SUMMARY
========

PARAMETERS
 * /HEADWAY_SCALE: 1.0
 * /SPEED_SCALE: 1.0
 * /T: 0.6
 * /description: following_real_ve...
 * /ego_vel_topic: /vel
 * /enable_fs: True
 * /headway_topic: /lead_dist
 * /hwil: True
 * /margin: 30.0
 * /mode: prompt
 * /model: /home/user1/catki...
 * /readonly: True
 * /relative_vel_topic: /rel_vel
 * /rosdistro: noetic
 * /rosversion: 1.15.14
 * /th1: 0.4
 * /th2: 1.2
 * /th3: 1.8
 * /use_accel_predict: True
 * /use_lead_vel: False
 * /use_margin: False
 * /w1: 4.5
 * /w2: 5.25
 * /w3: 6.0
```

```
NODES
  /
    bashscript2 (can_to_ros/rosbag_record.sh)
    controller (onnx2ros/prompt_mode)
    followerstopperth4rl_node (followerstopperth4rl/followerstopperth4rl_node)
    lead_info (can_to_ros/lead_info)
    saveparam (transfer_pkg/saveparam.py)
    simple_mini_car_from_lead_distance (can_to_ros/simple_mini_car_from_lead_distance)
    subs_fs (can_to_ros/subs_fs)
    velocity_controller_readonly_node (velocity_controller/velocity_controller_node)

ROS_MASTER_URI=http://localhost:11311

process[subs_fs-1]: started with pid [187]
process[lead_info-2]: started with pid [188]
process[simple_mini_car_from_lead_distance-3]: started with pid [193]
process[controller-4]: started with pid [201]
process[followerstopperth4rl_node-5]: started with pid [204]
process[velocity_controller_readonly_node-6]: started with pid [210]
process[bashscript2-7]: started with pid [219]
process[saveparam-8]: started with pid [223]
[saveparam-8] process has finished cleanly
log file: /home/user1/.ros/log/367d3342-d0b8-11ec-bd95-0242ac110002/saveparam-8*.log
[bashscript2-7] killing on exit
[controller-4] killing on exit
[velocity_controller_readonly_node-6] killing on exit
[simple_mini_car_from_lead_distance-3] killing on exit
[subs_fs-1] killing on exit
[lead_info-2] killing on exit
[followerstopperth4rl_node-5] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
WARNING: topic [/car/panda/gps_active] does not appear to be published yet
pygame 2.0.1 (SDL 2.0.14, Python 3.8.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
Created experiment folder at data/simulate/1652225018_10May22_23h23m38s

Running experiment with the following platoon: 0_trajectory_leader 1_ros_av 2_idm_human
with av controller ros (kwargs = {})
with human controller idm (kwargs = {})

Running experiment 1/1, lasting 4223 timesteps.
Using trajectory /home/user1/catkin_ws/src/trajectory_training/dataset/data_v2_preprocessed_west/2021-03-22-22-23-58_2T3MWRFVXLW056972_masterArray_0_4223/trajectory.
Progress: 0.7% (29/4223 env steps)
Progress: 1.9% (80/4223 env steps)
Progress: 3.1% (130/4223 env steps)
Progress: 4.3% (181/4223 env steps)
Progress: 5.5% (231/4223 env steps)
Progress: 6.7% (282/4223 env steps)
Progress: 7.9% (333/4223 env steps)
Progress: 9.1% (384/4223 env steps)
Progress: 10.3% (435/4223 env steps)
Progress: 11.5% (486/4223 env steps)
Progress: 12.7% (536/4223 env steps)
Progress: 13.9% (587/4223 env steps)
Progress: 15.1% (638/4223 env steps)
Progress: 16.3% (689/4223 env steps)
Progress: 17.5% (740/4223 env steps)
Progress: 18.7% (790/4223 env steps)
Progress: 19.9% (841/4223 env steps)
Progress: 21.1% (892/4223 env steps)
Progress: 22.3% (942/4223 env steps)
Progress: 23.5% (992/4223 env steps)
Progress: 24.7% (1043/4223 env steps)
Progress: 25.9% (1093/4223 env steps)
Progress: 27.1% (1144/4223 env steps)
Progress: 28.3% (1195/4223 env steps)
Progress: 29.5% (1246/4223 env steps)
Progress: 30.7% (1296/4223 env steps)
Progress: 31.9% (1346/4223 env steps)
Progress: 33.1% (1397/4223 env steps)
Progress: 34.3% (1447/4223 env steps)
Progress: 35.5% (1498/4223 env steps)
Progress: 36.7% (1549/4223 env steps)
Progress: 37.9% (1599/4223 env steps)
Progress: 39.1% (1650/4223 env steps)
Progress: 40.3% (1701/4223 env steps)
Progress: 41.5% (1751/4223 env steps)
Progress: 42.6% (1801/4223 env steps)
Progress: 43.9% (1852/4223 env steps)
Progress: 45.1% (1903/4223 env steps)
Progress: 46.3% (1954/4223 env steps)
Progress: 47.5% (2004/4223 env steps)
Progress: 48.7% (2055/4223 env steps)
Progress: 49.8% (2105/4223 env steps)
Progress: 51.0% (2155/4223 env steps)
Progress: 52.2% (2206/4223 env steps)
Progress: 53.4% (2257/4223 env steps)
Progress: 54.6% (2307/4223 env steps)
Progress: 55.8% (2357/4223 env steps)
```

```
Progress: 57.0% (2408/4223 env steps)
Progress: 58.2% (2459/4223 env steps)
Progress: 59.4% (2509/4223 env steps)
Progress: 60.6% (2560/4223 env steps)
Progress: 61.8% (2610/4223 env steps)
Progress: 63.0% (2661/4223 env steps)
Progress: 64.2% (2712/4223 env steps)
Progress: 65.4% (2762/4223 env steps)
Progress: 66.6% (2812/4223 env steps)
Progress: 67.8% (2863/4223 env steps)
Progress: 69.0% (2914/4223 env steps)
Progress: 70.2% (2965/4223 env steps)
Progress: 71.4% (3016/4223 env steps)
Progress: 72.6% (3066/4223 env steps)
Progress: 73.8% (3117/4223 env steps)
Progress: 75.0% (3167/4223 env steps)
Progress: 76.2% (3217/4223 env steps)
Progress: 77.4% (3268/4223 env steps)
Progress: 78.6% (3318/4223 env steps)
Progress: 79.8% (3368/4223 env steps)
Progress: 81.0% (3419/4223 env steps)
Progress: 82.2% (3470/4223 env steps)
Progress: 83.4% (3521/4223 env steps)
Progress: 84.6% (3572/4223 env steps)
Progress: 85.8% (3623/4223 env steps)
Progress: 87.0% (3673/4223 env steps)
Progress: 88.2% (3723/4223 env steps)
Progress: 89.4% (3774/4223 env steps)
Progress: 90.6% (3825/4223 env steps)
Progress: 91.8% (3875/4223 env steps)
Progress: 93.0% (3926/4223 env steps)
Progress: 94.2% (3977/4223 env steps)
Progress: 95.4% (4028/4223 env steps)
Progress: 96.6% (4079/4223 env steps)
Progress: 97.8% (4129/4223 env steps)
Progress: 99.0% (4180/4223 env steps)
Progress: 100.0% (4223/4223 env steps)
Saved emissions file at data/simulate/1652225018_10May22_23h23m38s/emissions/emissions_1.csv
Wrote data/simulate/1652225018_10May22_23h23m38s/figs/training_1.png
Wrote data/simulate/1652225018_10May22_23h23m38s/figs/custom_metrics_1.png
Wrote data/simulate/1652225018_10May22_23h23m38s/figs/base_state_1.png
Wrote data/simulate/1652225018_10May22_23h23m38s/figs/sim_data_av_1.png
Wrote data/simulate/1652225018_10May22_23h23m38s/figs/platoon_0_1.png
Wrote data/simulate/1652225018_10May22_23h23m38s/figs/system_1.png
Wrote data/simulate/1652225018_10May22_23h23m38s/figs/speed_accel_profiles_1.png
Wrote data/simulate/1652225018_10May22_23h23m38s/figs/time_space_diagram_1.png


Metrics aggregated over 1 runs:

system_mpg: 36.86  0.00 (min = 36.86, max = 36.86)
system_speed: 28.51  0.00 (min = 28.51, max = 28.51)
av_mpg: 34.00  0.00 (min = 34.00, max = 34.00)
platoon_0_mpg: 38.37  0.00 (min = 38.37, max = 38.37)
count_crash: 0.00  0.00 (min = 0.00, max = 0.00)
count_low_headway_penalty: 0.00  0.00 (min = 0.00, max = 0.00)
count_large_headway_penalty: 3701.00  0.00 (min = 3701.00, max = 3701.00)
count_low_time_headway_penalty: 0.00  0.00 (min = 0.00, max = 0.00)
av_headway (mean): 551.36  0.00 (min = 551.36, max = 551.36)
av_headway (std): 310.59  0.00 (min = 310.59, max = 310.59)
av_headway (min): 61.21  0.00 (min = 61.21, max = 61.21)
av_headway (max): 1025.89  0.00 (min = 1025.89, max = 1025.89)
av_speed (mean): 28.40  0.00 (min = 28.40, max = 28.40)
av_speed (std): 5.67  0.00 (min = 5.67, max = 5.67)
av_speed (min): 5.09  0.00 (min = 5.09, max = 5.09)
av_speed (max): 46.12  0.00 (min = 46.12, max = 46.12)
platoon_0_speed (mean): 28.40  0.00 (min = 28.40, max = 28.40)
platoon_0_speed (std): 5.63  0.00 (min = 5.63, max = 5.63)
platoon_0_speed (min): 5.81  0.00 (min = 5.81, max = 5.81)
platoon_0_speed (max): 43.15  0.00 (min = 43.15, max = 43.15)
av_leader_speed_difference (mean): 2.22  0.00 (min = 2.22, max = 2.22)
av_leader_speed_difference (std): 6.09  0.00 (min = 6.09, max = 6.09)
av_leader_speed_difference (min): -16.53  0.00 (min = -16.53, max = -16.53)
av_leader_speed_difference (max): 24.76  0.00 (min = 24.76, max = 24.76)
instant_energy_consumption (mean): 1.87  0.00 (min = 1.87, max = 1.87)
instant_energy_consumption (std): 1.86  0.00 (min = 1.86, max = 1.86)
instant_energy_consumption (min): 0.00  0.00 (min = 0.00, max = 0.00)
instant_energy_consumption (max): 24.21  0.00 (min = 24.21, max = 24.21)
rl_reward (mean): -0.94  0.00 (min = -0.94, max = -0.94)
rl_reward (std): 0.71  0.00 (min = 0.71, max = 0.71)
rl_reward (min): -3.42  0.00 (min = -3.42, max = -3.42)
rl_reward (max): 1.00  0.00 (min = 1.00, max = 1.00)
rl_episode_reward: -3967.98  0.00 (min = -3967.98, max = -3967.98)
n_cutins: 0.00  0.00 (min = 0.00, max = 0.00)
n_cutouts: 0.00  0.00 (min = 0.00, max = 0.00)
n_vehicles (mean): 3.00  0.00 (min = 3.00, max = 3.00)
n_vehicles (std): 0.00  0.00 (min = 0.00, max = 0.00)
n_vehicles (min): 3.00  0.00 (min = 3.00, max = 3.00)
n_vehicles (max): 3.00  0.00 (min = 3.00, max = 3.00)

Experiment logs have been saved at data/simulate/1652225018_10May22_23h23m38s/logs.txt
Experiment folder is data/simulate/1652225018_10May22_23h23m38s
```

Table 5: Terminal outputs from a successful run

Line numbers	Source	Notes
1-15	`git pull` command in trajectory_script.sh	Updating the trajectory_training repository with changes since when the docker image was built
16	just_bag.launch	Added my own bagrecord node that works on a non-RPi computer
17	ROS nodes	Error because there is no VIN in the appropriate folder in libpanda, but that does not cause problems
18-200	ROS nodes	Runs through the output from starting the launch file twice because create_simulation() function runs an extra time before the sim starts
24-59	ROS nodes	Tells you what topics have been created (what topics the rosbag_record is recording in the bagfile)
200	rosbag_record.sh script in can_to_ros package	The usual rosbag_record.sh doesn't work, which is why I added the just_bag.launch to run rosbag_record_swil.sh
201-349	trajectory_training simulation	Normal outputs from trajectory_training
211-294	trajectory_env.py	So many progress updates because ROS nodes run in real time, but the progress updates don't necessarily show up on the terminal immediately when being run inside a docker container.
349	simulate.py	Name of the folder where all outputs except rosbags are saved