TOWARDS SUPPORTING END-TO-END LEARNING INSIDE A HARDWARE-SOFTWARE INFRASTRUCTURE IN ROBOTICS

by

Thomas Hansen

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

May, 2023

Date of final oral examination: June 26th, 2023

Committee members:
Dan Negrut
Peter Adamczyk
Radu Serban

place holder

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Professor Dan Negrut, for his advice and guidance through this project. Further I would like to thank my committee members, Distinguished Scientist Radu Serban and Professor Peter Adamczyk.

I would also like to thank Luning Bakke for her help writing PyChrono simulations, as well as Huzaifa Unjhawala for his work calibrating the ART vehicle throttle. Additionally the team that worked on ART was indispensable, including Harry Zhang, Stefan Caldararu, Sriram Ashokkumar, Shouvik Chatterjee, Alexis Ruiz, and others.

Finally, I would like to thank everyone in the Simulation Based Engineering Laboratory for their support and assistance through this process.

CONTENTS

Contents . . . iii

List of Tables . . . vi

List of Figures . . . vii

1 Introduction . . . 1
- 1.1 Motivation . . . 1
- 1.2 Overview . . . 1
- 1.3 Thesis Organization . . . 3

2 Background . . . 4
- 2.1 Chrono . . . 4
- 2.2 PyChrono . . . 4
- 2.3 Chrono Terrain Model . . . 4
- 2.4 Chrono Vehicle Model . . . 6
  - 2.4.1 Overview of dART . . . 7
  - 2.4.2 Suspension . . . 7
  - 2.4.3 Steering . . . 8
  - 2.4.4 Drivetrain . . . 9
  - 2.4.5 Tire Models . . . 9
  - 2.4.6 Brakes . . . 10
  - 2.4.7 Throttle Calibration . . . 10
- 2.5 Robot Operating System . . . 12
- 2.6 Docker . . . 14
- 2.7 Docker Compose . . . 14
- 2.8 PyTorch . . . 15

3 System Overview . . . 16
- 3.1 Real World ART . . . 16
  - 3.1.1 ART . . . 16
  - 3.1.2 ARCLab Motion Capture . . . 16
- 3.2 Simulated Environment . . . 17
  - 3.2.1 Chrono and PyChrono . . . 17
  - 3.2.2 dART . . . 17
  - 3.2.3 Gym . . . 18
  - 3.2.4 Stable Baselines3 . . . 18

4 Autonomy Research Testbed . . . 19

4.1 Autonomy Research Testbed Hardware Components 19
4.1.1 Power Distribution 19
4.1.1.1 Motor Power 20
4.1.1.2 Jetson Power 20
4.1.1.3 High Level Electrical Diagram 21
4.1.2 Vehicle Steering 23
4.1.3 Chassis 23
4.2 Sensors 24
4.2.1 Camera 24
4.2.2 Xsens MTi-7 GNSS/INS 25
4.3 Motion Capture and Retroreflective Markers 26
4.4 Onboard Computer 27
4.5 Autonomy Research Testbed Software Components 29
4.5.1 Software Structure 29
4.5.2 Setting up Environment 30
4.5.3 Running in Simulation 32
4.5.4 Running in Reality 34
4.5.5 Data Collection 36
4.6 Accessing the Jetson 36

5 Autonomy Toolkit 38
5.1 Dev Service 38
5.1.1 NX Service 41
5.2 Chrono Service 42
5.3 noVNC Viewer 43
5.4 Hardware in the Loop Testing 44

6 End to End Learning 46
6.1 Reinforcement Learning 47
6.1.1 OpenAI Gym 47
6.1.2 gym-chrono 49
6.1.3 Custom Gym Environments 49
6.1.3.1 Register Environment 50
6.1.3.2 Choosing Action and Observation Spaces 51
6.1.3.3 Custom Environment Structure 52
6.1.3.4 Reward Function 53
6.1.3.5 Cone Setup 56
6.1.4 Stable Baselines3 57
6.1.4.1 Policy Proximal Optimization (PPO) 60
6.2 Training in Chrono Environment 61
6.2.1 Training with Gym and Stable_Baselines3 61

6.3 Running the Agent . . . 62
6.4 Observations and Challenges . . . 64

7 Conclusions and Contributions . . . 66

8 Future Work . . . 68
8.1 Future Work on Learning Model . . . 68
8.2 Future Work Simulation . . . 69
8.3 Future Work on ART Vehicle . . . 70

Bibliography . . . 71

LIST OF TABLES

LIST OF FIGURES

1.1 The ditigal Autonomy Research Testbed (dART) driving through a cone path using PID controls. 2

2.1 Here you can see the ruts in the soil match the groves on the tire. 5
2.2 Example of double wishbone suspension, image from citation [33] 7
2.3 A diagram of what the Pitman arm motion looks like, from Project Chrono's page on wheeled steering. 8
2.4 A diagram of what the Pitman arm topology looks like, from Project Chrono's page on wheeled steering. 9
2.5 The ramp and throttle tests performed by the vehicle [37]. 11
2.6 The mean posterior torque and loss curves [37] 11

3.1 Shown here is a picture of the original ART vehicle. The white bumper indicates the front of the vehicle, with a GPS/IMU sensor hanging off the rear suspension. 17
3.2 An overview of the ART platform, where we can see the interface of the Autonomy Stack and its associated algorithms, with the simulation or real vehicle. 18

4.1 Image of the buck converter on ART1. It's displaying the output voltage of 11 volts, and the Jetson power rocker switch is on (green) while the motor rocker switch (red) is off. Unmentioned are the power distribution blocks (right and left), which just serve to connect wires in parallel to each other. 21
4.2 A high level diagram of the power distribution for the system. We have three buses, one for 11v, 5v, and gnd. These distribute power to each of the sensors and components and all share a common ground. 22
4.3 In image of the front of the ART2 vehicle, where one can see the linkages that make up the pitman arm steering. 23

4.4 Here one can see the camera in its mount. The two red mounts allow for tilt forward and back, while the bottom mount and bumper allow for swivel. This all sits on the front of the vehicle, ahead of the suspension and wheels. 25

4.5 Here we can see the component board with labels for the different holes, in this case being four we used for the retroreflective marker placement. Notably missing are the two markers we attached to the front and rear suspension of the vehicle, and the marker which can sit on top of the lidar mount. 27

4.6 Here are two retroreflective markers which were placed on the second ART vehicle. 28

4.7 Here we can see, going from left to right, the Jetson Xavier NX, Jetson Xavier AGX, and the Jetson Xavier Orin. Each is powered by a variable output buck converter onboard the vehicle or the provided power supply which can be plugged into the wall. 29

4.8 A brief overview of the folder structure, with ART representing the root folder. Notably the blue folders are only built after `colcon build` is run. 29

4.9 Image of what my Optix7.5 install looks like. The version and name for the container must be the same to run properly, and the installer can only be downloaded from Nvidia's website. 32

4.10 Here the two switches can be seen on the ART2 vehicle, as well as the buck converter in the middle. The green switch on the right controls the Jetson, and the red switch on the left controls the motors and steering. In the center is the buck converter. 34

4.11 The power switch is boxed in red, and is the switch closest to the side. Do not press the other switches, as holding them down can cause a forced reset. 35

5.1 Diagram of the different nodes and topics that may be enabled while running the `dev` container 39

5.2 Example of what the noVNC viewer will look like before starting a simulation. The user should be able to interact with the screen to open or close windows. . . 43

6.1 Here you can see a high level diagram of what the software structure would look like in the ART platform. Here you can see many of the topics and nodes in Fig. 3.2 have been removed in favor of a reinforcement learning agent (running Proximal Policy Optimization, or PPO). 46
6.2 Here we can see a high level overview of the communication channels created by Gym. The agent sends actions to the environment (in this case a custom environment) which returns rewards and observations, allowing it to then perform another action. This repeats until the custom environment either finishes or cannot proceed anymore. 48
6.3 A brief overview of how gym-chrono is structured. Blue boxes mean the folder does not need to be modified in order to add another environment. New folders can be made so long as they are registered within Gym. 50
6.4 An example of a sparse reward function and a shaped reward function. Implementing sparse functions over variables which could be shaped can result in slow or no learning, as the agent must try every possibility until it reaches the correct solution. 54
6.5 An sample of one of the cone layouts from training. 56
6.6 An example of a shorter, straight cone path. This is a simpler path, which may be easier to train. 56
6.7 The first frame of training, while the vehicle is still at its starting position. You can see the cone path in front of it. This frame was taken at 720p resolution. . . 57
6.8 A frame of the scaled resolution being used by the vehicle, with a lower resolution. 58
6.9 Example of semi-circle, with the vehicle in its starting position of $[0, 0]$. The rear of the dART vehicle is surrounded by the semi-circle, while the front is pointing towards the opening in the cones. 59

6.10 One of the bird's-eye view images taken from training. Here the vehicle, surrounding landscape, and cones are visible. The course was to learn to drive forward 6 meters, which it is successfully completing in this final frame. 63
6.11 The dART vehicle can be seen escaping the path by driving between two cones. 64

8.1 An example of GAN images being used in simulation in the ARC lab, compared to the real images [10]. 69

TOWARDS SUPPORTING END-TO-END LEARNING INSIDE A HARDWARE-SOFTWARE INFRASTRUCTURE IN ROBOTICS

Thomas Hansen

Under the supervision of Professor Dan Negrut
at the University of Wisconsin-Madison

This thesis project augments a framework that combines software and hardware elements (Autonomous Research Testbed platform, Chrono, OpenAI Gym, and Stable Baselines3), and whose goal is that of enabling end-to-end learning in robotics. In a broader sense, this framework is also meant to assist research efforts in the Simulation-Based Engineering Lab that target the sim-to-real gap in robotics. This thesis touches on two major topics. First is the ART vehicle, for which this work provides a detailed discussion of how the vehicle has been designed, realized, and how to use it. From there, the thesis goes over the associated software stack and the corresponding digital twin, dART. The second topic is tied to the approach that one can take to enable, via simulation, end-to-end learning in robotics. To that end, in simulation, raw sensor input is used to train a reinforcement learning agent to navigate an off-road course. The course is defined by cones on each side of the path, and the vehicle must not collide with them while reaching its final destination at the end of the path. In terms of future work, it remains to increase the accuracy and the robustness of the end-to-end control policy obtained, and to demonstrate that the control policy produced works as expected when deployed in the real world.

Dan Negrut

1 INTRODUCTION

1.1 Motivation

Simulation is a lucrative and evolving field, and it can be a powerful tool when designing new autonomy algorithms. In many cases, building and testing all aspects of an autonomous algorithm in real life is infeasible or financially impractical, and testing algorithms in simulation first could not only reduce end costs but produce a better, safer product in less time [5].

With this in mind, this thesis is a research effort designed to construct an end-to-end autonomous system based on the Gym library and implemented within the lab's gym-chrono package. Gym is an open source reinforcement learning framework which makes it easier to implement reinforcement learning in simulation. With this in mind, Gym will be used to train the simulated dART vehicle to follow a path defined by cones, which could be translated into real world use. Further to better model off-road conditions, the environment uses Chrono soil contact method, or SCM.

Ultimately I hope to use this new found knowledge to further improve end-to-end learning within the Autonomous Research Testbed, improve upon our gym-chrono API, and further sim-to-real development as a whole. This serves as another step towards end-to-end learning on dART, advancing on Simone Benatti and Aaron Young's work training dART using Gym on cone data [3].

1.2 Overview

My research combines the existing Autonomy Research Testbed, or ART, and Gym Reinforcement Learning framework to produce a novel end-to-end learning environment. In this work, I will design an environment which mimicks reality and perform end-to-end learning techniques to teach the digital ART vehicle controls based on a given environment.

The ultimate goal of the vehicle is to drive between cones, without colliding into them. Eventually it will reach an unmarked destination at the end of the path, completing the simulation, or otherwise fail. This is an extension of the Autonomy Research Testbed's current abilities, which reads in cone data through our cone detection algorithm, and uses either a PID or MPC control agorith to choose the path forward. An example path can be seen in Fig. 1.1, where the vehicle is positioned mid way through a cone path.

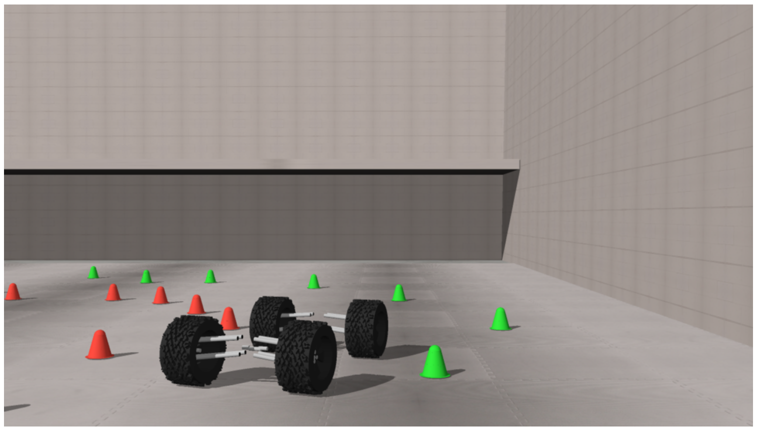


Figure 1.1: The ditigal Autonomy Research Testbed (dART) driving through a cone path using PID controls.

There are a few tools I will use that create the foundation of my research.

1. Chrono, which serves as the simulation engine [34, 32]

2. OpenAI Gym, which serves as a framework to connect simulation engines to reinforcement learning algorithms

3. Stable Baselines3, which houses reinforcement learning algorithms I will use to train my model [25, 7]

4. ART, which is the physical vehicle that will serve as the final testing to confirm our results

5. dART model, which represents a digital ART vehicle that will be driving in simulation, which has been calibrated to match ART

Put together, we will be able to build a more complete end-to-end learning simulation on the ART platform than before, helping close the sim to real gap [16, 17]. My major contributions will lie in connecting these elements by creating a Chrono environment, fitting it into the existing Gym framework, running the simulation, and then testing and comparing the results in a real life situation.

The existing state of the field is growing, but I believe my research makes meaningful contributions to it. Similar research has been done with Chrono in the Simulation-Based Engineering Lab, or SBEL, where dART ran in a Chrono simulation inside the Gym framework and was able to navigate around cones [2]. The results however were never carried over into the real world and the simulation was rather rudimentary, using privileged GPS information. I will be building on this work, creating an all new Chrono simulation and taking my results into the ART vehicle in real life to compare how similar or dissimilar the Chrono simulation is to real life.

1.3 Thesis Organization

The thesis is broken up into eight chapters, most of which describe one of the aforementioned tools: Chrono, OpenAI Gym, Stable Baselines3, ART, and dART.

Chapter 2 goes over important tools which I used but didn't significantly contribute to, such as Chrono and Docker. From there, I look at the overall structure of my research in 3, which goes into detail about many of the major components of my research. Next I talk about ART, dART, and it supporting software in Chapter 4 and Chapter 5, including how to use it or set it up yourself. I go over the machine learning processes used and developed for this project in the Chapter 6, including the cone detection algorithm and the reinforcement learning algorithm. In Chapter 7 and 8, my work and contributions are summarized and I consider where the largest potential contributions are for future research.

2 BACKGROUND

2.1 Chrono

Chrono is a multi-physics dynamics engine [34], which serves as the foundation and structure for the simulation component of my research. Chrono is written in C++ and version 8.0 is licensed under BSD-3. It contains a number of different soil, vehicle, and sensor models making it a powerful state-of-the-art simulation engine. I primarily interface with chrono using the PyChrono API [26].

2.2 PyChrono

Primarily throughout my research I'll be writing code in PyChrono, instead of using C/C++ to write for Chrono.

PyChrono itself is not a reimplementation of Chrono, but rather directly interfaces with the C++ API of Chrono. This is done using a tool called SWIG, which uses a YACC parser to read in the C++ code and generate new python API's and create a document module [1]. Currently, there are options to install PyChrono through conda or build it from source through `make` [26].

2.3 Chrono Terrain Model

Chrono has a variety of terrain models we can choose from, ranging from a fully rigid terrain to a granular terrain. Generally more accurate terrains are more computationally complex, and require more time to solve. Due to the use of reinforcement learning, I had to take computational time into consideration while running the simulation. Each simulation will have to be run over and over as we train the model, which may take a considerable amount of time. Ultimately, Soil Contact Method, or SCM, should give us a similar amount of accuracy compared to granular without taking as many resources to compute. We can see an example of what SCM should look like in this image from the Project Chrono website. In Fig. 2.1 you can see the tractor tire deforming the mesh structure, which represents the soil.

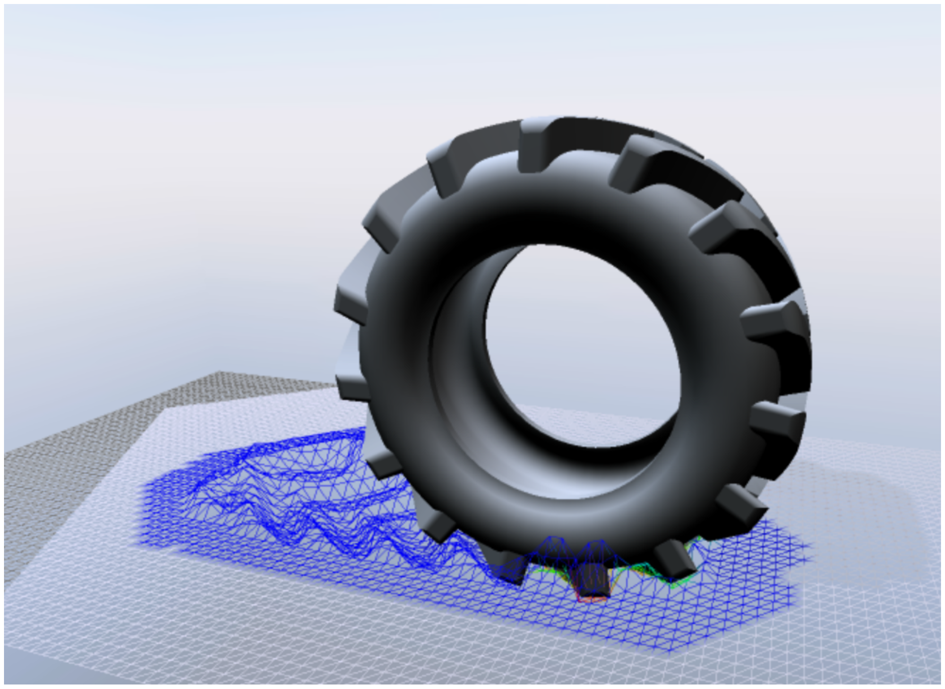


Figure 2.1: Here you can see the ruts in the soil match the groves on the tire.

In SCM the soil acts like a foundational spring, and by driving over the soil we apply pressure causing deformation. This does however mean you cannot deform the soil horizontal to the terrain, rather you can only apply vertical pressures. The relationship is defined as such:

$$\sigma = \left(\frac{k_c}{b} + k_\phi\right) y^n,$$

where σ is the contact patch pressure, y is the wheel sinkage, k_c represents the cohesive effect of the soil, k_ϕ represents soil stiffness, and n represents the exponential relationship that describes the hardening effect as the soil is pressed down [35]. Continuing with the generalized Bekker model we need to determine where the tire contacts the terrain. To approximate the footprint we calculate the contact tire width b, which is then approximately equal to the area of the contact patch, A, and the perimeter L, giving us the equation below.

$$b \approx \frac{2A}{L}$$

Using this information, we can begin to calculate how the vehicle interacts with the ground. Friction force information is exchanged between the vehicle and terrain model, and tangential forces may be calculated depending on the contact method used. These will be used to determine the vehicle's acceleration or deceleration, which is used to calculate state information.

2.4 Chrono Vehicle Model

In Chrono, vehicles are modeled using the Chrono::Vehicle module. The goal of Chrono::Vehicle is to provide an easy way for users to simulate wheeled or tracked vehicles while seamlessly integrating into the rest of Chrono. This process is made possible through a set of API's and Templates, so users can easily create new vehicle models or utilize the built in vehicle models [32].

The digital ART, or dART, vehicle is a crucial component of this research which allows researchers to effectively test control algorithms in simulation before testing them on the real world vehicle. In Chrono, dART exists as the RCCar model and received important updates to its powertrain model in Chrono version 8. Although the model is called RCCar in Chrono::Vehicle, I will be referring to it as dART for the duration of the paper for continuity with the ART platform. It's an extension of the ChWheeledVehicle class which is based off of ChVehicle, and can be called directly in Chrono and PyChrono.

2.4.1 Overview of dART

When building a new vehicle, there are a number of templates that Chrono provides which can be calibrated to produce a new vehicle. Chrono vehicle models can closely resemble those in real life, including suspension, steering, driveline, wheels, and brakes. We have attempted to do so, matching dART as closely as possible to the real ART vehicle.

2.4.2 Suspension

The dART vehicle model uses a double wishbone suspension in the front and rear, as does the real vehicle.

A double wishbone suspension is characterized by a coil spring mounted to two wishbones to control for vertical wheel movement, and are independent for each of the four wheels. An example of what this looks like can be seen in Fig. 2.2.

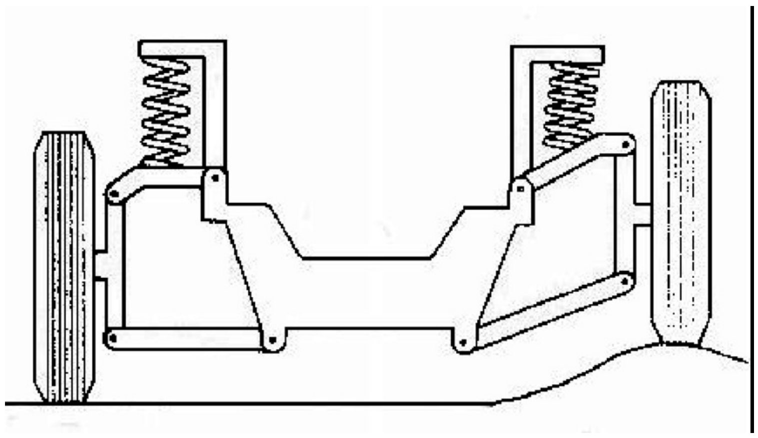


Figure 2.2: Example of double wishbone suspension, image from citation [33]

The double wishbone ensures the movement is only vertical, while the spring at the top compresses or lengthens as the terrain forces the wheel up or down. It's quite common to see this form of suspension on full sized vehicles, and not just smaller vehicles. Chrono implements double wishbone as a template.

2.4.3 Steering

ART uses a Pitman arm steering in the front wheels to steer the vehicle, matching the steering used in the ART vehicle. Chrono::Vehicle already has a Pitman arm submodule implemented as the ChPitmanArm class.

The Pitman arm steering mechanism is a four bar linkage, where the chassis sits in the middle as a rigid body and the Pitman arm is used to manipulate the steering link, which pushes additional linkages to turn the tires. A diagram of what this looks like in the model can be seen in Fig. 2.3, while a more abstract but clear image of where the joints exist can be seen here Fig. 2.4.

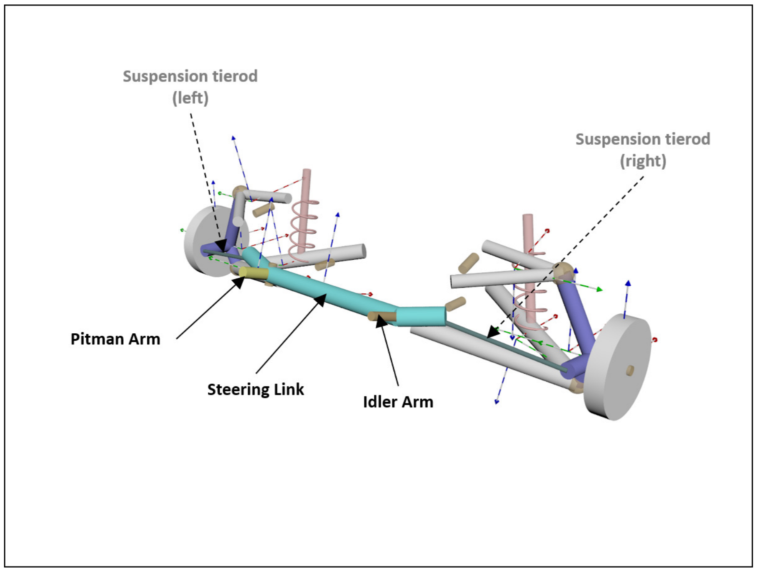


Figure 2.3: A diagram of what the Pitman arm motion looks like, from Project Chrono's page on wheeled steering.

As is the case in the real ART vehicle, the rear wheels are fixed and have no steering mechanism.

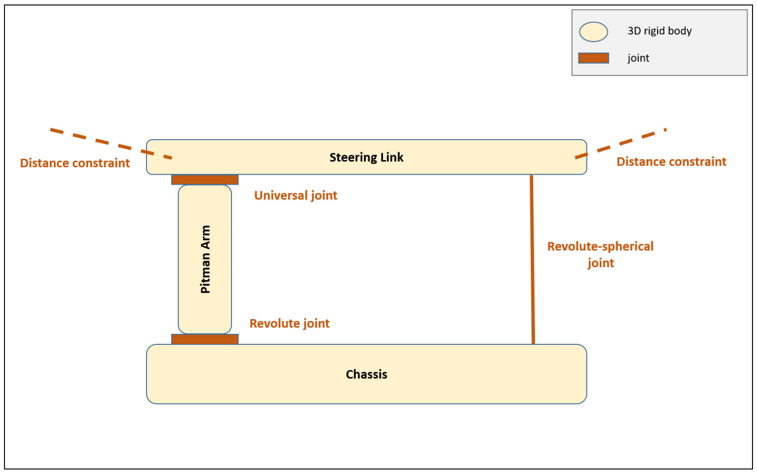


Figure 2.4: A diagram of what the Pitman arm topology looks like, from Project Chrono's page on wheeled steering.

2.4.4 Drivetrain

In Chrono, the Drivetrain is the collection of components that deliver power to the vehicle's wheels. It can be thought of as both the powertrain and driveline, and is used to model the power transfer from the engine to the wheels. There are templates for the driveline baked into Chrono::Vehicle, which are used to model the ART vehicle in dART. Chrono offers a few different driveline models, including a four-wheel shafts-based drivetrain, two-wheel shafts-based drivetrain, four wheel kinematic drivetrain, and an X-wheel kinematic drivetrain.

The dART vehicle uses a four-wheel kinematic drivetrain, based on the simple driveline model. This was done as it most accurately models the electric motor used in the ART vehicle. The vehicle has a 4WD system, and the electric motor doesn't use a complex transmission.

2.4.5 Tire Models

Chrono offers a few solutions for tires, including TMeasy and Rigid tires. TMeasy tire model attempts maintain a balance between being an easy to use tire model while also calculating complex features, such as a 3-dimensional slip and parametrization of the tire [15]. dART is able to take advantage of either method, although since this research focuses on the use of

SCM Rigid tires are used for all of the simulations. Rigid tires are the simplest model Chrono offers, and are completely rigid when interacting with the ground or other rigid objects. This means it is faster to compute contact points and friction than the TMeasy tire model, while also working well with SCM.

2.4.6 Brakes

Since brakes aren't used on the real ART vehicle, they were not implemented in the digital version either. Although the underlying vehicle platform does allow for a negative voltage to be fed into the motor to slow and reverse its direction, this isn't something the ART software stack implements.

Instead the vehicle stops due to the large amount of resistance through the motor and drivetrain, which is modeled in the Throttle Calibration section below. This resistance is substantial given the weight of the vehicle, and the research doesn't encounter any issues with the vehicle rolling.

2.4.7 Throttle Calibration

One recent change made for the dART model was to improve the throttle calibration. This is done by running a series of experiments on the real vehicle to learn how it responds to different vehicle inputs. From there the experiment data is mapped to a throttle posterior torque and loss curves for the real vehicle, and then the posterior torque and loss curves for the real vehicle are converted into parameters and mapped to the dART model. These parameters are passed into the dART `Chrono::vehicle` model through the functions `SetMaxMotorVoltageRatio`, `SetStallTorque`, and `SetTireRollingResistance`.

The torque and loss curves are calculated into model parameters using a Bayesian inference approach, which uses an assumed prior and the torque and loss curves to calculate a posterior distribution, which contains our model parameters [36, 37]. Here the Bayesian inference is used to solve for a set of unknown parameters in the model, and this works well due to

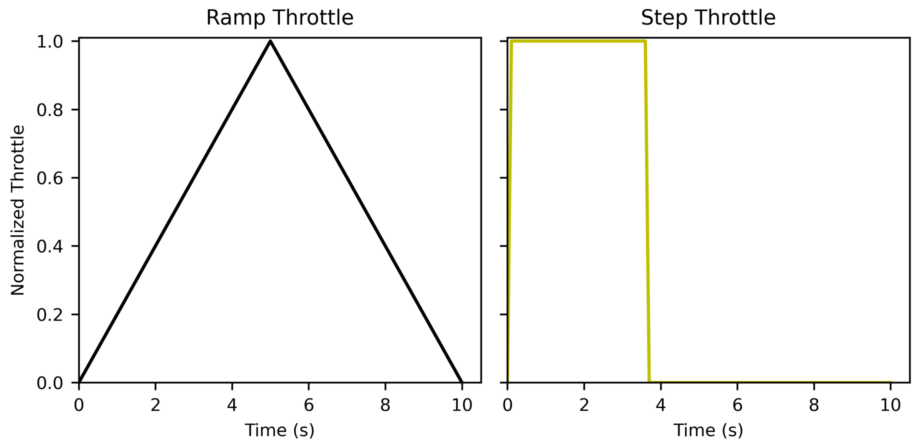


Figure 2.5: The ramp and throttle tests performed by the vehicle [37].

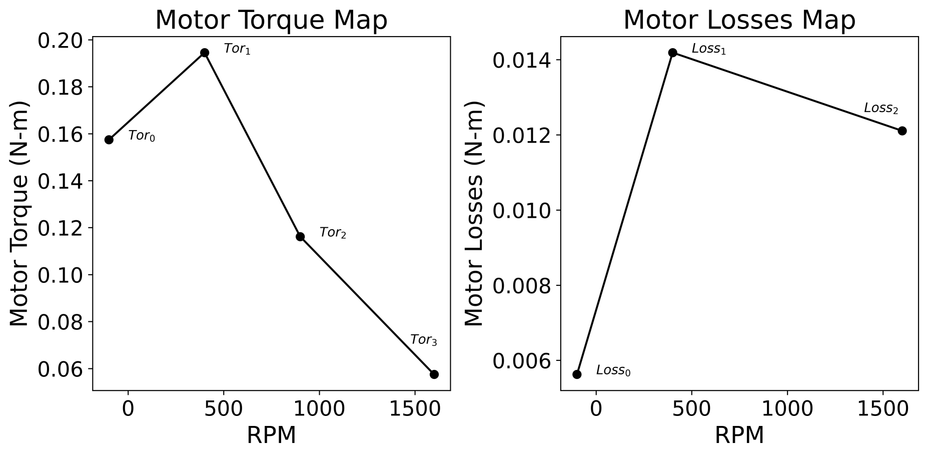


Figure 2.6: The mean posterior torque and loss curves [37]

its ability to quickly solve while being able to handle noise in the system [19]. The goal of Bayesian inference is to estimate the posterior distribution, represented below by $p(\theta|y)$.

$$y = \mathcal{G}(\theta) + \epsilon$$

$$p(\theta|y) \propto exp\left(-\frac{1}{2}||y - \mathcal{G}(\theta)||_{\Gamma}^{2}\right),$$

where θ is the known distribution of values, which we generate by performing tests, while $\mathcal{G} : \mathcal{X} \rightarrow \mathcal{Y}$ represents an unknown model and ϵ represents the system noise $\epsilon \sim N(0, \Gamma)$ [36]. To run the tests we focused on gradual increases and decreases in throttle (ramp) and sharp increases or decreases in throttle (step), which are visible in Fig.2.5. A set of simple and repeatable throttle and steering inputs were submitted to the vehicle, which were run 10 times in a row. From here the data was averaged to find θ. Using the model parameters θ, it is possible to calculate the throttle torque and loss curves, which can be seen in Fig. 2.6. Accurate state data was captured using a Motion Capture system, which is able to capture the vehicles position in real life with millimeter precision.

Ultimately the calibration parameters must be passed into the model during setup, as shown below.

```
import pychrono as chrono
import pychrono.vehicle as veh
...
vehicle = veh.RCCar()
...
vehicle.SetMaxMotorVoltageRatio(0.16)
vehicle.SetStallTorque(0.3)
vehicle.SetTireRollingResistance(0.06)
```

This allows the user to carry over the throttle calibration to other models if we so choose, or make adjustments on the fly.

2.5 Robot Operating System

Robot Operating System, or ROS, is an open-source framework which provides a flexible environment for connecting different software packages, primarily used in robotic systems. The ART platform uses ROS2, however many of these features are the same going between

ROS1 and ROS2. It contains a standardized set of communication protocols, tools, and other features which facilitate the development and control of robotic systems. In particular, ROS defines the publish and subscription models used when different parts of the system talk to each other.

There are a number of components one must understand to how ROS projects work. Developers will create nodes, which are software packages that perform a specific function (such as calculating the controls of a vehicle, or sending throttle commands to the ESC and motor). These nodes will communicate using topics, which they will either publish to (i.e. send data out on) or subscribe to (i.e. listen for data). In ROS2, these topics communicate in a decentralized fashion, using a publish-subscribe mechanism called Data Distribution Service, or DDS. DDS allows nodes to discover each other dynamically and exchange messages, without relying on a user or central master. Ultimately this allows complex projects to be built with many running parts that all communicate with each other in an efficient manner.

ROS has more features than just a publish-subscribe mechanism. A number of packages also exist which aid in building robotic systems, three of which are mentioned below.

1. **colcon build** The build system used in ROS2, discussed briefly in Section 4.5.1.

2. **ros2 bag** Sometimes referred to as rosbag or ros2bag, this package allows for topic data to be collected and stored, to either be played back or analyzed later. More can be read about its use with the ART platform in Section 4.5.5.

3. **ros2 launch** ROS offers the option for launch files, which were revamped in ROS2. These launch files tell ROS which nodes to spin up or can run commands, such as ros2 bags. This simplifies the process of performing functions in a ROS project, such as starting the vehicle or collecting data.

In many cases, how the ART platform uses ROS2 is specified in Chapter 4, or more information can be found in the ROS2 docs.

2.6 Docker

A major feature of ATK is its use of containerization, specifically through its implementation of Docker. Docker's main advantages are that it can consistently bring up the same environment over and over on multiple compute platforms, which reduces the likelihood of errors by maintaining the same environment and version numbers for the software being used.

The Docker platform has a few different important keywords to be aware of, including the container, image, network, and volume. The Docker image is a read-only set of instructions which will set up the environment you run in, often this is an operating system, and in ATK we use Ubuntu 20.04. Although a base template is often pulled from the internet, they can be modified before startup and will have a unique image ID on your machine. A container is the environment the image sets up for use, and must be spun up and down. It's assigned a unique container ID, and runs whatever image was given to it. Since the image is read-only, on startup you can mount a volume to your container, which will allow you to read and write local files which will persist even if you stop the container. In this case we mount the ATK git repository in the container, so any ros2bags or edits you make will persist into and out of the container.

2.7 Docker Compose

Docker Compose is a separate library which can be used to make modifications to docker images when starting up the container. For our Autonomy Stack, we use Docker Compose as the backend for our ATK config file [27].

Docker Compose is written as a yml file, which is another way of writing JSON, without the brackets. From here it's broken up into the default containers and services which can be enabled or disabled. We use all of these to define the containers in ATK and set which feature sets must be enabled when the container is spun up.

2.8 PyTorch

PyTorch is an open-source machine learning framework developed by Facebook based on the Torch library. The ultimate goal is to provide an efficient method for training deep learning models. Similar to Google's TensorFlow, PyTorch implements a python wrapper over Torch, which is a Lua library containing different machine learning algorithms which was originally designed to provide three main advantages: ease of use for developing numerical algorithms, be easily extendable, and fast. Many of the computationally intensive algorithms are implemented in OpenMP/SSE and CUDA [6, 23].

Ultimately ART uses PyTorch for a few reasons. One is that it is a large, popular machine learning framework making development easier and reducing development time. Second is that it's tightly integrated with ONNX, which is a framework for pre-built machine learning models. These models allow ART to easily apply transfer learning to existing image models, and only train the final few layers, producing more accurate results than we could ever produce alone. Lastly its popularity has led to its use in other libraries, most notably Stable Baselines3 which was used to train the reinforcement learning agent.

3 SYSTEM OVERVIEW

My research is heavily based upon or a part of a few platforms; including ART, Chrono, and Gym. Because of this, it seems necessary to go over what my research is and what it isn't, as well as what each of these tools or platforms are used for.

3.1 Real World ART

Everything in my research is directly based on real world vehicles and dynamics. Because of this, we spent a lot of time calibrating dART to match the real world vehicle, so its actions in simulation closely match those in reality.

3.1.1 ART

The Autonomy Research Testbed, as pictured in Fig. 3.1, is a vehicle and software stack which the Simulations Based Engineering Lab (SBEL) developed and uses to test control algorithms and sensors. It serves as the bridge between real world and Chrono simulations, as it has a digital twin, often referred to as dART, which is calibrated to operate as similarly to ART as possible.

Recently we have used this to test control policies such as MPC and test a Bayesian inference calibrating technique. In my research however ART served as the baseline for what dART should be, and all of the actions the vehicle takes in the end-to-end learning were based on real calibrations of ART.

3.1.2 ARCLab Motion Capture

Motion capture was also an important tool, which I go into detail with in Chapter 4.3. We performed a lot of tests in the ARCLab, where we can achieve millimeter precision on ARTs position, similar to what can be seen in simulation.

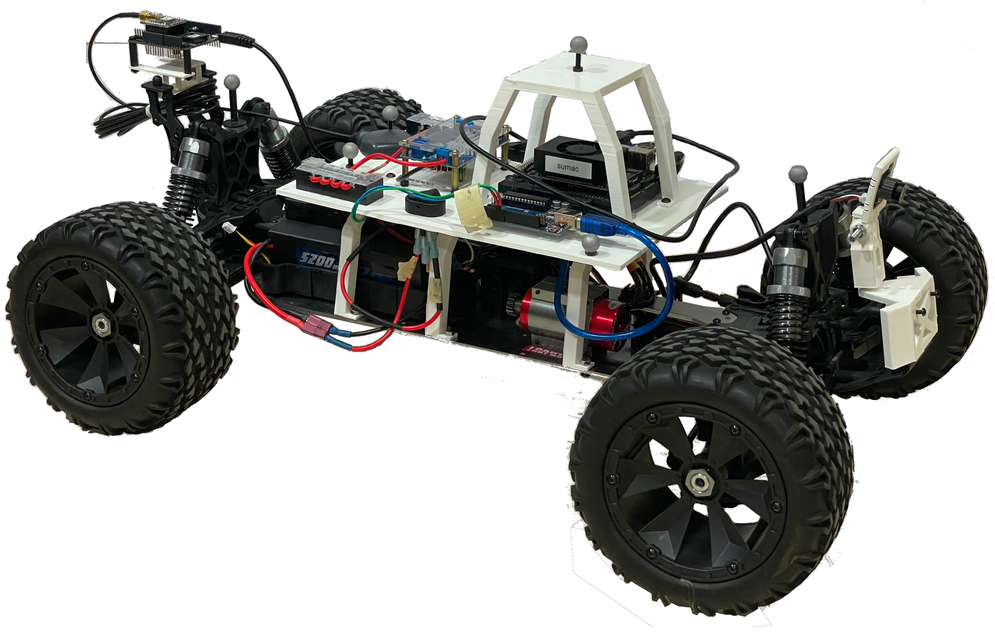


Figure 3.1: Shown here is a picture of the original ART vehicle. The white bumper indicates the front of the vehicle, with a GPS/IMU sensor hanging off the rear suspension.

3.2 Simulated Environment

3.2.1 Chrono and PyChrono

As mentioned in Chapter 2.1, Chrono is the simulation engine written in C/C++ which can be used to run simulations, and is often used in the SBEL lab to perform research aimed at closing the sim-to-real gap. PyChrono, as discussed in Chapter 2.2 is a python API generated by SWIG.

3.2.2 dART

The digital autonomy research testbed, or dART, is the digital twin of ART which I used in my testing to simulate ART. It's represented in Chrono as `RCCar`, as one of the vehicle packages built on top of ChVehicle, which can be read about in Chapter 2.4.

One of the advantages of the platform is it's easy to switch between running in dART and ART. As you can see in the high level diagram, Fig. 3.2, the autonomy stack and algorithms interface the same way, regardless of whether you're running in simulation (dART) or a test in real life (ART).

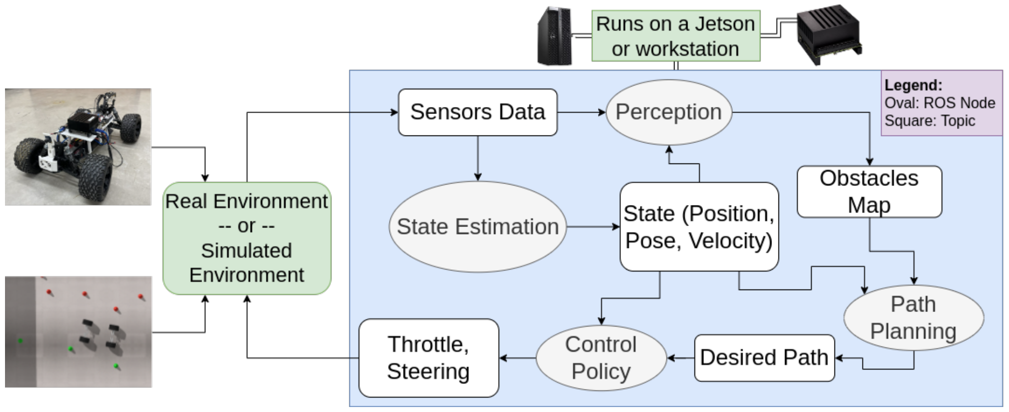


Figure 3.2: An overview of the ART platform, where we can see the interface of the Autonomy Stack and its associated algorithms, with the simulation or real vehicle.

Implementing the throttle and steering calibrations on the RCCar model can be read about here in Chapter 2.4.7.

3.2.3 Gym

Gym is a framework used for reinforcement learning, and its major advantage is simplifying environments into a common framework to make it easier to apply reinforcement learning algorithms to them. I discuss this more in Chapter 6.1.1

3.2.4 Stable Baselines3

Stable Baselines3 is a library containing a number of different reinforcement learning algorithms, including Proximal Policy Optimization, or PPO, which is what was ultimately used. You can read more about it in Chapter 6.1.4.

4 AUTONOMY RESEARCH TESTBED

The Autonomy Research Testbed, or ART, is a fully autonomous capable vehicle built to perform testing in real life which we can compare to the version in simulation, referred to as dART. Using both ART and its digital twin, dART, users can directly compare autonomy algorithms in both real life and simulation [12]. ART is based off of the 1/6 sized model RedCat Racing Shredder, and is powered by a Jetson Xavier AGX and a 1300Kv battery. The chassis uses an Ackermann steering model [39] which allows the wheels to pitch differently as the vehicle turns, and the Jetson is able to perform much more powerful machine learning algorithms due to its access to Nvidia CUDA and other nvidia libraries than a more traditional Raspberry Pi.

Our lab has two separate vehicles for testing, which we will sometimes refer to as ART1 or ART2. Functionally, however, these vehicles are identical save for the colors of some of the printed plastics. To prevent potential issues when testing however data will always be collected with either ART1 or ART2.

4.1 Autonomy Research Testbed Hardware Components

In this research, the ART vehicle has been divided into four major components: power distribution, vehicle dynamics, chassis, and sensors.

4.1.1 Power Distribution

The platform is powered by two batteries which contain a shared ground and two separate power rails, one to power the motor and one to power the Jetson.

4.1.1.1 Motor Power

The power distribution has been modified from the original vehicle. Originally the ESC was designed to take up to two 7.4V LiPo batteries connected in series (four cells), however this produced a remarkably fast vehicle which was beyond the abilities of the perception algorithms on the current hardware to handle.

We use a single two cell battery connected to the ESC, which receives its inputs from an Arduino UNO. An SPST rocker switch sits between the battery and the ESC as a manual method to control whether power is being supplied to the motor. The Arduino takes in a zero to one throttle value from the /control/vehicle_inputs topic, and maps that across its minimum and maximum PWM values. From where it sits on this range it is linearly mapped out and sent to the vehicle, where the maximum PWM signal is lower than the theoretical max of the vehicle. This is fed to a 1300Kv brushless motor which is then geared to all four wheels.

4.1.1.2 Jetson Power

The power for the Jetson was developed separately from the RC car since the original 2.4GHz radio controller had considerably lower power requirements. Due to the ARM based chip on the Jetson, we are able to get away with much lower power requirements than a comparable x86 chip, and the entire Jetson is powered by an 11 volt barrel connector.

Our solution starts with a three or four cell LiPo battery, which sits below the component board. It's connected to a SPST rocker switch which connects to a DC buck converter [8], which can be seen in Fig 4.1, downvolts the input to a constant 11 volts while providing better power efficiency than a linear regulator [18]. This can then be fed into the Jetson safely within its tolerance band, and can reach its max clock speed of 12v to 19v. A four cell battery is recommended for input, however one can run at lower voltages without issue as well.

Lastly the Arduino and other sensors are powered off the Jetson USB ports, which creates a problem as the USB port maintains a virtual ground which is slightly different from the ground fed into the vehicle. To resolve this we link the ground of the Arduino back to the virtual ground of the Jetson, setting the two equal, which is visible in Fig 4.2.

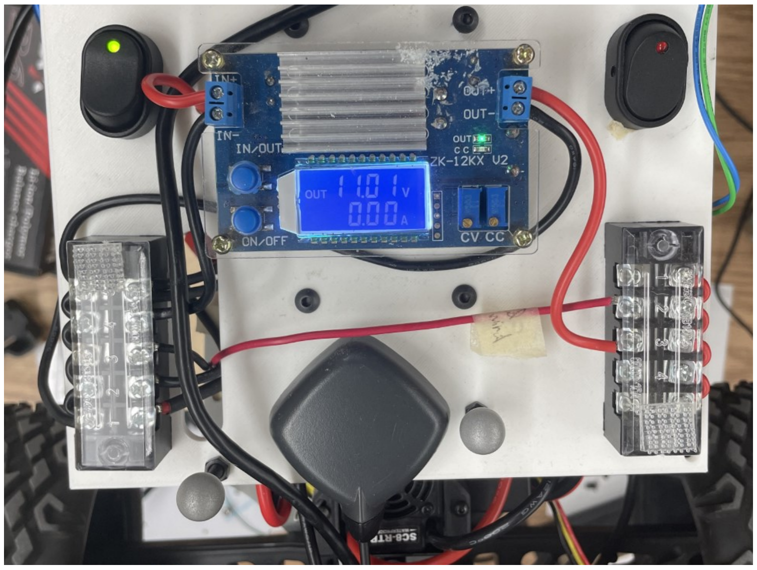


Figure 4.1: Image of the buck converter on ART1. It's displaying the output voltage of 11 volts, and the Jetson power rocker switch is on (green) while the motor rocker switch (red) is off. Unmentioned are the power distribution blocks (right and left), which just serve to connect wires in parallel to each other.

4.1.1.3 High Level Electrical Diagram

Although the platform is extendable, there exists a necessary foundation that must set up for the platform to work.

In Fig 4.2 we can see what the power distribution looks like for essential systems. The IMU/gnss and camera are not included, however they're each powered by the USB connection which also transmits data. Everything included in the diagram is required.

Notably the 5V bus is not powered by any individual battery. Rather it is based off of the 5v pin on the Arduino Uno. We additionally must connect one of the Arduino ground

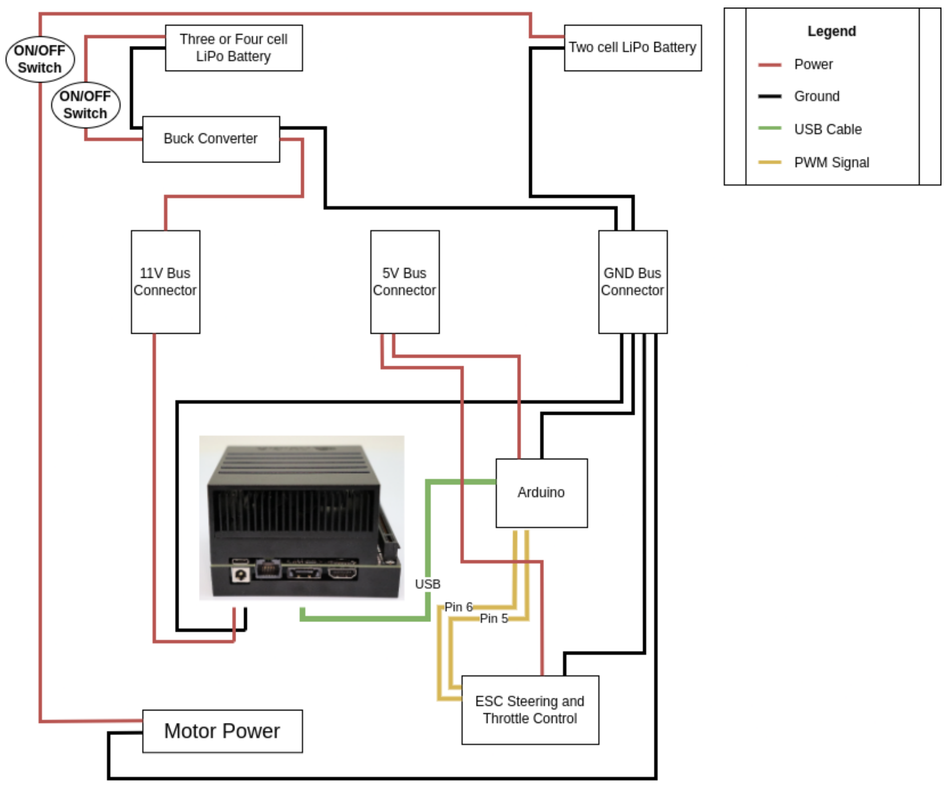


Figure 4.2: A high level diagram of the power distribution for the system. We have three buses, one for 11v, 5v, and gnd. These distribute power to each of the sensors and components and all share a common ground.

connectors to the global gnd bus, otherwise one can experience odd behavior when applying a low pwm signal to the vehicle.

The ESC Steering and throttle control are based on the included ESC from the Red Cat Racing Shredder vehicle. We remove the radio transmitter connections and replace them with the Arduino connection, using the shared ground and 5v power. Pin 5 is used to drive the steering, and has a direct connection through the ESC to the servo motor which turns the vehicle. Pin 6 connects to the ESC to deliver the pwm used for throttle control. Both of these PWM values are bounded by the Arduino from 1000μs to 1980μs, with 1500μs being neutral. The Arduino driver code can be found in the autonomy research testbed codebase, alongside the ROS packages.

4.1.2 Vehicle Steering

The ART vehicle is based off of the Red Cat Racing Shredder and as such it follows their steering and tire setup. Notably, it uses a Pitman arm steering, which is the same style of steering used in most cars. This is shown in Fig. 4.3, however a more detailed overview of ackerman steering exists in Fig. 2.3, where its implementation in Chrono and the dART vehicle is also discussed in the section 2.4.3.

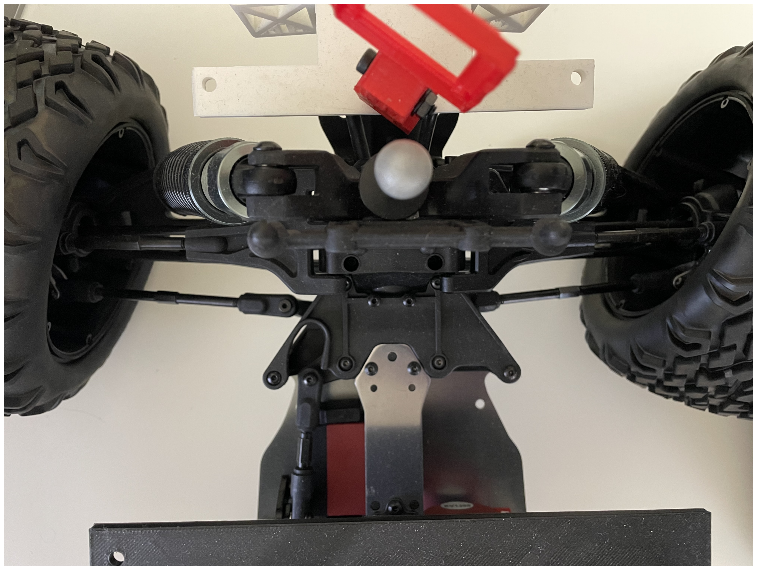


Figure 4.3: In image of the front of the ART2 vehicle, where one can see the linkages that make up the pitman arm steering.

4.1.3 Chassis

Most of the raised chassis for the vehicle is 3D printed, allowing for all the mounting points for components and sensors. All other components are from the 1/6th scale RedCat Racing Shredder. We used a 30% infill for most of the components, while resorting to an 80% infill for the top platform, since it has to carry weight.

We used Solid Works to design the models, which we then exported to an STL (standard triangle language) model so it could be printed on 3D printers. There are 4 base components (three arches to support the component board and a component board) and 4 sensor components (bumper, two models for supporting the camera, and an IMU board). Additionally there is a LIDAR support however this wasn't used in simulation model or real life.

4.2 Sensors

There are a number of sensors which have been implemented into the autonomy stack, and any sensor could be implemented by adding a ROS node and proper communications with the Jetson. Currently, we've done a lot of work with camera, IMU/GPS, and motion capture for testing and calibration of the vehicle. That said, we focused on a front-facing camera in my research.

4.2.1 Camera

We used a 1080p fisheye USB camera, referred to as FHD 1080P. This camera connects to the Jetson through a USB connection, and is accessible in the autonomy stack as /dev/video0. Further we use two different encoding options for the camera: mjpeg and yuvu. mjpeg seems to have better hardware support on the Jetson, however the yuvu video encoding allows for a faster frame rate (up to 30fps).

Notably what we've found is that the performance of the camera is heavily impacted by the performance of the Jetson. Even when setting the camera to higher frame rates, we find that it's unable to maintain that high frame rate under load from the Open Neural Network Exchange (ONNX) neural network we use for the cone detection. Unfortunately we're currently tied to using this camera since the cone detection model was trained on its images, however in the future it would be exciting to attempt to train a more generalized cone detection algorithm to better handle different cameras and lenses.

Using the camera mount we printed, the camera orientation can rotate around a center axis and tilts forward and back, and is held in place using the tension of the screws holding the piece together. Finally the mount was designed to fit the camera exactly, and clips in without screws or glue.

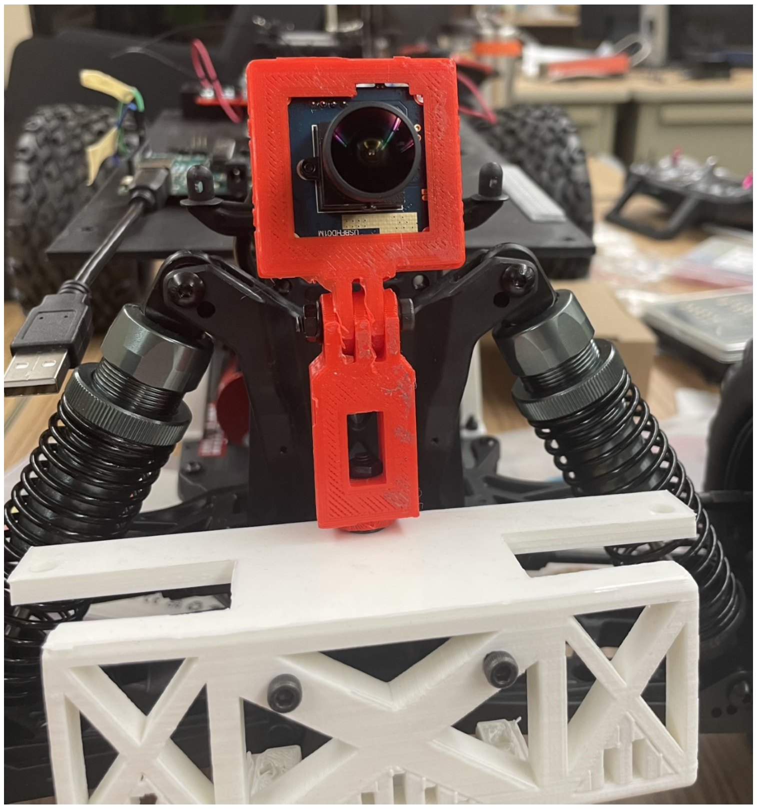


Figure 4.4: Here one can see the camera in its mount. The two red mounts allow for tilt forward and back, while the bottom mount and bumper allow for swivel. This all sits on the front of the vehicle, ahead of the suspension and wheels.

4.2.2 Xsens MTi-7 GNSS/INS

The Xsens MTi-7 is a combination GNSS, magnetometer, and IMU sensor. It has two outputs, one to the GNSS receiver which sits on the component board, and one to the Jetson which it connects through a USB connection. Once connected, we use a provided ROS node to convert the output into topics which can be fed into the autonomy stack.

Notably for the ART platform, we do not use the onboard filtering mechanisms in the MTi-7 since we cannot recreate them exactly in simulation. Because of this, there are a number of the onboard Kalman Filters used in the MTi-7 which are ignored, and the ART platform provides its own filters to perform the computations. This means we ignore all the topics output as `/filter/*` and calculated values such as velocity.

The MTi-7 is mounted in the rear of the vehicle, slightly behind the rear axle. There are no rubber dampers in between the IMU and vehicle, rather it is mounted using three plastic screws onto a 3D printed mount. The GPS receiver is connected to the component board using 3M Dual Lock tape, so it can be easily removed.

4.3 Motion Capture and Retroreflective Markers

One major advantage we had when testing the ART vehicle is the ability to use a Motion Capture lab, which contained 13 OptiTrack cameras that were capable of high precision location detection while updating the vehicle at 100Hz. When launching in the Motion Capture lab, we ran a different launch file, titled `art_reality_lab.launch.py` which would spin up the official mocap optitrack ROS node [29] for the foxy version of ubuntu. From here we were able to interact with the exact position data, which is usually just to ros2bag the output along with the rest of the system information. In other cases, such as MPC testing, we used the motion capture data to get velocity, and at other times we used the motion capture system to simulate GPS and compare that against real velocity recordings.

OptiTrack motion capture functionality works by tracking retroreflective markers on an object, in this case vehicle, across multiple camera angles. Two of these markers can be seen in Fig 4.6 These are then reconstructed to determine the position of an object visually [14]. The placement of these markers are important, however, so there exists a number of holes on the component board for them to be placed, which can be seen in Fig 4.5. Notably they shouldn't be symmetric, and should vary as close to random as possible with each other. Additionally the more points there are mean the user will generally get better results, which is why we use six to seven, depending on the setup. The holes on the board should be the right size to allow the markers to be placed anywhere. As seen in Fig 4.6 the markers are placed at different heights using M2 screws, to further aid in increasing accuracy. They're held steady using the pressure between the screw head and a washer and nut.

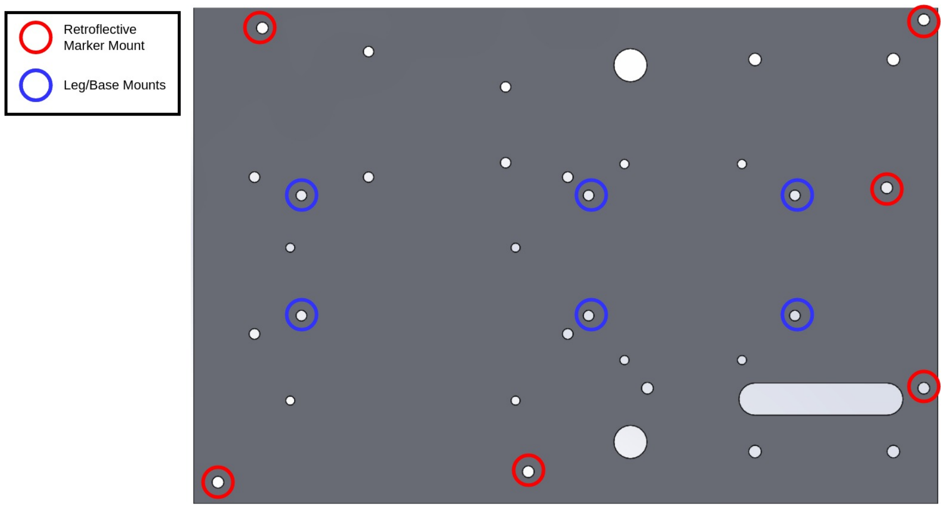


Figure 4.5: Here we can see the component board with labels for the different holes, in this case being four we used for the retroreflective marker placement. Notably missing are the two markers we attached to the front and rear suspension of the vehicle, and the marker which can sit on top of the lidar mount.

Placement of the markers shouldn't be taken too seriously though, as again we want to emphasize randomness increases their accuracy. Because of this we placed them at unspecified heights using M2 screws.

4.4 Onboard Computer

Due to the containerization of the autonomy stack, there exists a lot of flexibility in the onboard computer we could use. For ART, Jetson Xaviers are used in the setup, and the power distribution systems have been designed around them. The advantage with using Jetsons is that they're CUDA enabled, allowing us to take things like PyTorch off of the CPU and run them more efficiently [23]. Currently the Jetsons are affixed to the component board with 3M DualLock instead of screws, to make removing the computer for testing easier.

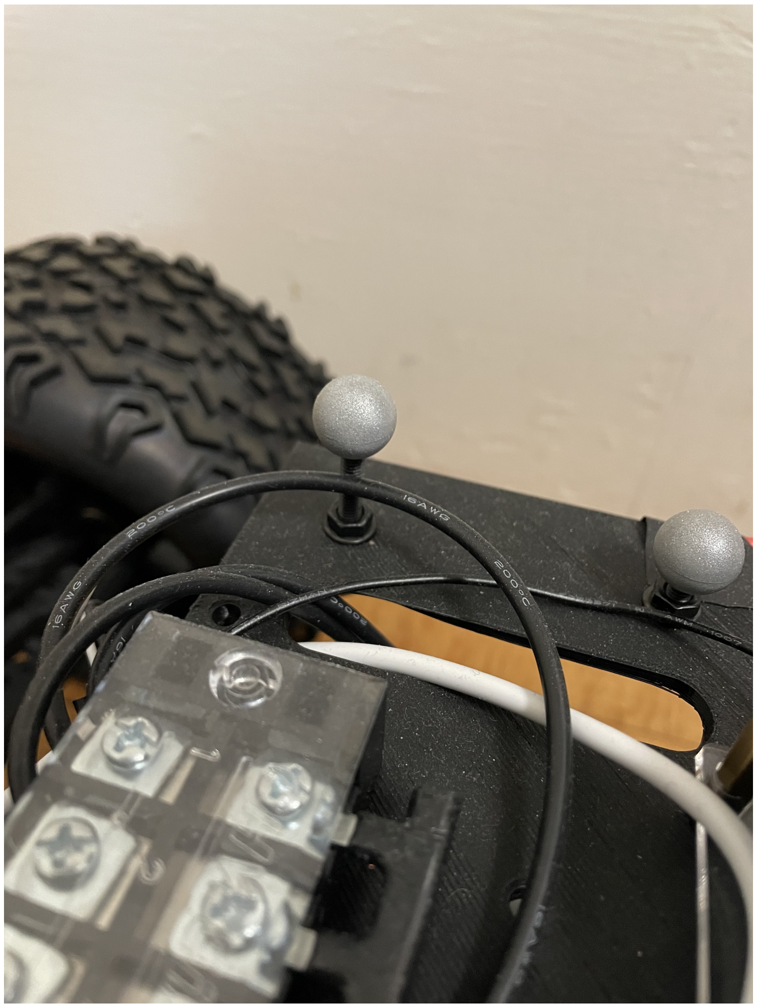


Figure 4.6: Here are two retroreflective markers which were placed on the second ART vehicle.

We've used two separate Xaviers over the course of the project, and will likely be moving to a third for one of the ART vehicles. Initially we started with a Jetson Xavier NX, however have moved up to a Jetson Xavier AGX as it became available. Soon we will be able to plug in and start testing a Jetson Xavier Orin. All Jetsons we've tested work well with ATK and are visible in fig 4.7, and we believe this to be true due to the containerization properties of ATK and similar hardware across boards.

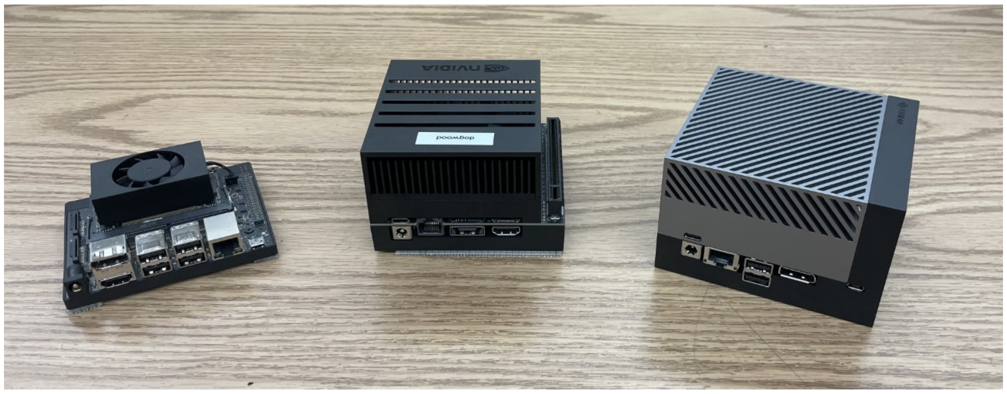


Figure 4.7: Here we can see, going from left to right, the Jetson Xavier NX, Jetson Xavier AGX, and the Jetson Xavier Orin. Each is powered by a variable output buck converter onboard the vehicle or the provided power supply which can be plugged into the wall.

4.5 Autonomy Research Testbed Software Components

4.5.1 Software Structure

ART has a large software frame in addition to the vehicle which runs all of the code. It was designed to be modular and extensible, just like ATK, and can take on a number of additional sensors and control algorithms if desired. Functionally the project is split up into three folders, as seen in Fig. 4.8.

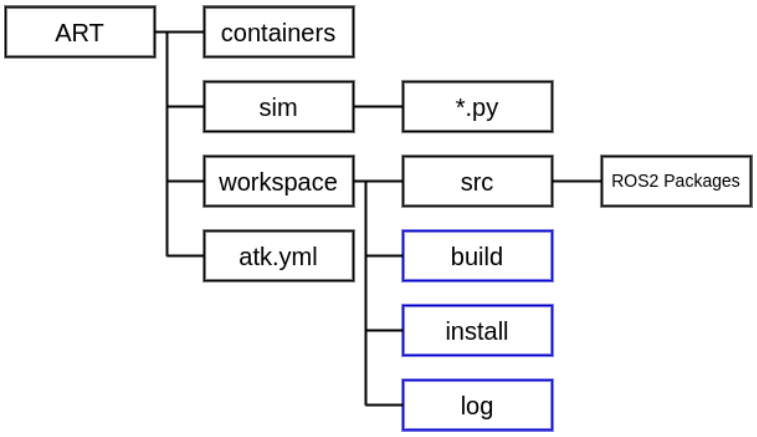


Figure 4.8: A brief overview of the folder structure, with ART representing the root folder. Notably the blue folders are only built after `colcon build` is run.

The `containers` folder is used by ATK to build the containers we use, and shouldn't need to be modified.

The `sim` folder is where any simulation will run. These simulations will only be run when not testing in reality, and are connected to ROS through the Chrono-ROS bridge [11].

Lastly the `workspace` folder contains all of the ROS2 code for the vehicle, and is where most of the changes will take place. It is set up like a normal ROS2 directory, with a `src` folder, and is build by running

```
$ colcon build
```

Alternatively, one can build with simulink, which will simply link to the python code in `src` instead of moving it into `install`. This will allow the user to continue to make edits to python code in `src` without having to rebuild every time. To use it, simply pass the simulink flag.

```
$ colcon build --symlink-install
```

Inside the 'src' folder one can find each of the ROS2 packages. Most of the launch files are located in the 'common' package and in subdirectories such as art_launch. Each of the items in Fig. 5.1 match their package name with their function.

Building will create `build`, `install`, and `log` folders. When building, one may rebuild over them, or to perform a clean build the user would have to delete these three folders before building again.

4.5.2 Setting up Environment

Whether on an ARM or x86 based computer, the process for setting up the Autonomy Research Testbed should be similar, due to the use of containerization. Much of the setup I'm about to explain should only have to be done once.

First one must install the Autonomy Toolkit, or ATK. More information about how this works can be read about in Chapter 5, however it can easily be installed using pip, as follows.

```
$ pip install autonomy-toolkit
```

One may instead want to install it in a virtual environment as well, to protect local installs. This is not necessary, however he's an example of how we do it.

```
$ mkdir venv
$ python -m venv venv
$ pip install autonomy-toolkit
```

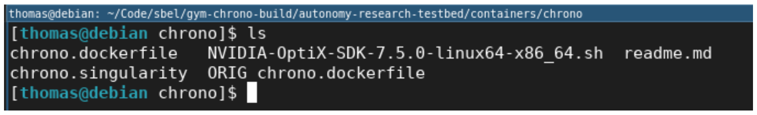
Next the user will have to clone the repo locally from the github repository and enter the repository.

```
$ git clone git@github.com:uwsbel/autonomy-research-testbed.git -b master
$ cd autonomy-research-testbed
```

From here the user will have to install or build a number of dependencies, which should only have to be done once. Many of the ROS2 packages are git submodules which must be pulled, and the YoloV5 cone detection network must be pulled from git large file store, or git lfs.

```
$ git submodule update --init
$ git lfs install
$ git lfs pull
```

Lastly to run the PyChrono container the user will need to place the Optix Version 7.5 in the correct folder. Unfortunately due to licensing restrictions we cannot ship the Optix installer in the repo, however it can be downloaded and placed in the Chrono folder. This exists in the cloned repo at `autonomy-research-testbed/containers/chrono` as pictured in 4.9. Downloading the installer will require an Nvidia account, and the end result should look as follows.



```
thomas@debian: ~/Code/sbel/gym-chrono-build/autonomy-research-testbed/containers/chrono
[thomas@debian chrono]$ ls
chrono.dockerfile   NVIDIA-OptiX-SDK-7.5.0-linux64-x86_64.sh  readme.md
chrono.singularity  ORIG_chrono.dockerfile
[thomas@debian chrono]$
```

Figure 4.9: Image of what my Optix7.5 install looks like. The version and name for the container must be the same to run properly, and the installer can only be downloaded from Nvidia's website.

Now it is possible to enter the container. The last thing which will have to do as part of setup is build one of the sensor ROS2 packages. This package in particular isn't built by colcon build, and the user must manually build the C++ binaries from their source. This can be done from inside the `dev` container. First enter the `dev` container as follows, and then build the C++ binaries in Bluespace AI package.

```
$ atk dev --custom-cli-args gpus
$ pushd src/bluespace_ai_xsens_ros_mti_driver/lib/xspublic && make && popd
```

It is also possible to go directly into the package `lib` folder and run make if that is found to be easier. After this however all the initial setup is done, it should be possible to freely start up and run simulations.

4.5.3 Running in Simulation

To run the ART Stack using Chrono, the simulation environment must be started independently of the ROS2 launch file. At this time there are a few options in the master branch, including `demo_ARCLAB_IROS.py` and `demo_ARCLAB_cones.py`. Each will work and each have a number of options written into the code, and running it should look like this:

```
$ python demo_ARCLAB_cone.py
```

Additionally the python file can be run without entering into the container. This may be done using the following ATK command:

```
$ atk dev --services chrono --custom-cli-args gpus --python demo_ARCLAB_cone.py
```

Some values that are worth keeping in mind is `step_size`, which will determine the accuracy of the simulation, where smaller is more accurate but more computationally intensive. Additionally `vis` should be set to `True`, and will allow the camera output to be seen through the `noVNC` VNC viewer.

After starting the PyChrono simulation, the software from the autonomy stack will be run in the separate `dev` container. The default version of this is the `art_simulation.launch.py` launch file, although it may have to be edited if the user wants to make changes to the sensor configuration.

Once inside the dev container, the launch file can be started with a command like this:

```
$ ros2 launch art_simulation_launch art_simulation.launch.py \
>   hostname:=art-chrono vis:=true
```

Where art-chrono is the name of the container, which is defined at the top of the `atk.yml` file and is visible as the host name for the Chrono container. Additionally `vis` is an optional flag which may be set to false, however we recommend running with it on so the user can see the Chrono visualizations in `http://localhost:8080/`.

If someone ever needs to build and maintain multiple containers on the same machine (such as a shared computer or multiple builds), they can change the hostname in the `atk.yml` file. This is done by changing the project name, which changes the host name of the container and the Docker container names so they won't conflict. In the example code from `atk.yml` below, the project name `art` can be set to something more unique.

```
project: art
default_containers:
  - dev
  - vnc
custom_cli_arguments:
```

4.5.4 Running in Reality

Running in reality is similar to using just the `dev` container in simulation, and that is by design. We wanted the ROS-to-Chrono bridge to look as much like real life as possible, and reuse as much of the code as we can. Further installation and setup on the Jetson will be same as on a workstation, and is listed in Section 4.5.2

Before running the user will need to make sure the vehicle is on and running.

To do this they will have to ensure both batteries are on and plugged in. Below the top platform two places for the batteries to fit should be available; one will be a plastic container for the three cell battery, and the four cell battery will sit between the drivetrain and support structures for the ART vehicle. Each battery should plug into a banana connector seen on each side of the vehicle.

From here two switches should be visible as seen in Fig. 4.10. The green switch turns on the Jetson, and the red switch turns on the motor. We've separated them both because they carry separate voltages, and because we wanted an independent, physical kill switch for the vehicle.

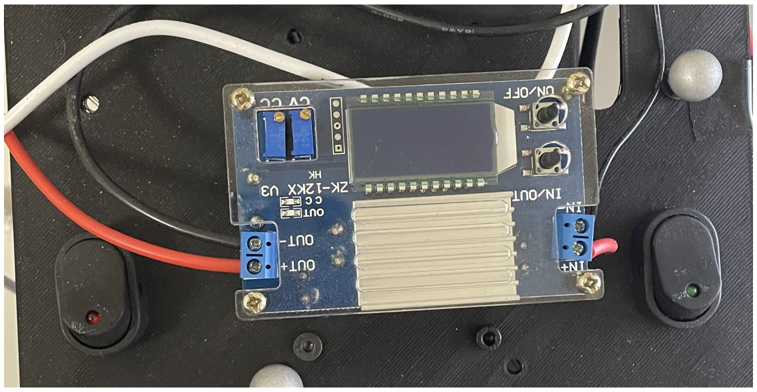


Figure 4.10: Here the two switches can be seen on the ART2 vehicle, as well as the buck converter in the middle. The green switch on the right controls the Jetson, and the red switch on the left controls the motors and steering. In the center is the buck converter.

After turning on the Jetson power using the green switch, the Jetson itself must be turned on, which can be done with the switch visible in Fig. 4.11. It will likely take a few minutes to start up. Once this has happened, the user will have to ssh into the Jetson to be able to control it, and there are a few methods of doing that listed here in Chapter 4.6.

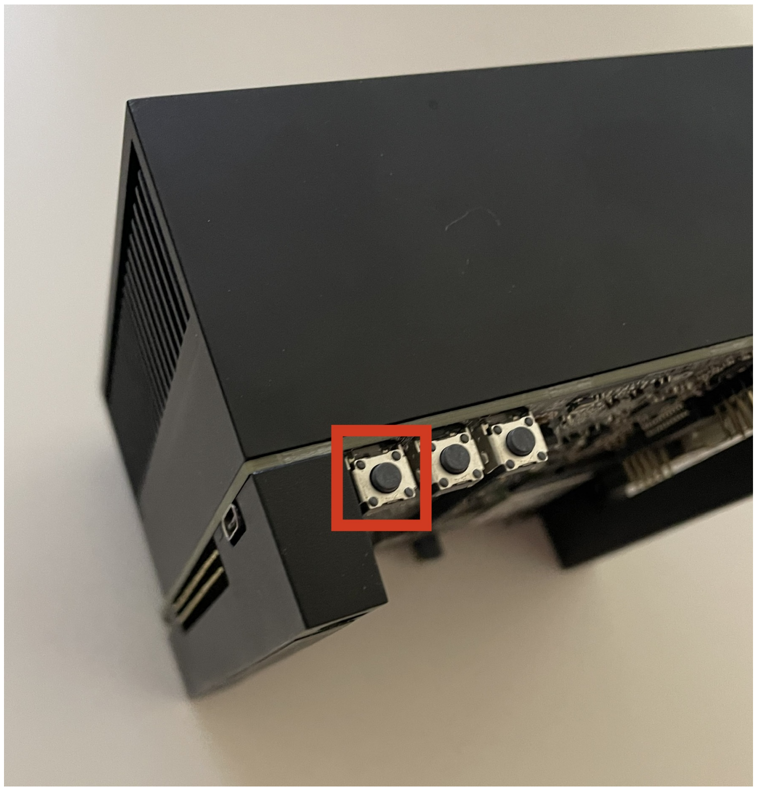


Figure 4.11: The power switch is boxed in red, and is the switch closest to the side. Do not press the other switches, as holding them down can cause a forced reset.

Once inside the Jetson, the dev container should be started using the nx service.

```
$ atk dev --service nx --custom-cli-args gpus
```

After this, turn on the motors by flipping the red switch. Both switch LED's should now be on.

Now the tests may be run. An example script for running a launch file is shown below.

```
$ ros2 launch art_launch art_reality.launch.py throttle_gain:=1.0
```

It should be noted `throttle_gain` is only used when using PID controls and isn't needed for MPC.

That should be it. The vehicle should be running and driving about.

4.5.5 Data Collection

Data collection using ART is relatively straightforward. We leverage ROS2 bag feature to create custom launch files which will save the data locally to the Jetson. From there we can copy the data off of the Jetson at a later date and extract it into CSV files.

A launch file that will build a ros2bag is available in the art_launch package. This ros2bag will record all topics currently being used when running in reality, titled `art_rosbag_reality.launch.py`, and may be edited as needed. The file can be ran as follows:

```
$ ros2 launch art_launch art_rosbag_reality.py
```

From here the data may be copied over to a machine and extracted in whichever fashion is preferred. It's worth noting that although ROS2 bags use a sqlite database, the data is encoded, so the user will need to decode it first. This may include importing some custom messages. To solve this problem the python package `ros2 bag` can be used to decode the data in a python file and write out to a CSV, which can be configured to use custom messages.

4.6 Accessing the Jetson

Functionally the Jetson is just like any other computer, so logging into it is a similar experience when compared to logging into any Ubuntu machine. Once the operating system and user account are set up, a user should be able to easily ssh in with the IP address. Finding the IP address will often require plugging the Jetson into a monitor, logging on, and getting the IP address using the linux terminal command `hostname -I`. The Jetson and machine being used to access the Jetson will have to be on the same network.

In situations where the vehicle is being used outdoors where no router is present, the Jetson itself can be used as a wireless hotspot. This is most simply done by connecting the

Jetson to a display, and entering the Wifi settings. From there, you will be able to construct a local wireless network which you can then connect another computer to. After doing so you can get the Jetson's IP address and connect through ssh.

5 AUTONOMY TOOLKIT

The Autonomy Toolkit is a tool developed in house to facilitate development and testing of ART in both real life and simulation. Functionally it's a wrapper of Docker Compose, and it's designed to bring up different services depending on whether we're running the ART vehicle or its digital twin in Chrono, dART [12]. The service is modular, however, and can not only be applied to different projects but exists as a python package which can easily be installed using pip [28]. A full walkthrough on configuring and running ART through ATK can be found in Chapter 4.5, however I will attempt to provide a more in depth explanation of ATK here.

During testing, there are three major services used which can be brought up by running two separate commands. The autonomy stack is brought up in the 'dev' container, and is the same for both real life and when running simulation. The Chrono service, however, is only started when running the digital twin of ART and runs a Chrono container and VNC, which can be used to see visuals from Chrono.

Like Docker Compose, Docker containers brought up with ATK can be accessed through Docker as usual, mapping up, down, build, and attach flags directly to Docker Compose.

Depending on the version of Docker being used, it may not allow GPU passthrough by default, however an Nvidia GPU is required for using the sensor package in Chrono [9] and to speed up the perception node. Depending on the Docker version, the user may install an additional Docker package. The user will also have to use a specific GPU passthrough flag built into ATK, which will be shown later in this chapter.

5.1 Dev Service

The dev service is run for every container that is brought up, and contains the bulk of the work. Despite the name, this container is used for both development and testing. Fig. 5.1 is a broad overview of the topics and nodes available to the container, which can be run after spinning up the container. Although I didn't use lidar or GPS/IMU for this project, it is helpful to understand the capabilities of the platform.

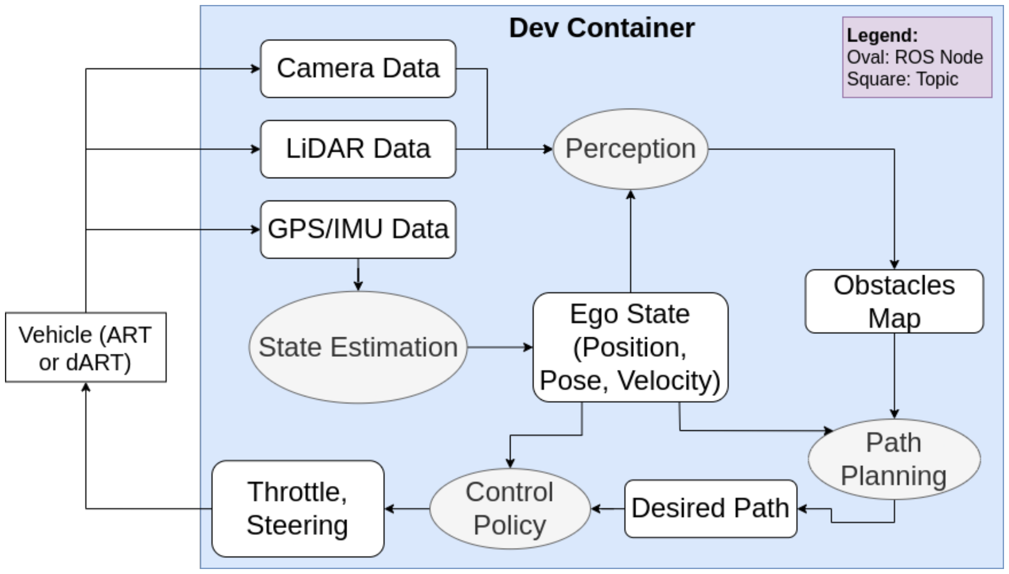


Figure 5.1: Diagram of the different nodes and topics that may be enabled while running the `dev` container

An example command, which would enter us into a development container with ros is as follows:

```
$ atk dev --custom-cli-args gpus
```

The above command would then drop the user in the workspace folder of the autonomy research testbed. There are a number of different commands one can use, however, and they follow the following format.

```
$ atk [flags] dev --service [services] --custom-cli-args [arguments]
```

As of ATK version 1.0.11, all of the flags, services, and arguments, are as follows in the autonomy research testbed default config file `atk.yml`.

- [**flags**]

  - **-v** : Verbosity
  - **-h** : Help
  - **--dry-run** : Performs a dry run and outputs everything that's planned to happen to screen.

- [**services**]
  - **chrono** : Creates a Chrono container
  - **pip-pychrono** : Creates a Chrono container but uses the conda PyChrono binaries instead of compiling from source. Speeding up install time however may not match Chrono versions and isn't recommended.
  - **nx** : Creates a container set up for the AGX, which includes pulling ARM-specific binaries.
  - **vnc** : Creates the VNC container which can be viewed at localhost:8080. This should be spun up automatically by the Chrono service and should never have to be brought up independently.

- [**arguments**]
  - **gpus** : Allows for GPU passthrough into the Docker container and is required when working with Chrono sensor and highly recommended when working with PyTorch.
  - **devices** : Checks to make sure the devices specified in the `atk.yml` file exist. If they don't, the build fails.
  - **network_mode_host** - Sets the Docker network_mode to "host", which changes the network configuration of the container. This can be useful for enabling Docker swarms, although is largely beyond the scope of my dissertation.

- **optix** : Alternative way to configure optix by mounting it to the users local optix folder instead of installing locally. This method isn't recommended but can aid in getting around licensing issues.
- **singularity** : Builds the Docker container using singularity instead of Docker.

One should be aware that since ATK is built off of Docker Compose, the structure between the `.yml` documents are interchangeable. If it's not specified in the ATK docs, it will be specified in the Docker Compose docs.

5.1.1 NX Service

Running on a Jetson requires its own service, which is distinct from `dev` or `chrono` . It is an ARM based platform, and thus there are a number of differences that must be considered when running on an ARM architecture instead of x86. The NX service takes care of the differences in chip architecture by opting to load and run compatible binaries. The NX service is run using the following command:

```
$ atk dev --service nx --custom-cli-args gpus devices
```

One notable example is the PyTorch dependencies, which by default aren't compatible with the Jetson due to differing architectures (x86 vs ARM). They had to be compiled from source, and then are pulled from the new ARM binaries when building the container on the Jetson. Functionally this is done as a bash script, which is called when the dev Docker file is run with the NX service. The docker file recursively calls every script in the 'nx/scripts' directory, which includes l4t.sh. This script uninstalls the torch binaries previously installed with pip, while installing additional build tools and configuring the .whl (pronounced wheel) files which are downloaded from Nvidia. The user may then proceed with pulling the onnx container from git lfs as usual and assemble the cone detection that way.

The devices flag may also be run, which only checks to make sure all of the devices are plugged in when the container is brought up. This is important as the devices are found using default names, and disconnecting and reconnecting a sensor mid use can break the connection and make it so the autonomy stack can no longer find the sensor or device.

5.2 Chrono Service

The Chrono service is an extension of the dev container, however it installs Chrono and works in conjunction with the Chrono ROS bridge to send simulated sensor details to the autonomy stack as if they were real data. The project workflow looks like Fig. 5.1 above. The container is brought up through ATK as well, and can be done so using the following command.

```
$ atk dev --service chrono --custom-cli-args gpus
```

One big advantage of the Chrono container is that it builds PyChrono and Chrono within the container, allowing the user to choose specific branches or even make changes to Chrono during testing. Chrono is cloned and built directly in the dockerfile using ninja, and sites in the home directory where it can be modified and reinstalled. This proved useful when the throttle was calibrated, and it was possible to easily switch branches or make edits to the Chrono build without rebuilding and tearing down the existing container.

If the user doesn't want to build Chrono, there is also an option to just install binaries of PyChrono. This isn't always recommended, since the user will have less control over what parts and versions of Chrono are being installed, however it does reduce container build time significantly. PyChrono itself is installed through conda, however many of its dependencies are installed through pip, which is how the name pip-pychrono was chosen. This can be installed as follows.

```
$ atk dev --service pip-pychrono --custom-cli-args gpus
```

Once the Chrono container has been started, the service immediately brings the user to the sim folder where PyChrono simulations can easily be launched. There are a number of existing simulations such as demo_ARCLAB_IROS.py and demo_ARCLAB_cone.py, although new simulations can be added as well. All of the simulations have been upgraded to Chrono version 8.0, and the existing simulations contain the motor throttle calibration.

5.3 noVNC Viewer

`noVNC` is an open sourced VNC client which gets its name since it can be viewed in a browser, without installing a dedicated client on the host machine. The `noVNC` viewer is used to easily send the desktop environment between the Docker container and the host machine, allowing the user to see output from the Chrono service simulation. As part of the Chrono service, the VNC service should never have to be spun up manually. The user will however be able to see the container ID by running the `docker ps` command.

Unlike the other containers, this one is not accessed through the command line. On the host machine, the user will open a web browser and go to the URL `http://localhost:8080`. Once there, they should see a screen that looks something like Fig. 5.2.

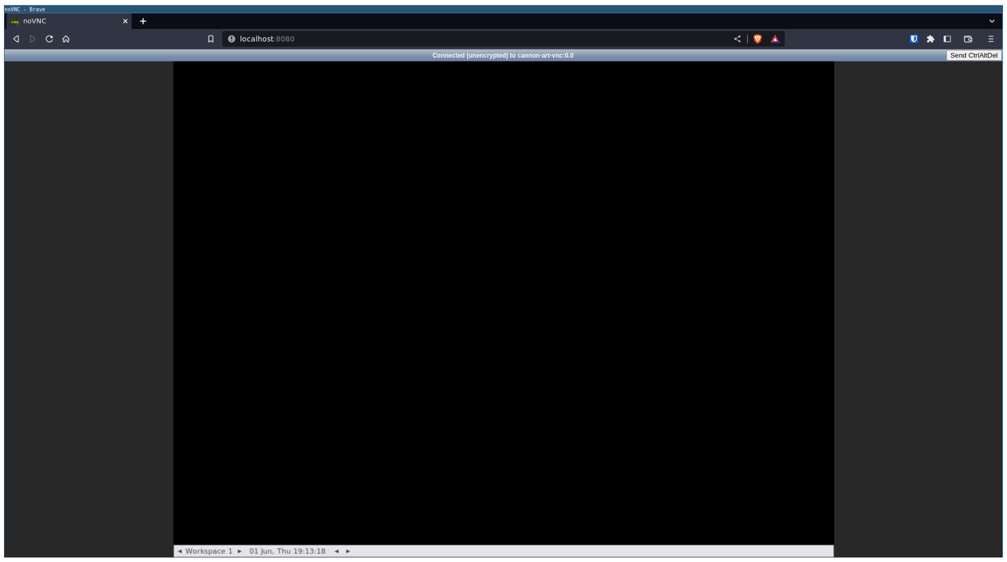


Figure 5.2: Example of what the noVNC viewer will look like before starting a simulation. The user should be able to interact with the screen to open or close windows.

Depending on the Chrono simulation run, the user will see different visualizations appear inside `noVNC`. Additionally through the `atk.yml` file it is possible to change the output port through port forwarding, allowing the user to select different destinations.

5.4 Hardware in the Loop Testing

Hardware in the loop is an important testing feature which allows us to run the ART autonomy stack on the Jetson, while running the Chrono simulation on a more powerful computer, which will be referred to as the host machine. In essence the sensors and output data is still simulated, however the autonomy stack (such as the ROS2 nodes) will run on the Jetson, instead of the host machine. This makes it easier to troubleshoot issues which may be related to running on an ARM architecture or Jetson AGX power constraints, while still running in simulation.

When running hardware in the loop, most things will work the same, however the host and Jetson will have to be configured to talk to each other. These instructions are for Linux users, although a similar solution should exist for Windows and Mac.

1. Connect the Jetson and Linux host machine together using a direct ethernet cable. Using ethernet reduces latency and bandwidth issues which may otherwise be seen, however isn't technically necessary

2. Ensure ethernet connections are working. The easiest way to do this is go into the network configuration for both the host machine and Jetson and make sure they see each other.

3. Open `etchosts` file and add the other devices IP address and host name. So if the Jetson's host name is `chokecherry` at IP address `192.168.0.1`, the user would add the line `192.168.0.1 chokecherry` to the `hosts` file in the host machine.

4. On the host machine, the user will have to add a new display variable to the `.bashrc` file. To do this go to the home folder, and if the user is using bash they will open the hidden file labeled `.bashrc`. For zsh, they will instead have a file titled `.zshrc`. Open that file and add the following line to the bottom.

   ```
   export DISPLAY=vnc0.0
   ```

   Once done, close and reopen the terminal to reload the `.bashrc` script.

5. Start up the Jetson container like normal and the Chrono container on the host machine. On the host machine, this will look like

   ```
   $ atk dev --service chrono --custom-cli-args gpus
   ```

   From here the python environment may be started like normal on the host machine. Next, in the Jetson the NX service should be started.

   ```
   $ atk dev --service nx --custom-cli-args gpus
   ```

   After which the user will have to pass in the IP address of the host machine, running the Chrono instance.

   ```
   $ ros2 launch art_simulation_launch art_simulation.launch.py \
   >   hostname:=art-chrono vis:=true ip:=<host machine IP>
   ```

6. Finally the user should be able to open their `http://localhost:8080/` noVNC instance to visualize the Chrono simulation, assuming they have vis set to true.

Having the ability to use hardware in the loop is a powerful tool which allows for much better troubleshooting, particularly in final testing.

6 END TO END LEARNING

One of the major goals of my research is implementing an end-to-end solution to vehicle controls, where the vehicle could take in images from the camera and output the correct steering and throttle responses. For the ART platform, this means training the agent to interpret images from the Chrono camera sensor, and learn to output a throttle and steering response as to stay between the cones. The ultimate high-level overview will look like Fig. 6.1, where sensor data feeds directly into the agent, which outputs vehicle commands.

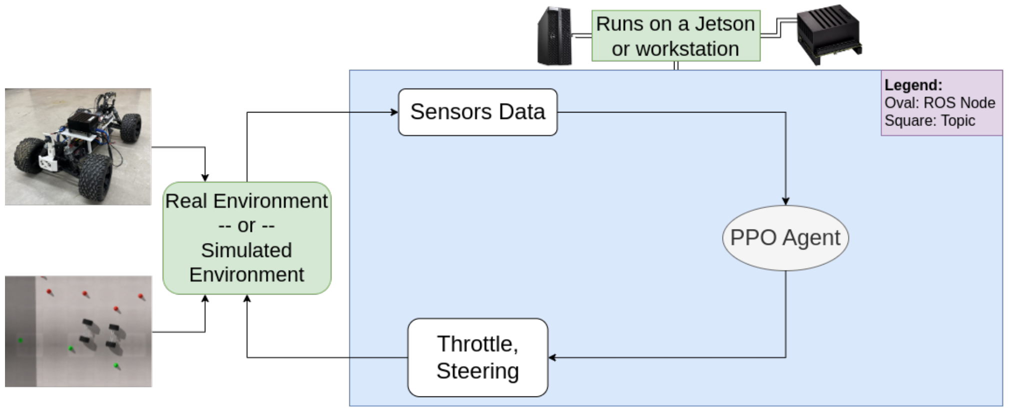

Runs on a Jetson or workstation
Legend:
Oval: ROS Node
Square: Topic
Sensors Data
Real Environment -- or -- Simulated Environment
PPO Agent
Throttle, Steering

Figure 6.1: Here you can see a high level diagram of what the software structure would look like in the ART platform. Here you can see many of the topics and nodes in Fig. 3.2 have been removed in favor of a reinforcement learning agent (running Proximal Policy Optimization, or PPO).

This differentiates itself from previous work on reinforcement learning with ART, as the throttle calibration has been improved, a more recent PPO reinforcement learning agent with a different reward function is being used, and the autonomy stack will no longer be having an intermediary network detecting cone position [2]. Further it will rely only on the camera sensor, and it won't be using GPS data to learn its position.

6.1 Reinforcement Learning

A foundation of my work relies on reinforcement learning, which will be leveraged to create a real world end-to-end learning environment for the ART vehicle. Gym will act as the environment in which the vehicle exists, and the agent will be trained using algorithms from Stable Baselines3 to detect cones and navigate through a course. Ultimately this thesis attempts to build an agent that will become transferable to the real life ART vehicle.

Fundamentally, reinforcement learning algorithms are a Markov Decision Process with a set of states S, a set of actions A, and a reward value R. Ultimately the goal is to maximize the reward function $R : S \times A \times S \Rightarrow R$ [21]. Here I'll attempt to go over each, and why these decisions were made.

6.1.1 OpenAI Gym

Gym is a library designed to help with reinforcement learning, originally produced by OpenAI [4]. Gym manages environments, which are simulations managed by the reinforcement learning agent, and attempts to introduce some standardization into reinforcement learning, allowing users to create custom environments or use default environments to test different algorithms. Gym itself does not house the reinforcement learning, rather guides the reward policy and feeds the observations back into the reinforcement learning algorithm.

The advantage to using Gym over building a unique solution to manage the environment is primarily the standardization of API's. It may not always be clear which reinforcement learning algorithm is best, and Gym makes it easier to switch between different libraries without changing the underlying code.

A crucial aspect of Gym is how the simulation interacts with the agent. In particular, Gym manages the specifications for how rewards are returned and the observed results. A policy agent exists with the main program, which starts at a random position. From there, it will perform an action and receive back the reward and observed result of that action, after which it will continue to perform actions until the program has ended. An example of this can be seen in fig 6.2.

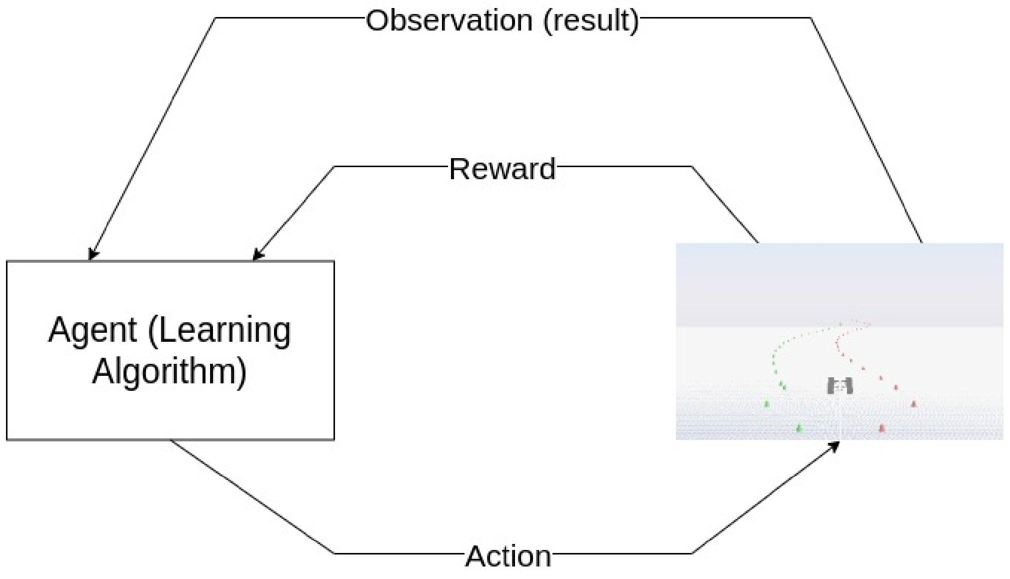


Figure 6.2: Here we can see a high level overview of the communication channels created by Gym. The agent sends actions to the environment (in this case a custom environment) which returns rewards and observations, allowing it to then perform another action. This repeats until the custom environment either finishes or cannot proceed anymore.

The agent is what interacts with the environment to perform the learning, and represents one of the learning algorithms which I will use from Stable Baselines3; although a number of different libraries and algorithms could be used instead.

Currently Gym is no longer under active development, and has been replaced by Gymnasium. That said, the latest release is stable and due to its popularity and relatively complete feature set I have decided to continue to use it here. Installing it is quite simple; the user can install Gym along with all of its example environments through pip.

```
$ pip install gym[all]
```

6.1.2 gym-chrono

For my research, I used gym-chrono which is a custom environment developed for use with Chrono models[2]. It's based on Gym, and can be installed locally using pip.

The gym-chrono package officially supports three environments: chrono_pendulum-v0, chrono_ant-v0, and chrono_hexapod-v0, although more unsupported custom environments exist as well. The existing environments are all written for Chrono version 7.0 and do not contain the required sensors or vehicles, necessitating the creation of a new environment within this package.

For my research, I have created a new custom environment, using the existing base class from gym-chrono while also using Chrono version 8.0 for the simulation. The environment uses the camera sensor to detect cones and the digital ART vehicle. Rewards will be determined by how well it stays within the cone lines while still moving towards the end goal, and the observed result will be the input from the camera sensor.

The action space will be throttle and steering, with a range of 0 to 1 and -1 to 1 respectively. The camera sensor serves as the observation space, with the entire image being passed back to be processed through the neural network used for reinforcement learning.

Rewards are computed based on relative success. Here I will attempt to return a larger reward the more quickly the vehicle is moving towards the end result, while returning a smaller reward the more time has passed since the start. Colliding with cones or timing out will result in a failure, which will also produce a low reward.

6.1.3 Custom Gym Environments

To be able to use the gym-chrono package in my research, I first had to add a custom Gym environment to the package. I named this environment `off_road_rccar-v0`. Custom environments follow a set file structure to encourage uniformity, which can be seen in Fig. 6.3.

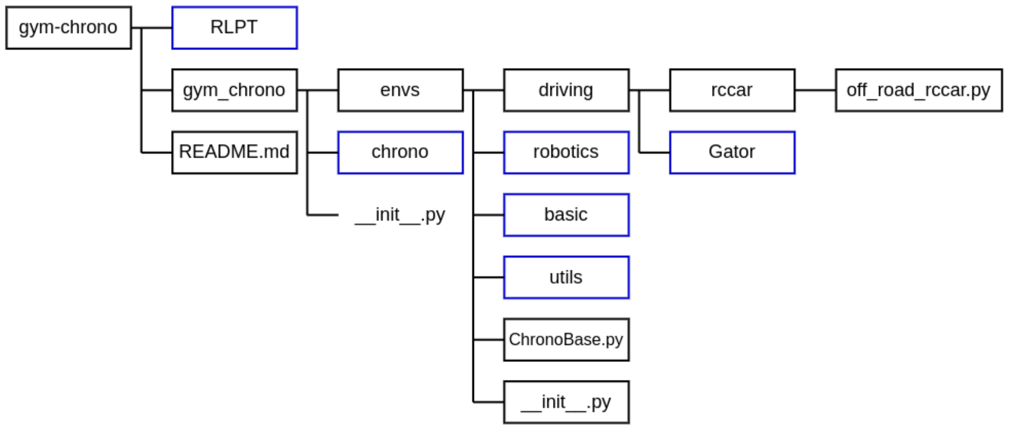


Figure 6.3: A brief overview of how gym-chrono is structured. Blue boxes mean the folder does not need to be modified in order to add another environment. New folders can be made so long as they are registered within Gym.

Normally custom Gym environments must all be a subclass of their custom class `gym.Env`, however since this was part of gym-chrono I was able to base it off of the custom super class, `ChronoBaseEnv` instead.

6.1.3.1 Register Environment

First we must register the environment inside gym-chrono. This is done by adding the environment name to the `gym_chrono/__init__.py` file. This is where Gym finds information about the environment, such as the version number and name. All other environments sit alongside this one as well in the file

```
import logging
from gym.envs.registration import register


register(
    id='off_road_rccar-v0',
    entry_point='gym_chrono.envs:off_road_rccar'
)
```

Secondly we must import the new environment along with the rest of the environments, in `gym_chrono/envs/__init__.py` as a normal Python file (which would of course be followed by the rest of the custom environments in gym-chrono).

```
from gym_chrono.envs.ChronoBase import  ChronoBaseEnv
from gym_chrono.envs.driving.rccar.off_road_rccar import off_road_rccar
```

This will allow us to make the environment, but it won't let us run anything yet.

6.1.3.2 Choosing Action and Observation Spaces

Choosing an action and observation space are of high importance in Gym, and determine the flow of the rest of the program. These are set as public object variables, and Gym will read their type when determining how to learn.

Gym offers a few variable types for creating custom observation and action, which are defined in the Gym spaces library. I will be using their Box type, due to its similarity to matrices and the ability to define its relative size. It works like a normal array, and can take float values unlike its discrete counterparts. Once defined, Box variables function like a numpy array, making them easy to manipulate and understand. Lastly, it works well with PPO.

For my action space, I want to pass in a throttle and a steering input, each of which are bounded by 0 to 1 and -1 to 1, respectively. This ranges and the size of the action space are defined as public variables in the environment class during the init function, so that Gym is able to find them and use them.

Next the observation space must be defined, which will function similarly to the action space. For the Chrono camera sensor, the camera has a red, green, and blue value, each of which are integers ranging from 0 to 255. Additionally it has a transparency value, however this is discarded and not fed back into the agent. Because of this, I did not take advantage of the floating point option in the Box type and used discrete integers. Further the size of the

Box is defined by the number of pixels on the camera, meaning we must use the same sized frame for both the camera in Chrono and the camera on the vehicle. Ultimately this created a relatively large observation space to train over, which was a three dimensional array which spanned the camera width, height, and three color values deep.

Like the action space, the observation space is set in the init function to a public variable, which can by read in by Gym.

6.1.3.3 Custom Environment Structure

Each custom environment must have at minimum four methods contained within an object, although Render doesn't have to be fully implemented. Other methods like `close(self)` do exist but can be inherited without implementing.

- `__init__(self)`

This method is called once when the object is first created, however never called again. It generally contains configuration information and other object initialization that will not be modified in the future.

- `reset(self)`

This method is called every time the environment resets, which occurs once at the beginning and then every time the agent reaches a terminal condition that ends the simulation.

- `step(self, action)`

The step method is roughly equivalent to a timestep in Chrono, and increments the simulation forward. In doing so it accepts an action, which is an array containing the steering and throttle values. These are fed into the simulation, and the observed state S along with the reward R is returned. These can then be fed back into the agent, to produce new actions.

The step function also returns a `done` value, which indicates whether the simulation has completed. If it has, the library managing the learning algorithm, in this case stable_baselines3, can call the reset function for the next pass.

- `render(self)`

The render method renders the simulation for the user to see, which in this case is equal to the observation. This slows down training, and for that reason is usually only called when debugging.

The challenge for me was implementing a Chrono environment in these methods. Namely, I had to be able to completely reset the environment every time the `reset` method was called, and I had to consider values such as Chrono's step size as hyper parameters which needed to be tuned for the agent.

6.1.3.4 Reward Function

The reward function is one of the most important parts of reinforcement learning, since it's what the agent relies on during training. The goal of the agent is to maximize the reward, and the goal of the reward function should be to feed rewards to the agent as it chooses appropriate actions, and penalize it when it does poorly.

One way we can do this effectively is to use **shaping functions**, as opposed to a sparse reward function. These are functions which gradually feed in positive rewards as the agent does better, and gradually penalizes it as it does worse. You can visually see what this looks like in Fig. 6.4, where we want to avoid only rewarding at the end, as it can make it difficult or impossible to properly train an agent. Ideally we want to keep shaping rewards between 0 and 1.

Additionally we want to factor in **terminal conditions**, which are large rewards or penalties given when a specific goal or targeted state is achieved. These may seem to be in direct contrast to shaped functions, however there are many things a user may want to strongly discourage or reward. For example, the agent doesn't need to explore after falling off the terrain, so a high terminal penalty for driving off the terrain will help discourage future exploration. On the flip side, I do want to strongly encourage finishing correctly, so I implemented a high terminal reward before ending the session.

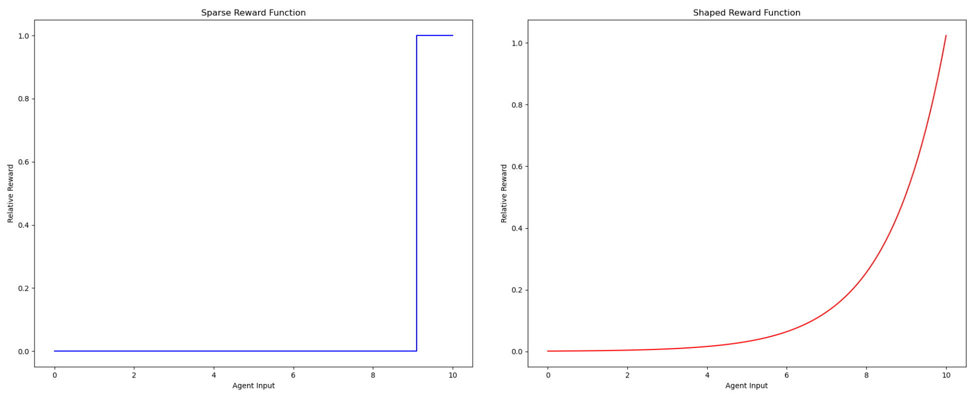


Figure 6.4: An example of a sparse reward function and a shaped reward function. Implementing sparse functions over variables which could be shaped can result in slow or no learning, as the agent must try every possibility until it reaches the correct solution.

When training the agent for my research I relied on distance to the end and time to complete as major shaping factors. I used a simple euclidean distance calculation for the distance to the known endpoint, and although the vehicle didn't move in the z direction it was included to fight against any out of bounds errors that came up. I also included a tuning coefficient d_c which ultimately helped keep the distance within the -1 and 1 range. Once within 0.75 meters of the end point the simulation was ended and the agent rewarded with a terminal position.

$$cur\_dist = \sqrt{(veh\_pos_x - end\_pos_x)^2 + (veh\_pos_y - end\_pos_y)^2 + (veh\_pos_z - end\_pos_z)^2}$$

$$dist\_rew = d_c\,|previous\_dist| - cur\_dist$$

Ultimately the calculated distance was subtracted from the previous distance and multiplied by the tuning coefficient d_c to get the reward for the proximity to the endpoint.

We also wanted to produce a shaped penalty function for time. This was just a linear function, taking the time since starting and multiplying it by another tuning coefficient t_c, which would slowly penalize the vehicle for taking too long until it timed out. `chrono_time` starts at 0 and counts up in seconds after the chronon simulation has started. The penalty function is shown below.

$$time\_pen = t_c * chrono\_time$$

Finally we had a penalty for running into cones. Colliding with cones would end the simulation and had a separate terminal penalty, however in addition we would penalize the vehicle for each cone it ran into. *c_collisions* was an array of every cone, which is initialized to 0 but incremented to 1 when the vehicle collides with a cone. Further there was a c_c which was another tuning coefficient that would modify the penalty to the vehicle. In the event the vehicle ran into multiple cones, it would be penalized more than running into one. *num_cones* of course refers to the total number of cones.

$$cone\_pen = c_c \sum_{i=0}^{num\_cones} c\_collisions_i$$

All of these rewards and penalties were summed up at the end, and used as the reward value for each step.

$$reward = dist\_rew - time\_pen - cone\_pen$$

This was the reward returned at the end of the step, but with the exception of cones it doesn't include the terminal conditions. For those we would wait until the simulation was finished, so on its final step we would add or subtract a constant value for the vehicle.

This was done when the vehicle reached 0.75 meters from the end point, when it exceeded the z axis bounds, or when it fell out of bounds on the map. Although considered, I found I didn't need a time out for my training.

6.1.3.5 Cone Setup

The cone paths for the vehicle had to be generated automatically, for the Chrono environment to work properly. All cone paths follow a sinusoidal path, however the final position and amplitude of the curves can be adjusted for training. Visually one can see what one of these paths look like from the top in Fig. 6.5 and Fig. 6.6.

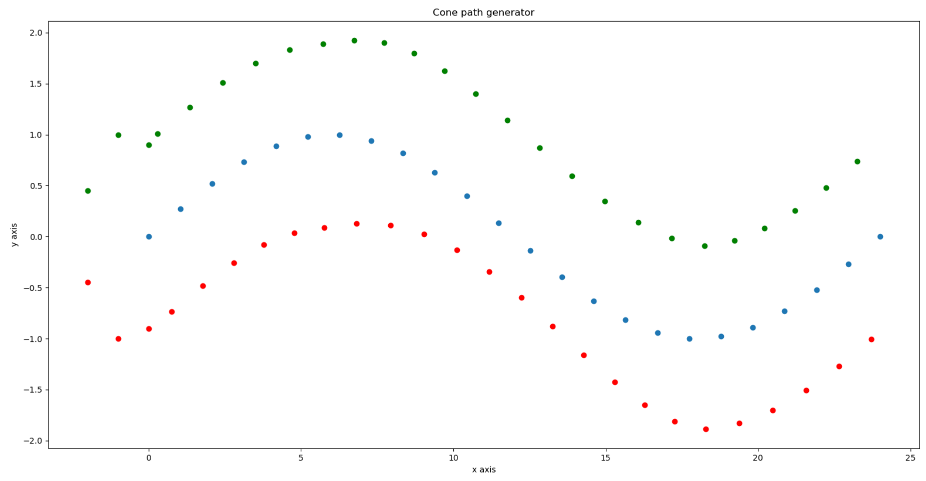


Figure 6.5: An sample of one of the cone layouts from training.

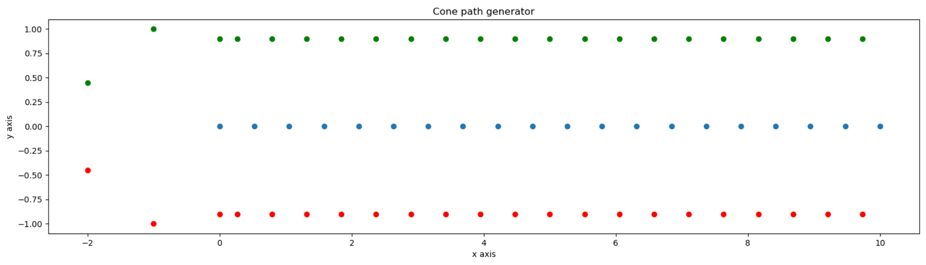


Figure 6.6: An example of a shorter, straight cone path. This is a simpler path, which may be easier to train.

In Fig. 6.7 is a high resolution example of what the camera sensor in Chrono sees when it's looking at the path. When training, it will learn just off of these images to detect the cones and choose a corresponding throttle and steering value. Due to the high computational costs of processing 720p video, the actual vehicle sees a downscaled 420x320 scale image, as seen in Fig. 6.8. A significant portion of the detail is now missing, however the agent can still identify cones and avoid them.

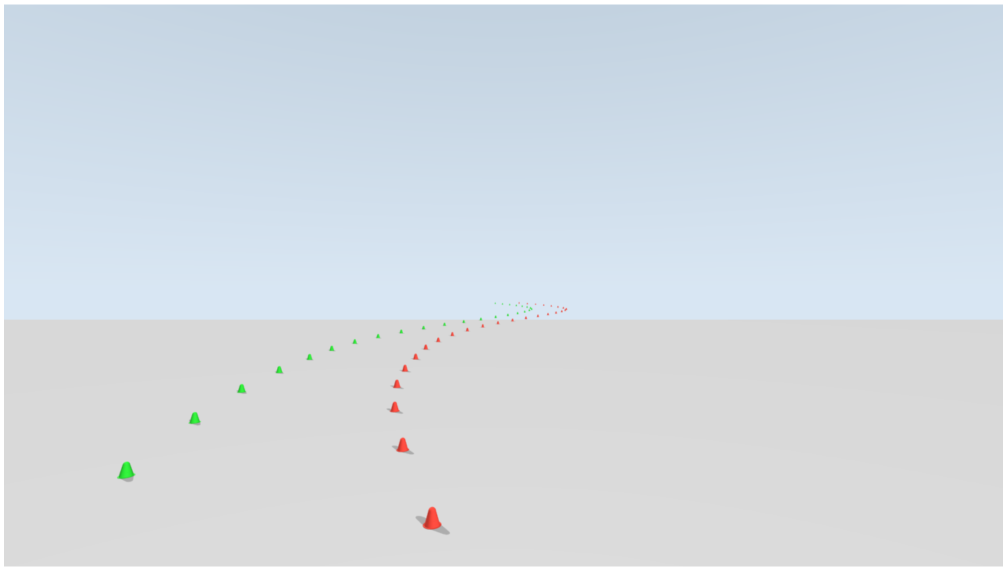


Figure 6.7: The first frame of training, while the vehicle is still at its starting position. You can see the cone path in front of it. This frame was taken at 720p resolution.

The cone setup works by taking an input of a destination and amplitude, and generating a path towards a target destination using 24 cones on each side. The path generation produces relatively similar paths which all follow a sine wave and are composed of 24 cones on each side.

Behind the vehicle are four additional cones, forming a semi circle around the dART vehicle as seen in Fig 6.9. This is done to prevent the vehicle from turning around to drive around the cones to the destination, as it was doing on previous occasions.

6.1.4 Stable Baselines3

Stable Baselines3 [25] is a library based on OpenAI Baselines [7], whose goal is to create an easier to use set of algorithms. It does away with the command line interface of baselines and

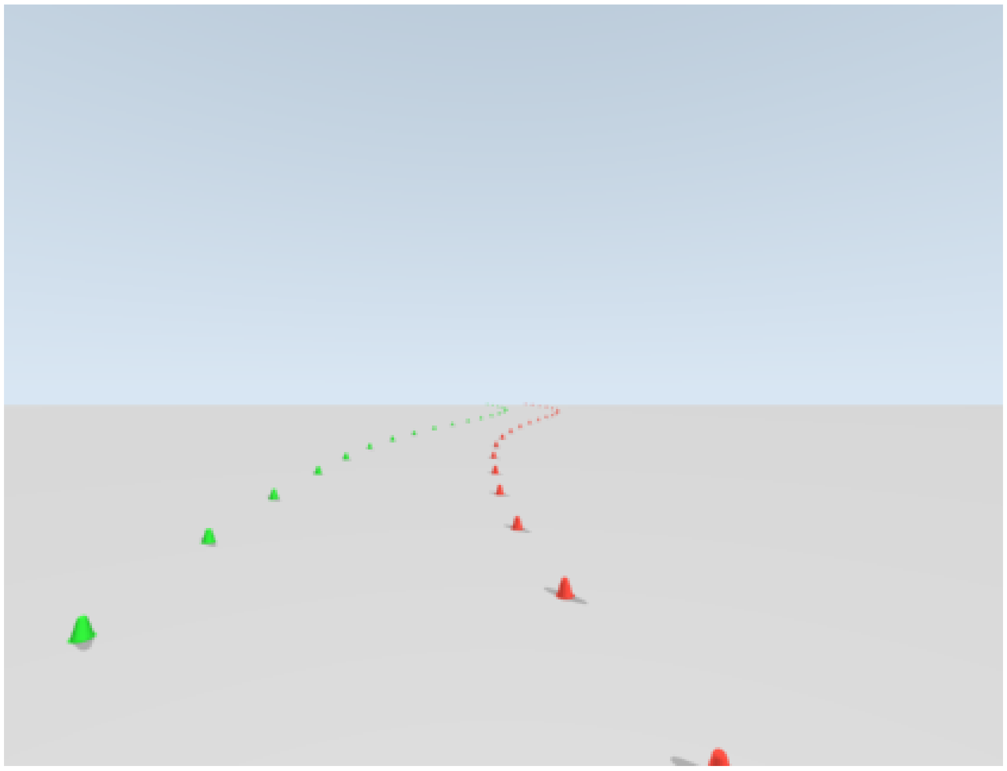


Figure 6.8: A frame of the scaled resolution being used by the vehicle, with a lower resolution.

instead implements a number of functions to write Python scripts, and introduces functions like `check_env(env)` which allow the user to run their environment through a number of preliminary assertions to help see if anything is incorrect.

Further it implements a number of Reinforcement Learning algorithms, as well as a base class where users can add their own. Reinforcement learning policies fall into two categories: **on-policy** and **off-policy**. **On-policy** algorithms do not keep a running history of all environments and outcomes, rather the agent is updated each time the environment runs and the data isn't saved. **Off-policy** algorithms keep a running database of observations, actions, and rewards, and train off a set of collected data.

Skipping the details, the seven algorithms are as follows.

- A2C, Advantage Actor Critic. A synchronous and deterministic version of Asynchronous Advantage Actor Critic (A3C)

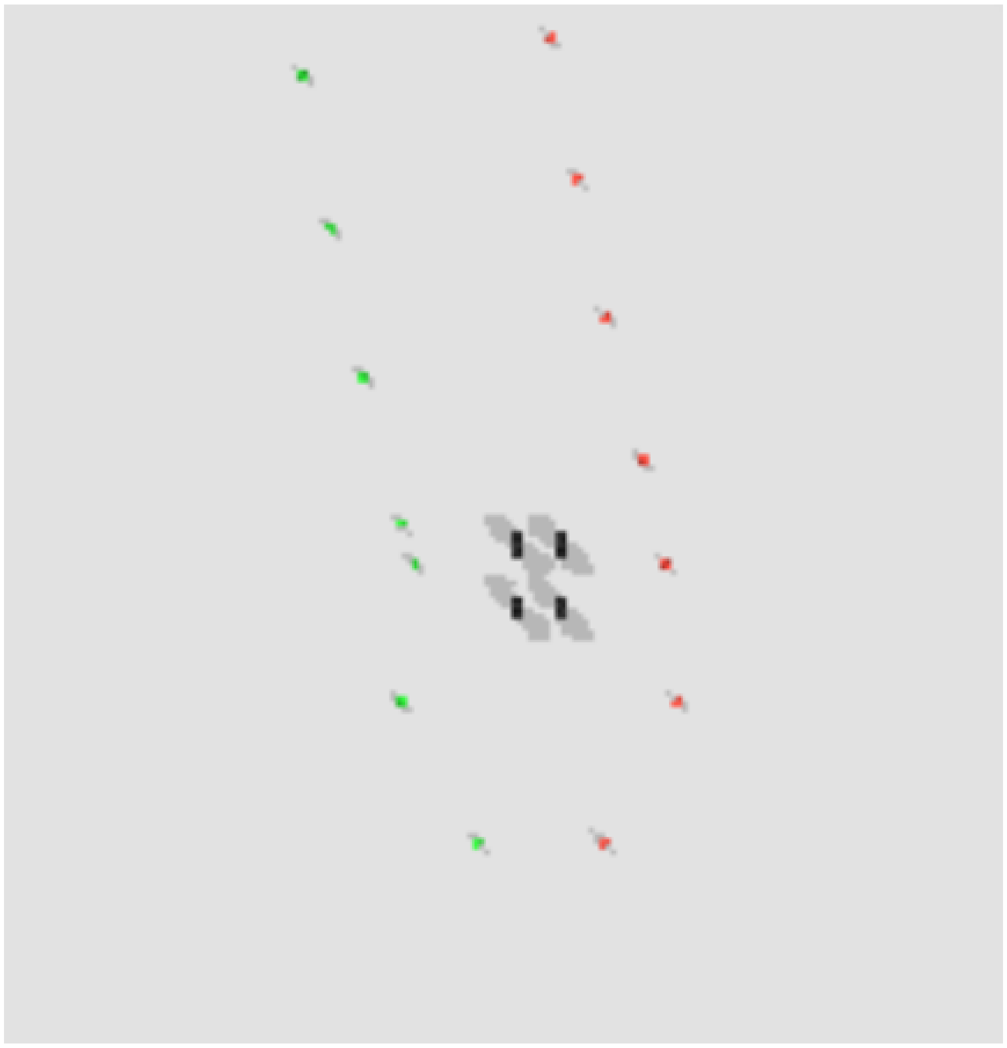


Figure 6.9: Example of semi-circle, with the vehicle in its starting position of $[0, 0]$. The rear of the dART vehicle is surrounded by the semi-circle, while the front is pointing towards the opening in the cones.

- DDPG, Deep Deterministic Policy Gradient. Combines DQN with a deterministic policy gradient [20]

- DQN, Deep Q Network [22]

- HER, Hindsight Experience Replay. A form of off-policy training which attempts to learn more from failed attempts, making it better at especially sparse environments with shaped reward curves and multiple objectives [24].

- PPO, Proximal Optimization Policy. A form of on-policy learning technique which was used in this research.

- SAC, Soft Actor Critic. An off-policy form of learning, similar to Q-Learning.

- TD3, Twin Delayed DDPG. Implements improvements on DDPG by taking parts of Q-learning while attempting to limit its tendency to over estimate. [13]

6.1.4.1 Policy Proximal Optimization (PPO)

I used PPO as the reinforcement learning actor in my research. First introduced in 2017, it was designed with a goal of being more scalable and robust than other competing reinforcement learning algorithms such as Q-learning or Trust Region Policy Optimization (TRPO) [30]. Since then it has been found to be surprisingly sample efficient for an on-policy algorithm, and has found use in other complex, multi-agent research topics [38]. This is in part due to PPO's use of TRPO, which constrains updates to a certain region around the current policy. The constraints on exploration, combined with PPO's simplification of tuning make PPO a good choice of reinforcement learning algorithm for my research.

A notable difference between PPO and other forms of Q-learning is that PPO doesn't use a replay buffer or history of stored experiences. As we train on the policy, it's implemented directly into the agent as it is encountered, and then the data is discarded.

Policy gradient methods usually start with an estimator of the policy gradient, and in the case of PPO it takes the following form:

$$L^{PG}(\theta) = \hat{E}_t[log\pi_\theta(a_t|s_t)\hat{A}_t],$$

where $\hat{A}_t$ is an estimator of the advantage function, π is the policy function, and L^{PG} is the policy gradient loss function. These all iterate through time, which is represented by the time step t.

The advantage function $\hat{A}_t$ then is in charge of calculating rewards, prioritizing high values that are returned quickly from the user defined reward function. The policy function π_θ is used to calculate the actions to take and predicted values. As the PPO algorithm updates, it uses the previous policy weights $\pi_{\theta_{old}}$ and advantage estimates $\hat{A}_t$ to calculate a possible solution for a given environment, and then optimizes the $L^{PG}(\theta)$ loss function. It updates the weights, and continues this process until the user ends the learning process [30].

6.2 Training in Chrono Environment

6.2.1 Training with Gym and Stable_Baselines3

The actual code for training is quite simple, thanks to Gym. Using the Gym API we can easily add in stable_baselines3's PPO algorithm [25] and begin training. A portion of the code is shown below.

```
env = make_vec_env('off_road_rccar-v0', n_envs=1)

model = PPO("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=25000)
model.save("ppo_rccar")

# returns an initial observation
observation, info = env.reset()

while True:
    action, _states = model.predict(observation)
    observation, reward, done, info = env.step(action)
```

Further by using the `model.save("ppo_rccar")` method, the agent is automatically saved as it trains and can be reloaded for further training. This not only allows us to use it after training, but in the event of a program crash we can pick up where we left off in retraining. Additionally `make_vec_env` takes in a parameter `n_envs`, which allows the user to train on multiple environments simultaneously, speeding up training time.

Ultimately I did all of the training on my home machine, running an Nvidia RTX A4500 graphics card and Intel 12700 cpu. The Nvidia graphics card was necessary for this setup, as Chrono sensor relies heavily on CUDA, a proprietary parallel computing package from Nvidia. The agent is saved as a local zip file, which makes it easy for the user to save or reload the model while training, including stopping to get a birds eye view. Visualizing what the vehicle was doing is done using a separate Chrono camera sensor which saves image frames locally to be viewed later.

Due to the high computational cost of training the full simulation, I would often start out on a smaller course, such as displayed in Fig. 6.10. If these tests were successful, I would move up to the full sized course.

Notably there is no visual endpoint for the vehicle, rather the reward function R is told where it is and the vehicle had to learn that the cones produced a negative reward and by following them it seemed to do better.

6.3 Running the Agent

An advantage of using a PPO agent is its simplicity to run on subsequent iterations. Similar to training, the model is run in a loop, except it is loaded from a `.zip` file which contains the model.

A portion of the code is shown below, where the model is stored in the same directory the python script is being run in, and is named `ppo_agent.zip`.

```
env = make_vec_env("off_road_rccar-v0", n_envs=1)


# PPO Policy
```

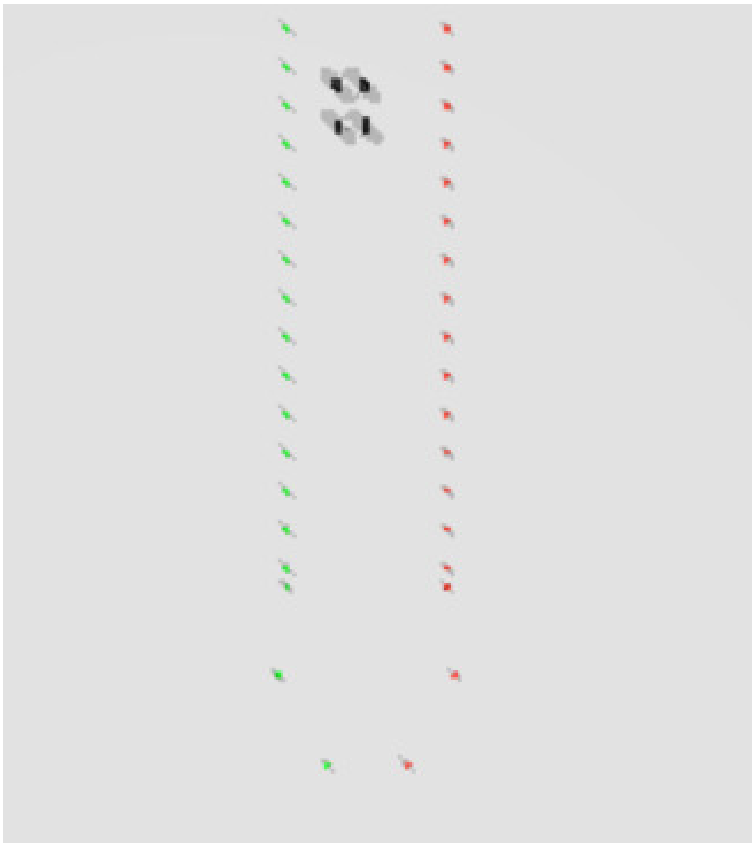


Figure 6.10: One of the bird's-eye view images taken from training. Here the vehicle, surrounding landscape, and cones are visible. The course was to learn to drive forward 6 meters, which it is successfully completing in this final frame.

```
model = PPO.load("ppo_agent")

observation = env.reset()


while True:

    action, _states = model.predict(observation)

    observation, reward, done, info = env.step(action)


    if done:

        print("Done!")

        break
```

```
env.close()
```

6.4 Observations and Challenges

There were a few things I noticed while training the agent. Notably during training, the agent would become confused as the difficulty of the field increased. One interesting observation was that on occasion the vehicle learned it could fit through the cones, and after a series of failed attempts would learn to escape the course as seen in Fig. 6.11, where it would wonder until it accrued a negative reward and then drive out of bounds.

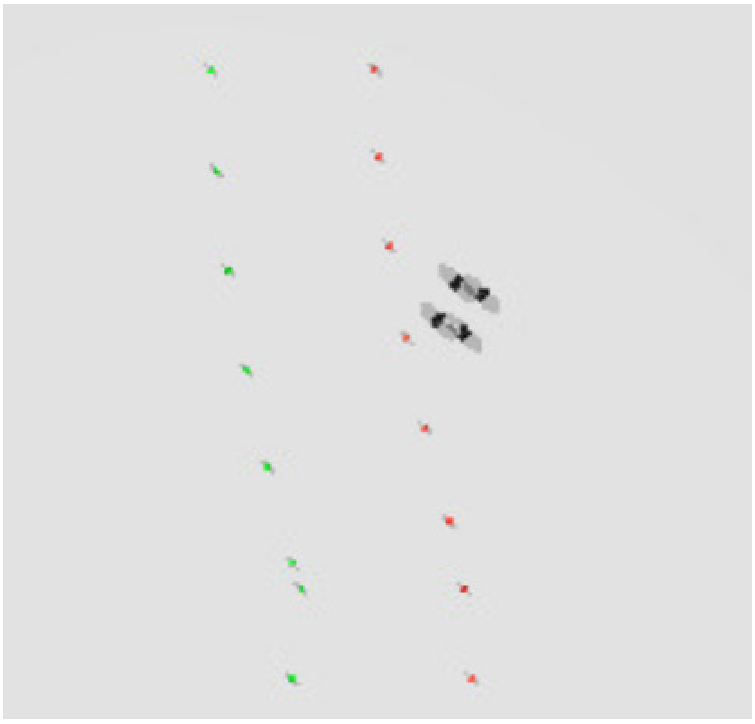


Figure 6.11: The dART vehicle can be seen escaping the path by driving between two cones.

It would be possible to instead punish the vehicle for driving outside the path at all, however I wanted to tie a direct relationship between colliding with cones and being penalized, instead of just an arbitrary boundary.

Training on paths as they became longer also was an increasingly difficult task. Due to the on-policy training and high number of samples, the agent would sometimes have trouble connecting the dots between an action and penalty. When this happens, it could end up accidentally learning incorrect behavior, getting stuck in a loop where it would fail for thousands of iterations in a row and have to be restarted.

Another observation is that the major complexity from learning with SCM came from implementing it in code, rather than the learning process itself. Since reinforcement learning requires running tens of thousands of simulations to learn a skill, I would require tens of thousands of simulations to begin completing the simulation. Doing so proved more difficult in SCM than a Rigid terrain, as the SCM was more susceptible to the vehicle encountering some form of catastrophic error, usually occurring when the vehicle first drops into the environment and comes into contact with the terrain. This resulted in a few changes, including a trigger to catch when the vehicle is acting erratically and moving the vehicle as close to the terrain as possible.

7 CONCLUSIONS AND CONTRIBUTIONS

This thesis is concerned with two major objectives: demonstrating a novel end-to-end learning algorithm in combination with the ART platform, and providing written guides for many features in the ART platform which do not exist in published form. Additionally, the thesis discusses multiple improvements to ART, including through improved calibration and model accuracy, testing in off-road conditions simulated by SCM [35], as well as improvements to the reinforcement learning algorithm and setup.

- The thesis relied heavily on dART and takes advantage of the recent throttle calibrations [36]. It also included a brief explanation of how it works, and the values needed to implement it other ART-based projects.
- I built a new Chrono environment using the Gym framework, capable of performing tens of thousands of tests repeatedly as required for training the reinforcement learning agent. This environment was designed around use of the dART vehicle model, although I tested it on other high quality Chrono vehicle models such as the humvee with success. Additionally the model used Chrono's SCM terrain environment, as opposed to a rigid terrain.
- I discuss the gym-chrono package, in particular how it is structured and how one may implement a new custom PyChrono environment within it.
- The complete end-to-end learning environment has been demonstrated to work using purely visual inputs to handle the throttle and steering without relying on additional sensors for small courses. This is a step up from previous work, which would require additional dedicated models to determine cone position or rely on other sensors, such as GPS.
- I provide a detailed guide on how to run ART and the ART and ATK software stacks, which currently do not have any published guides outside of the official ATK docs.

- This thesis presents a detailed overview of the ART vehicle, including power distribution, steering, sensor setup, and onboard computer.
- A description of how to run ART is presented, including a general discussion of the project file structure as well as how to launch the vehicle and save data.
- A description of how and why to use the Chrono and NX services is presented. Further I enumerate how to perform hardware in the loop testing on the ART platform.

8 FUTURE WORK

The nature of ART is that it's a platform for development; so as long as we can think of new tools to build, or the sim-to-real gap exists [17], there will be work left for us to do. Because of this, there are lots of potential areas for improvements, including to the reinforcement learning agent, closing the sim-to-real gap on the physical vehicle and digital twin, and improving the simulation we train in.

8.1 Future Work on Learning Model

In regard to the learning model, the path forward would be to continue to implement new versions of the reinforcement learning agent into the physical vehicle and perform testing to see where the vehicle does and doesn't succeed. Much of the work relating to real world testing still needs to be done, which would allow for comparisons to be drawn about the success of Chrono testing. We could potentially go back to the ARCLab where we did the throttle calibration and run the vehicle in there, where we could get high precision positional measurements for where the vehicle is and compare that to the result of the learning agent. One of the major innovations of this thesis was removing the ONNX cone detection from the vehicle in favor of an end-to-end model, and there is a lot more real world testing that can be done with this updated model.

Further it would be interesting to look more heavily into other reinforcement learning algorithms. PPO is only a few years old, and we're constantly seeing new algorithms for training RL models come out [31]. It's likely we'll see more algorithms come out and it would be interesting to continue to update our agent, experiment with Q-learning algorithms, or even create and test our own models on ART. One learning algorithm that might show promise is Hindsight Experiment Replay (HER). HER is an off-policy replay buffer based on a Deep Q Network (DQN) designed to take on sparse reward functions. This may mean we could perform fewer tests, while training on similarly complex reward functions [24].

8.2 Future Work Simulation

There are lots of improvements we can make to the simulation, including making it more realistic. One such improvement which has been done in Chrono in the past is to use GAN networks for improved image accuracy [10]. GAN networks, or generative adversarial networks, are used to create photo realistic images and would greatly improve the background. This would in turn improve our agents ability to distinguish cones from its surroundings, and would produce more realistic results in a simulated environment. We have seen this done successfully in Chrono as well, where the accuracy for learning on images from Chono sensor was improved by using a GAN. In Fig 8.1 we can see images from this previous research, where the author was able to produce high quality background images and fill in the gaps using a GAN model.

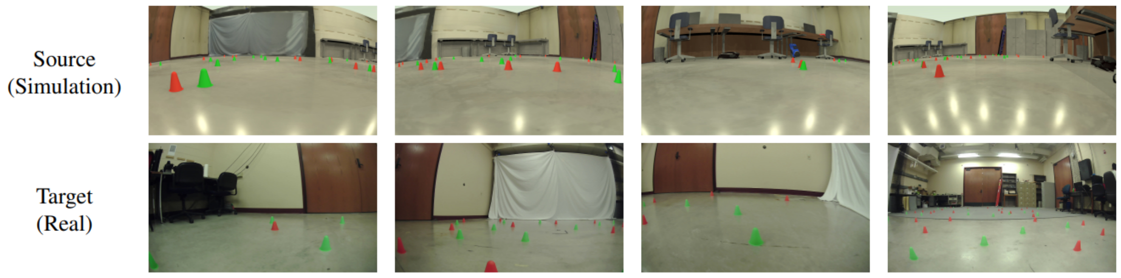


Figure 8.1: An example of GAN images being used in simulation in the ARC lab, compared to the real images [10].

Additionally the terrain used in my research was flat, which lacked some of the real world obstacles found in off-road terrain. Future versions would benefit from using a shaped terrain which might mimick small hills. Rigid body objects such as rocks in the path could also make the path more complex and challenge the ART vehicle.

8.3 Future Work on ART Vehicle

Lastly there can be more work done on the vehicle and its corresponding sensors. Calibrating sensors is directly related to closing the sim to real gap, and work can always be improved. Additionally new mounting hardware could improve the usability of the vehicle, to better suit new Jetson AGX and Jetson Orin computers.

BIBLIOGRAPHY

[1] David M Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, 2003.

[2] Simone Benatti, Aaron Young, Asher Elmquist, Jay Taves, Radu Serban, Dario Mangoni, Alessandro Tasora, and Dan Negrut. Pychrono and gym-chrono: A deep reinforcement learning framework leveraging multibody dynamics to control autonomous vehicles and robots. In *Advances in Nonlinear Dynamics*, pages 573–584. Springer, 2022.

[3] Simone Benatti, Aaron Young, Asher Elmquist, Jay Taves, Alessandro Tasora, Radu Serban, and Dan Negrut. End-to-end learning for off-road terrain navigation using the chrono open-source simulation platform. *Multibody System Dynamics*, (https://doi.org/10.1007/s11044-022-09816-1), 2022.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

[5] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, Chen Li, Franziska Meier, Dan Negrut, Ludovic Righetti, Alberto Rodriguez, Jie Tan, and Jeff Trinkle. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1), 2021.

[6] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number CONF, 2011.

[7] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. `https://github.com/openai/baselines`.

[8] Jens Ejury. Buck converter design. *Infineon Technologies North America (TFNA) Corn Desion Note*, 1:1, 2013.

[9] Asher Elmquist, Radu Serban, and Dan Negrut. A sensor simulation framework for training and testing robots and autonomous vehicles. *Journal of Autonomous Vehicles and Systems*, 1(2):021001, 2021.

[10] Asher Elmquist, Radu Serban, and Dan Negrut. Evaluating a GAN for enhancing camera simulation for robotics. *arXiv preprint arXiv:2209.06710*, 2022.

[11] Asher Elmquist, Aaron Young, Thomas Hansen, Sriram Ashokkumar, Stefan Caldararu, Abhiraj Dashora, Ishaan Mahajan, Harry Zhang, Luning Fang, He Shen, Xiangru Xu, Radu Serban, and Dan Negrut. Art/atk: A research platform for assessing and mitigating the sim-to-real gap in robotics and autonomous vehicle engineering, 2022.

[12] Asher Elmquist, Aaron Young, Ishaan Mahajan, Kyle Fahey, Abhiraj Dashora, Sriram Ashokkumar, Stefan Caldararu, Victor Freire, Xiangru Xu, Radu Serban, and Dan Negrut. A software toolkit and hardware platform for investigating and comparing robot autonomy algorithms in simulation and reality. *arXiv preprint arXiv:2206.06537*, 2022.

[13] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[14] Gutemberg Guerra-Filho. Optical motion capture: Theory and implementation. *RITA*, 12(2):61–90, 2005.

[15] W. Hirschberg, G. Rill, and H. Weinfurter. Tire model tmeasy. *Vehicle System Dynamics*, 45(sup1):101–119, 2007.

[16] Sebastian Höfer, Kostas Bekris, Ankur Handa, Juan Camilo Gamboa, Melissa Mozifian, Florian Golemo, Chris Atkeson, Dieter Fox, Ken Goldberg, John Leonard, et al. Sim2real in robotics and automation: Applications and challenges. *IEEE transactions on automation science and engineering*, 18(2):398–400, 2021.

[17] Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *European Conference on Artificial Life*, pages 704–720. Springer, 1995.

[18] Steven Keeping. Understanding the advantages and disadvantages of linear regulators,". *Electronic Product, Digikey Article Library*, 2012.

[19] Marc C Kennedy and Anthony O'Hagan. Bayesian calibration of computer models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(3):425–464, 2001.

[20] 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.

[21] Laëtitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. Reward function and initial values: Better choices for accelerated goal-directed reinforcement learning. In *Artificial Neural Networks–ICANN 2006: 16th International Conference, Athens, Greece, September 10-14, 2006. Proceedings, Part I 16*, pages 840–849. Springer, 2006.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[24] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464*, 2018.

[25] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *The Journal of Machine Learning Research*, 22(1):12348–12355, 2021.

[26] Project Chrono Development Team. PyChrono: A Python wrapper for the Chrono multi-physics library. `https://anaconda.org/projectchrono/pychrono`. Accessed: 2023-01-14.

[27] UW-Madison Simulation Based Engineering Laboratory. Autonomy Toolkit. `http://projects.sbel.org/autonomy-toolkit/`, 2022.

[28] UW-Madison Simulation Based Engineering Laboratory. PyPI: Autonomy-ToolKit. `https://pypi.org/project/autonomy-toolkit/`, 2022.

[29] ros drivers. mocap_optitrack. https://github.com/ros-drivers/mocap_optitrack/tree/foxy-devel, July 2022.

[30] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[31] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[32] R. Serban, M. Taylor, D. Negrut, and A. Tasora. Chrono::Vehicle template-based ground vehicle modeling and simulation. *Intl. J. Veh. Performance*, 5(1):18–39, 2019.

[33] Ganzi Suresh. *Design Optimization of a Wishbone Suspension of a Passenger Car.* PhD thesis, Jawaharlal Nehru Technological University Ananthapuram, 12 2013.

[34] A. Tasora, R. Serban, H. Mazhar, A. Pazouki, D. Melanz, J. Fleischmann, M. Taylor, H. Sugiyama, and D. Negrut. Chrono: An open source multi-physics dynamics engine. In T. Kozubek, editor, *High Performance Computing in Science and Engineering – Lecture Notes in Computer Science*, pages 19–49. Springer International Publishing, 2016.

[35] Alessandro Tasora, Dario Mangoni, Dan Negrut, Radu Serban, and Paramsothy Jayakumar. Deformable soil with adaptive level of detail for tracked and wheeled vehicles. *International Journal of Vehicle Performance*, 5(1):60–76, 2019.

[36] H. Unjhawala, R. Zhang, W. Hu, J. Wu, R. Serban, and D. Negrut. Using a Bayesian-inference approach to calibrating models for simulation in robotics. *Journal of Computational and Nonlinear Dynamics*, 18(6), 04 2023. 061004.

[37] Huzaifa Unjhawala, Thomas Hansen, Luning Fang, Shouvik Chatterjee, Zhenhao Zhou, Jinlong Wu, Radu Serban, and Dan Negrut. A fast and reliable reduced order model for large scale vehicle simulation. In *ECCOMAS Multibody Dyanmics Conference*, 2023.

[38] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35:24611–24624, 2022.

[39] Jing-Shan Zhao, Xiang Liu, Zhi-Jing Feng, and Jian S Dai. Design of an ackermann-type steering mechanism. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 227(11):2549–2562, 2013.