Development of an image-based autonomous driving system for an e-FSAE

31.58 Proyecto Mecatrónico

Manuel Valentin	Agustín Grillo
mvalentin@itba.edu.ar	*agrillo@itba.edu.ar*

Abstract

This thesis describes the research and implementation of an image-based self-driving algorithm, capable of commanding a vehicle around an unknown track. Through the use of Artificial Intelligence and simulation, various modern approaches are covered, following a completely different paradigm with respect to classical path planning methods. The idea of selecting models fully developed in simulation and deploying them on real life platforms is pursued as a way of accelerating data acquisition and validation of models. A modular software architecture is proposed, which allows hardware abstraction and addition of new features and algorithms. Finally, to validate models in real life scenarios, a robot was designed and engineered as a development platform.

We would like to thank our tutor, Dr. Ing. Pablo Leslabay for his supervision, support, and tutelage throughout the development of this project. Our gratitude also extends to our university, ITBA, for allowing us the opportunity to broaden our horizon in the robotics field as well as providing us with the skills required to face new challenges in the future. Additionally, we would like to thank Ing. Gabriel Torre for his support and guidance which was really influential in shaping this project. Our appreciation also goes to our families and friends for their encouragement and support all through our studies.

Introduction **4**

Formula Student **5**
Formula Student Rules: Static and Dynamic Event (DV) 6
Skidpad Event 6
Acceleration Event 7
Autocross/Sprint 7
Track Drive 8

State of the Art **9**
AMZ (Zurich) 9
MITeam Delft 10
EUFS 11
Formula Technion 13

Background **15**
Camera Technology Overview **15**
Pinhole Camera 15
Lenses and Aperture 17
Calibration 18
Image Rectification 20
Imagers and Bayern Pattern 20

Software Tools **21**
ROS and Gazebo 21
Simulator 22
Keras and TensorFlow 23

Control **23**
Introduction 23
Imitation Learning **24**
Introduction 24
Expert 25
Policy 26
Training 28
Simulation Results 29
Limitations 29
Reinforcement Learning **30**
Introduction 30
Environment 31

Perception (GAN) **31**
Introduction 31
Method Overview 34
Implementation 36
Partial results 40

Robot **41**

Design 41
Construction 42
Vehicle Dynamics and Control 44
Hardware 46
Camera Calibration 47
Simulation 49
Software and Control 50
Manual Mode 51
Driverless Mode 52
Simulation and Reality Comparison 53

Results **54**

Conclusion **57**

Future Projects/Improvement **57**

Bibliography **59**

Annex **61**
List of Materials 61

Introduction

In the past few years, there has been substantial advances in GPUs (graphics processing units) in terms of size reduction and higher processing speeds. This has allowed its use on various robot applications that aim to solve certain tasks through vision and Artificial Intelligence. Software has considerably improved and facilitated the way vision-based algorithms are developed, providing higher level programming and easy to use interfaces. The objective of this project is to take advantage of these tools and apply them in self-driving vehicles.

In the first section of this paper, the Formula Student international competition is described. It will provide context and display the challenges for this project. The objective, as described in the Autocross and Trackdrive event for the driverless category, is to **develop a self-driving algorithm capable of driving on an unknown track delimited by cones**. Research is carried out to identify the progress and status of other teams in pursuing this task as a way of gaining knowledge in state-of-the-art architectures and development tools.

A complete background on cameras is introduced in order to build the theoretical basis for the image-based approach chosen for this project. Concepts such as pixels, channels, and image rectification are presented to facilitate the understanding of the following sections. Furthermore, powerful development tools such as ROS, Gazebo, Machine Learning libraries, and simulators are presented as key aspects in building a modular software architecture for this project.

In the control section, an Imitation Learning approach is proposed, where a Deep Neural Network seeks to **mimic human** behavior while driving around a simulated track. Limitations of this approach give rise to the perception section, where the **simulation-to-reality gap is addressed**. This aims to obtain a robust driving policy trained solely in simulation. Finally, a **robot is engineered** to validate the developed models in real life scenarios. Details in construction and control of the robot are explained in this section.

Formula Student

Formula Student is an automotive engineering competition where teams around the world compete in a series of static and dynamic events. There are three categories in which the teams can participate including Internal Combustion Vehicles, Electric Vehicles and finally Driverless Vehicles. Formula Student is hosted by different countries all around Europe, all following the same set of rules.

Events:

- Formula Student UK (Silverstone) organized by IMECHE
- Formula Student Germany
- Formula Student Italy
- Formula Student Spain
- Formula Student East Hungary
- Formula Student Netherlands
- Formula Student Austria
- Formula Student Russia
- Formula Student Czech Republic

Figure 1. Formula Student Events [1]

For the purpose of our project, we consider that our vehicle complies with all the technical specifications, and therefore, we will only focus on the Driverless category and the rules regarding the specific static and dynamic events of that category. Inside the Driverless Category, there are two classes, Dynamic Driving Task (DDT) and Automated Driving System (ADS). In the DDT class, teams will have an opportunity to purchase an Automated Driving Systems Dedicated Vehicle (ADS-DV) and integrate an AI computer and sensors to the platform with its respective software. In the ADS class, teams are allowed to use an existing Formula Student Vehicle and adapt it for autonomous driving.

Formula Student Rules: Static and Dynamic Event (DV)

Static Events:

- Business Plan Presentation Event (S1)
- Static Design Event Class ADS and DDT (S3)
- Real World Event (S4)

Dynamic Events:

- Skidpad Event (D4)
- Acceleration Event (D5)
- Autocross / Sprint (D6)
- Track Drive (D8)

Skidpad Event

Layout

Each circle is marked with a line, outside the inner circle and inside the outer circle.

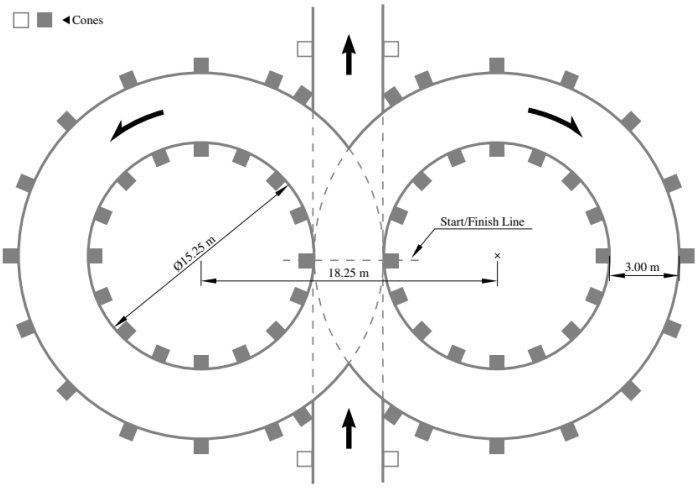

Figure 2. Skidpad Layout [2]

Basic Procedure (Details can be found on the Formula Student Rules 2020)

Each team has at least two runs. The line between the centers of the circles defines the start/finish line. A lap is defined as traveling around one of the circles, starting and ending at the start/finish line. The vehicle will enter perpendicular to the figure eight and will take one full lap on the right circle to establish the turn. The next lap will be on the right circle and will be timed. Immediately following the second lap, the vehicle will enter the left circle for the third lap. The fourth lap will be on the left circle and will be timed. Immediately upon finishing the fourth lap, the vehicle will exit the track. Finally, the vehicle will exit at the intersection moving in the same direction as entered and must come to a full stop within 25 m after crossing the timekeeping line, inside the marked exit lane.

Scoring

T_{team} is the team's best run time (run time is the average time of the timed left and timed right circle plus penalties which are added after the averaging). T_{min} is the fastest vehicle run time. T_{max} is 2*T_{min}.

$$SKIDPADSCORE = 75\left(\frac{\left(\frac{T_{max}}{T_{team}}\right)^2 - 1}{\left(\frac{T_{max}}{T_{min}}\right)^2 - 1}\right)$$

Acceleration Event

Layout

The acceleration track is a straight line with a length of 75 m from the starting line to the finish line. The track is at least 3 m wide. Cones are placed along the track at intervals of about 5 m. Cone locations are not marked on the pavement.

Basic Procedure

Each team has at least two runs. Time is taken between the crossing of the starting line and the finishing line. After crossing the finishing line, the vehicle must come to a full stop within 100 m inside the marked exit lane. Cone penalties will be applied as per Formula Student Rules.

Scoring

T_{team} is the team's best time, including penalties. T_{min} is the fastest vehicle time. T_{max} is 2*T_{min}.

$$ACCELERATION\ SCORE = 75\left(\frac{\frac{T_{max}}{T_{team}} - 1}{\frac{T_{max}}{T_{min}} - 1}\right)$$

Autocross/Sprint

Layout

The layout for this event is unknown, but the track is built based on the following guidelines:

- Straights: No longer than 80 m
- Constant Turns: up to 50 m diameter
- Hairpin Turns: Minimum of 9 m outside diameter (of the turn)
- Miscellaneous: Chicanes, multiple turns, decreasing radius turns, etc.
- The minimum track width is 3 m

Procedure

Each team has at least two runs consisting of one SINGLE lap. Using data collected in a previous run is not permitted for this event. After the run, the vehicle must come to a full stop within 30 m after the finish line.

Scoring

Tteam is the team's best run time, including penalties. Tmin is the fastest run time. Tmax is 2*Tmin.

$$AUTOCROSSSCORE = 75\left(\frac{\frac{T_{max}}{T_{team}}-1}{\frac{T_{max}}{T_{min}}-1}\right)$$

Track Drive

Layout

The track is an unknown closed loop circuit following the same guidelines as in the Autocross event. The length of one lap is approximately 200 m to 500 m.

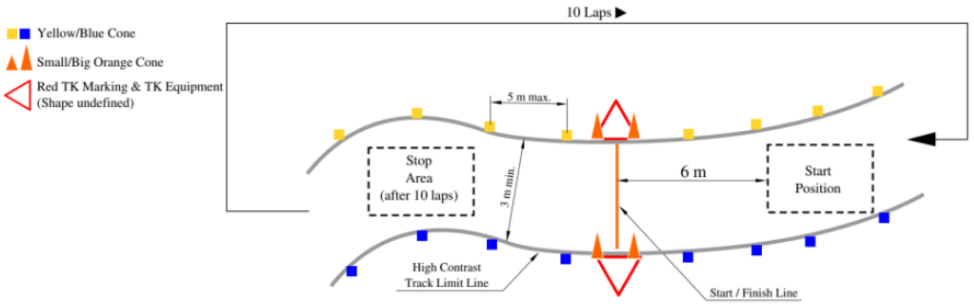

Figure 3. Trackdrive Overview [2]

Procedure

There will be at least one run consisting of TEN laps. Between each run, data stored must be deleted. After ten laps, the vehicle must come to a full stop within 30 m after crossing the finish line. Vehicle should count laps itself.

Scoring

Each lap of the Trackdrive event is individually timed. Ten points are awarded for every completed lap.

$$TRACKDRIVE\ SCORE = 150\left(\frac{\frac{T_{max}}{T_{team}}-1}{\frac{T_{max}}{T_{min}}-1}\right)$$

PENALTIES

Cones that are Down are not replaced during the run. There will be no re-run due to cones in the driving path or disorientation due to missing cones.

State of the Art

AMZ (Zurich)

The Academic Motorsport Club Zurich was founded by students of ETH Zurich, and since the introduction of the Formula Student Driverless class, they have been developing and improving autonomous driving for the purpose of this competition. AMZ has won first place at each competition they participated with excellent results. [3]

Figure 4. Zurich Racing Platform [4]

Regarding their autonomous racing platform, seen in figure 4, it is constituted by several software modules including environment perception and vehicle dynamics control. The perception unit uses LiDAR, vision, or both; following an algorithm to detect cones, estimate their 3D poses, and colors. This algorithm is based on a modified version of YOLO. For localization and mapping, they utilize SLAM, as racetracks are initially unknown. This way a map is provided including the boundaries of the racetrack. This entire effort is made in order to implement trajectory planning and control techniques that will make the car go even faster. It is accomplished through a Model Predictive Control (MPC) to find an optimal trajectory considering the dynamics of the vehicle.

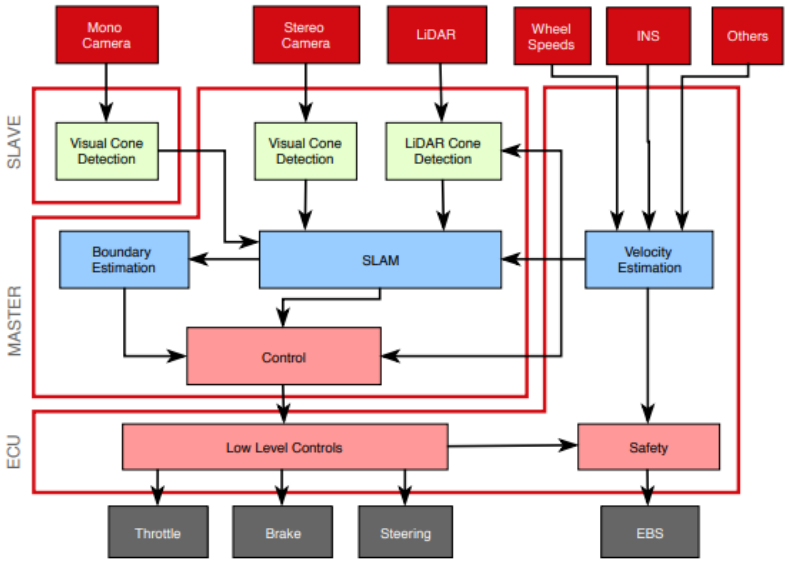

Figure 5. Control Architecture [5]

Regarding sensors used for the perception module, they use three CMOS cameras with global shutter in a stereo and mono setup for detecting cones at various distances. A diagram showing other sensors implemented for autonomous driving and their location in the vehicle are shown below.

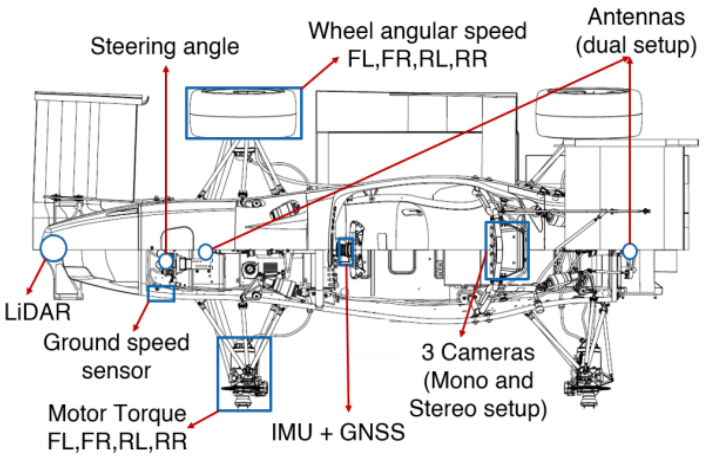

Figure 6. Vehicle Setup [5]

MITeam Delft

MITeamDelft is composed of two universities: MIT (USA) and Delft University of Technology (Netherlands). The team got podium finishes at all Formula Driverless competitions in which it raced (Formula Student Germany 2018 and 2019 and Formula Student Italy).

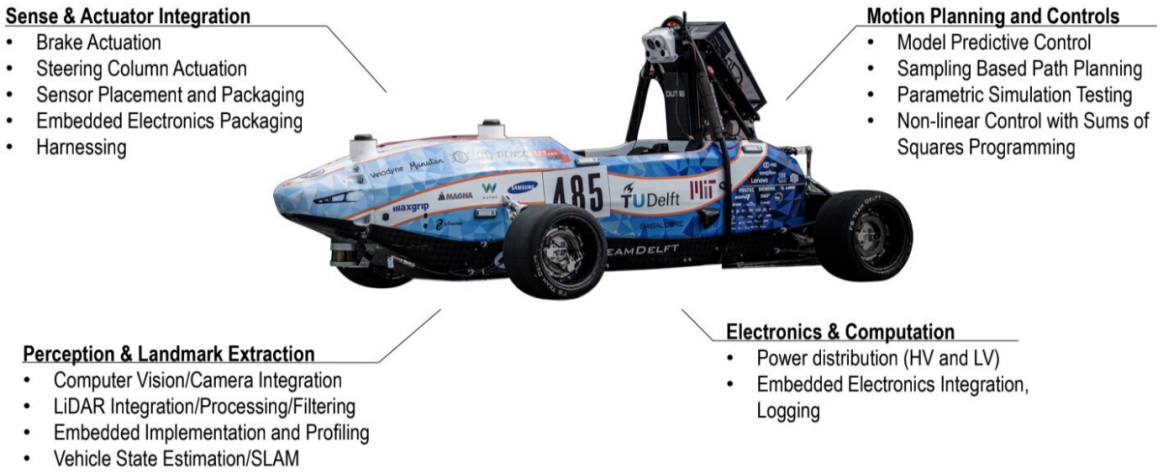

Figure 7. MITeamDelft Platform [6]

In 2018, they used a camera-based perception system, with a LIDAR to improve safety. More precisely, a stereo and a monocular camera for long-range and short-range detection. Both output images were processed by an NVIDIA Jetson Xavier developer kit, through a YOLOv3 (You only Look Once) algorithm.

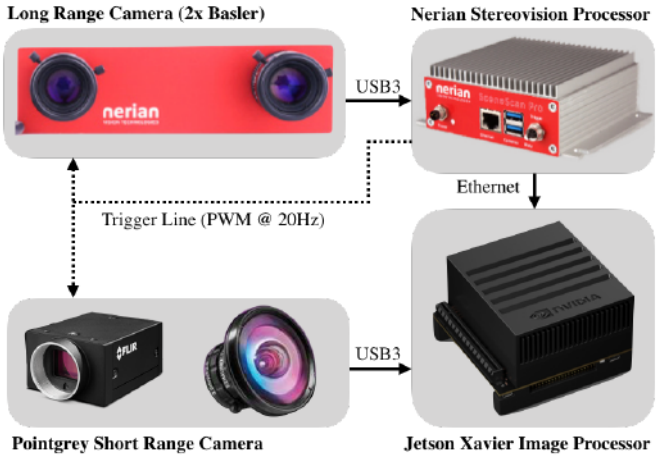

Figure 8. Sensor Setup [6]

In 2019, they used LIDAR-based perception system. For object and cone color detection with the LIDAR data, they use a Point-Voxel CNN (PV CNN) for efficient and fast 3D deep learning. This method is memory and computation efficient.

EUFS

The Edinburgh University Formula Student was established by a group of undergraduate students of various disciplines at the University of Edinburgh. There is a designated AI team looking to adapt a previous electric vehicle into a self-driving race car to participate in the Formula Student Driverless. Meanwhile, they have been participating in the DDT class where the vehicle is provided by the

competition, focusing only on the software. They won both competitions in 2018 and 2019. Following a similar architecture as in the other teams, they have different subteams that focus on the modules of perception, localization, mapping and planning. They also focus on building a software infrastructure in order to develop the car through the usage of simulations and automated testing. Also, for software evaluation in real world scenarios, they have developed a cheap testbed platform based on an electric wheelchair.

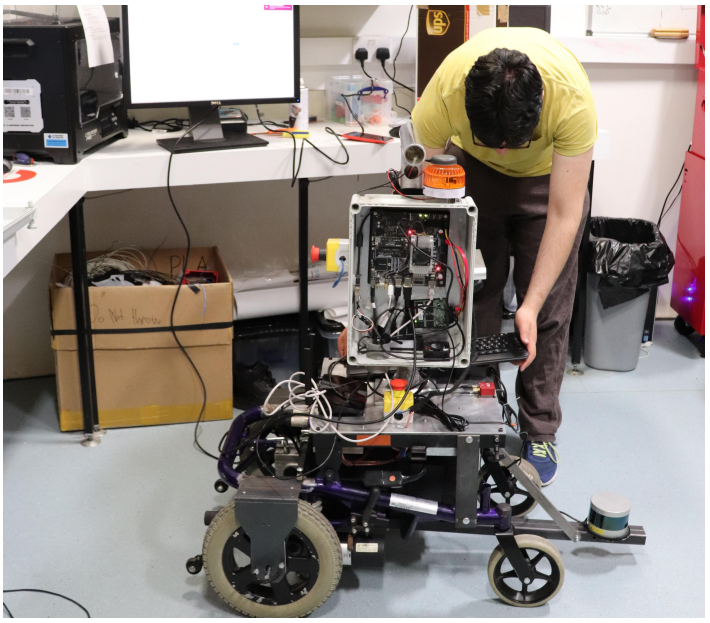

Figure 9. EUFS Development Platform [7]

Figure 10. Testing of Platform [7]

In parallel with the architecture mentioned above, they also proposed an end-to-end learning using deep reinforcement learning algorithms. This approach avoids the long pipeline architecture, decreasing the number of potential breakpoints. Raw input data from the sensors, especially a camera, is fed to a deep neural network, which decides an output command for the car. For this, the algorithm must train itself by trial and error, and as this can take long hours, they use a simulator (GAZEBO) to

replicate the environment. This approach requires a lot of computational power during training, but the algorithm can learn on its own and can achieve super-human behavior.

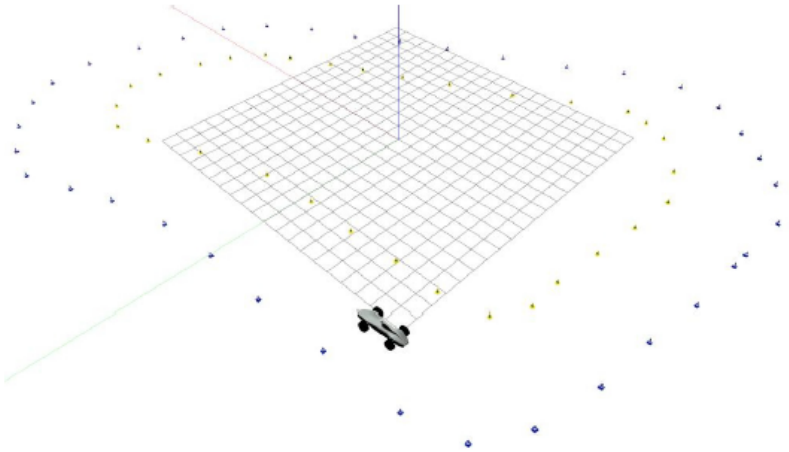

Figure 11. EUFS Simulator [7]

Formula Technion

The Formula Technion's electric driverless is a project where students from the Israel Institute of Technology of different expertise join to design, develop and build an electric-driverless race car. Alongside the classic approach followed by leading teams such as AMZ, they have been developing a Machine Learning based algorithm. To carry this out, they have developed an AI research platform to experiment with deep learning, specifically Imitation Learning. By steering the car manually in the simulation, they are able to record data. This information is then used to train a Deep Learning model, which steers the vehicle based on a single image.

Figure 12. Formula Technion's simulator [8]

Figure 13. Simulator's command [8]

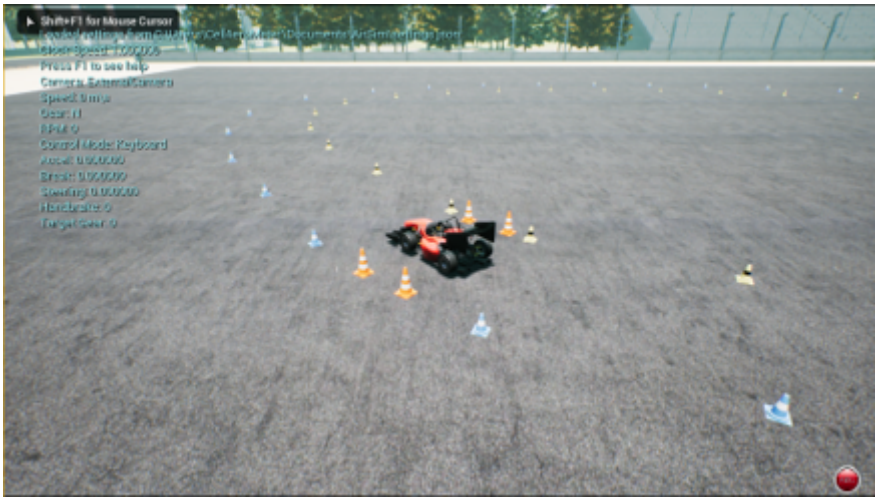

Figure 14. Example view of the simulator [8]

Background

Camera Technology Overview

Pinhole Camera

The pinhole camera is the simplest model of a camera, consisting of a tiny aperture in which light from the environment passes through and hits an Image Plane, forming an inverted image. In this model, lenses are not considered. The image plane is found at a distance f along the optical axis, also known as the focal distance of the camera. In figure 15 we see a visual representation of the different frames involved in this model and the projection of a certain Point P in the image plane.

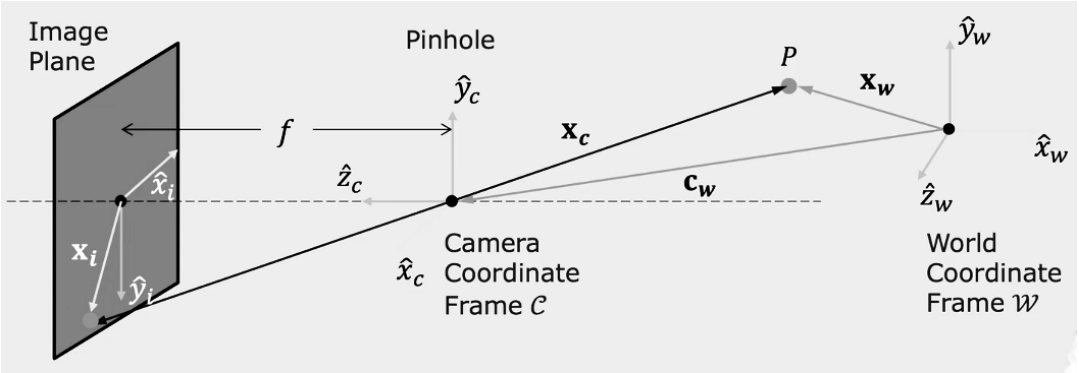

Figure 15. Pinhole Camera Model [9]

To pass from a Point P in the world coordinate frame to image coordinates, first there is a **coordinate transformation** in which Point P is found with respect to the camera coordinate frame. And secondly, a **perspective projection** to find the image coordinates in the image plane. It's important to remember that until this point all variables have a distance unit and pixels are introduced in the next step.

$$\underset{\text{Coordinates}}{\text{Image}} \qquad \underset{\text{Coordinates}}{\text{Camera}} \qquad \underset{\text{Coordinates}}{\text{World}}$$

$$X_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad \Leftarrow \quad X_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \quad \Leftarrow \quad X_w = \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix}$$

Real cameras obtain the information in the image plane in a discrete fashion, where the image plane is composed of a certain number of pixels both in x and y coordinates. The origin of the u, v frame is translated to have all pixel coordinates as positive.

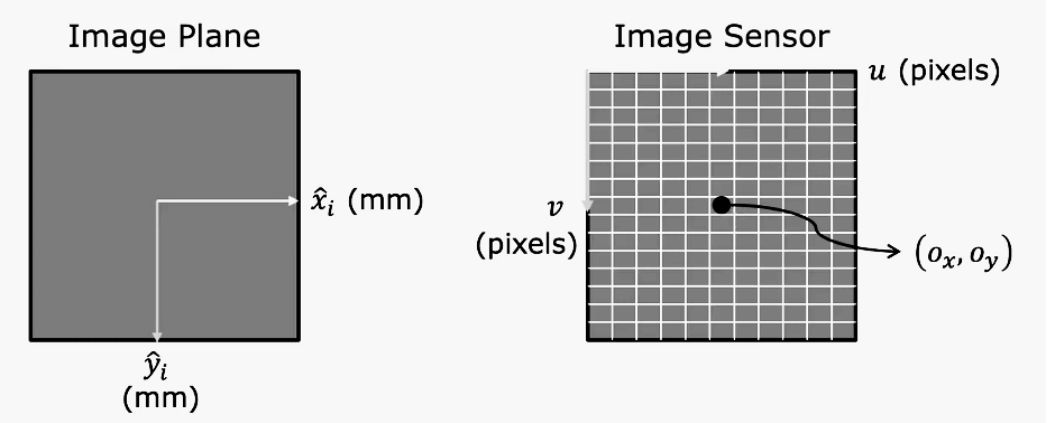

Figure 16. Discretization of Image Plane [9]

The following equations express the relation between a Point P in the camera coordinate frame and its projection in pixels in the image sensor. Both m_x and m_y corresponds to the number of pixels per unit distance [pixels/mm] in the horizontal and vertical direction, respectively. Combining these densities, with the focal distance, the focal distances in x and y are obtained. The point (o_x, o_y) in pixels is known as the principal point and marks the center of the optical axis.

$$u = m_x \cdot f \cdot \frac{x_c}{z_c} + o_x \qquad v = m_y \cdot f \cdot \frac{y_c}{z_c} + o_y$$

$$u = f_x \cdot \frac{x_c}{z_c} + o_x \qquad v = f_y \cdot \frac{y_c}{z_c} + o_y$$

The equations presented above are nonlinear, and it is therefore not possible to represent them in matrix form. Fortunately, using homogeneous coordinates, where basically a fictitious dimension is added, it is possible to rewrite the equations in matrix form.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

$$\tilde{u} = u \cdot \tilde{w}$$
$$\tilde{v} = v \cdot \tilde{w}$$
$$\tilde{w} = z_c$$

On the other hand, the matrix in charge of transforming Point P from the world coordinate frame to the camera coordinate is composed of a rotation matrix R and a translation vector t, shown below in the World to Camera section.

Camera to Pixel

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

World to Camera

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

The matrix in charge of the perspective projection is called the intrinsic matrix of the camera and plays an important role in the calibration and rectification of a camera. The matrix that takes care of the coordinate transformation is called the Extrinsic matrix. Combining both matrices, we obtain a single Projection matrix that maps a point P in the world frame to a pixel coordinate in an image.

$$\tilde{u} = M_{int} \cdot \tilde{x}_c \qquad \tilde{x}_c = M_{ext} \cdot \tilde{x}_w$$

$$\tilde{u} = M_{int} M_{ext} \tilde{x}_w = P \tilde{x}_w$$

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

Lenses and Aperture

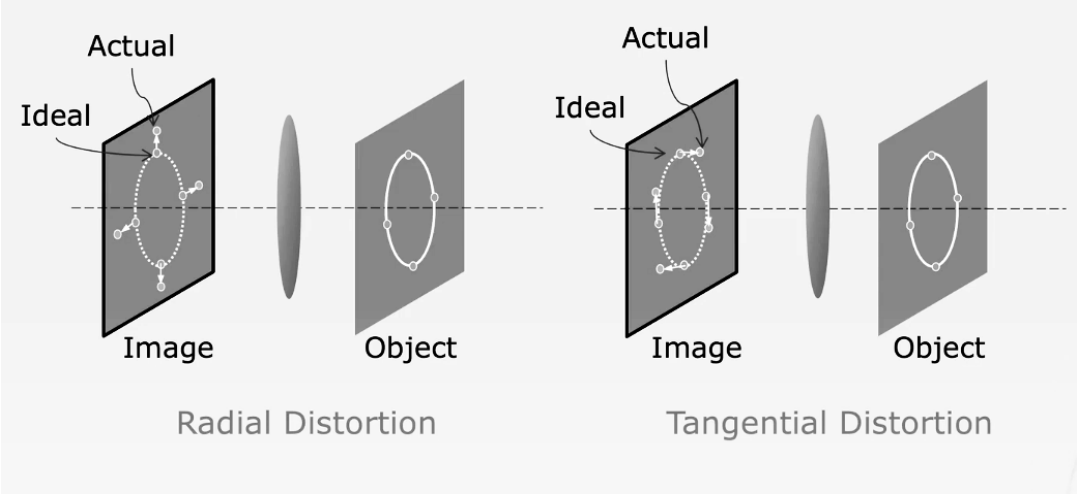

Figure 17. Types of Distortion [9]

The intrinsic model of the camera presented above does not consider distortions introduced by lenses which are commonly used in a camera setup in order to increase its field of view. In the calibration section of our hardware, a more complete model of the camera is presented, which adds a distortion vector to account for these factors. Depending on the lens type, there are two essential types of distortions, radial and tangential, as seen on figure 17. In real applications, the most common is the

radial distortion. It is caused by the focal length of the lens not being uniform over its diameter. This means that the magnification effect (i.e., the ratio between an object's size when projected on a camera sensor versus its size in the real world) of the lens changes depending on the distance between the optical axis and the ray of light passing through the lens. If the magnification decreases, it is called **barrel distortion** and if it increases, the resulting distortion effect is called **pin cushion distortion.** See figure below for a visual representation of these effects.

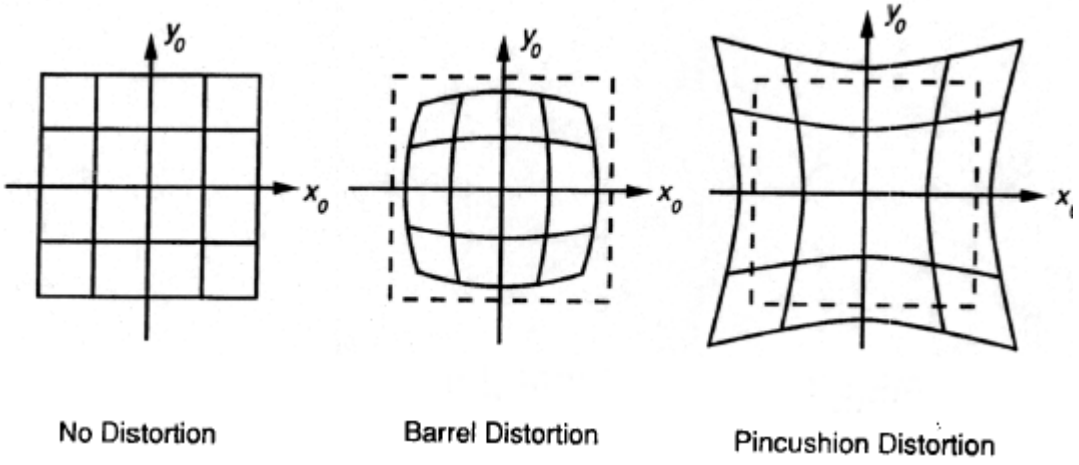

Figure 18. Radial Distortion Effect [10]

Calibration

The objective of a camera calibration is to obtain the unknown projection matrix, which will then be used to extract the intrinsic parameters of the camera in order to perform image rectification. In this section, one method to carry on a calibration is presented, which will have some differences with the one that was carried with the real hardware but both share the same concept.

$$\begin{bmatrix} u^{(i)} \\ v^{(i)} \\ 1 \end{bmatrix} \equiv \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_w^{(i)} \\ y_w^{(i)} \\ z_w^{(i)} \\ 1 \end{bmatrix}$$

Known Unknown Known

A calibration requires a target whose geometry is known beforehand and contains a pattern that is easily detected by computer vision algorithms. A chessboard is the most common target employed, see figure 19 where a cube with chess boards on its sides is shown.

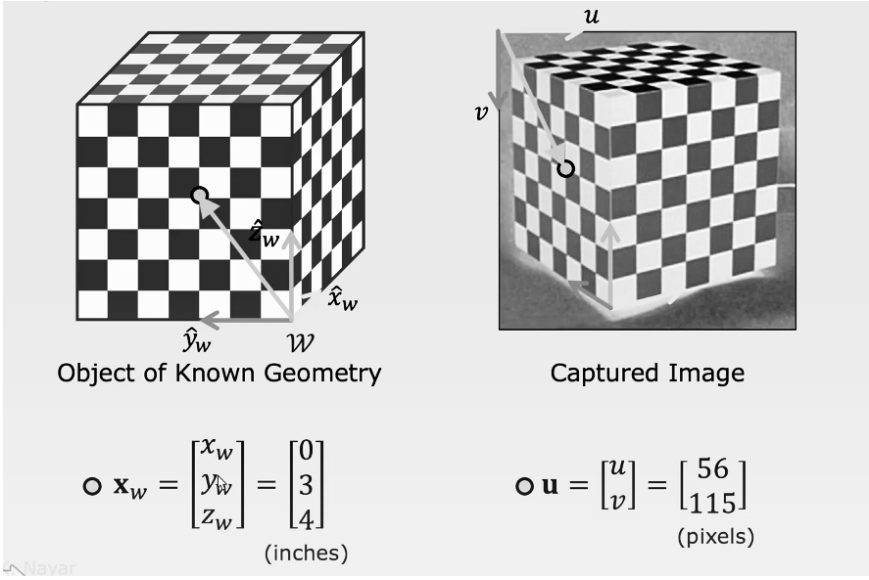

Figure 19. Target Calibration [9]

If the world frame is set in one of the corners of the cube, then the corresponding coordinate of each corner of the chessboard can be easily known as well as its corresponding pixel using simple computer vision algorithms which searches for corners in the captured image. With a single point, two equations (one for each pixel) are obtained.

$$u^{(i)} = \frac{p_{11} \cdot x_w^{(i)} + p_{12} \cdot y_w^{(i)} + p_{13} \cdot z_w^{(i)} + p_{14}}{p_{31} \cdot x_w^{(i)} + p_{32} \cdot y_w^{(i)} + p_{33} \cdot z_w^{(i)} + p_{34}}$$

$$v^{(i)} = \frac{p_{21} \cdot x_w^{(i)} + p_{22} \cdot y_w^{(i)} + p_{23} \cdot z_w^{(i)} + p_{24}}{p_{31} \cdot x_w^{(i)} + p_{32} \cdot y_w^{(i)} + p_{33} \cdot z_w^{(i)} + p_{34}}$$

Using N points found in the captured image, a system of equations can be built where the vector p is unknown. Using algebraic methods with some constraints in the solution, it is possible to find the vector p using eigenvalues and eigenvectors.

$$\begin{bmatrix} x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & 0 & 0 & 0 & 0 & -u_1 \cdot x_w^{(1)} & -u_1 \cdot y_w^{(1)} & -u_1 \cdot z_w^{(1)} & -u_1 \\ 0 & 0 & 0 & 0 & x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & -v_1 \cdot x_w^{(1)} & -v_1 \cdot y_w^{(1)} & -v_1 \cdot z_w^{(1)} & -v_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & 0 & 0 & 0 & 0 & -u_i \cdot x_w^{(i)} & -u_i \cdot y_w^{(i)} & -u_i \cdot z_w^{(i)} & -u_i \\ 0 & 0 & 0 & 0 & x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & -v_i \cdot x_w^{(i)} & -v_i \cdot y_w^{(i)} & -v_i \cdot z_w^{(i)} & -v_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & 0 & 0 & 0 & 0 & -u_n \cdot x_w^{(n)} & -u_n \cdot y_w^{(n)} & -u_n \cdot z_w^{(n)} & -u_n \\ 0 & 0 & 0 & 0 & x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & -v_n \cdot x_w^{(n)} & -v_n \cdot y_w^{(n)} & -v_n \cdot z_w^{(n)} & -v_n \end{bmatrix} \begin{bmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

$$Ap = 0$$

Once the matrix components of p are obtained, the only thing left is to extract the intrinsic parameters of the camera. Recalling the definition of p which is the combination of both the extrinsic and intrinsic matrices.

$$\begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = KR$$

Given that K is an Upper Right Triangular matrix and R is an orthonormal matrix, it is possible to uniquely decouple K and R from their product using QR factorization and therefore obtain the intrinsic parameters of the camera.

Image Rectification

The process of removing distortions from a camera image is called *rectification*. Later in the real camera calibration section, a more complex model of the camera is used which includes a distortion vector. Using these parameters, it is possible to take the original image and rectify it as shown in the image below. The model corresponding to the rectified image in theory should contain a null distortion vector.

Figure 20. Image Rectification [9]

When using deep learning based on a set of trained weights, it makes sense to rectify the image first before feeding it to a network. If the original image is employed, distortions (such as from different lenses) would lead to detection errors, as networks are usually trained on a distortion-free image set.

Imagers and Bayern Pattern

As mentioned in the previous section, real cameras obtain the information in the image plane in a discrete fashion using light sensitive elements. These elements sense the amount of light they receive and output a corresponding number of electrons. The generated electrons are converted into a voltage

in which an Analog to Digital converter takes care of transforming it into a discrete number. Each of these elements will correspond to a pixel.

To obtain color vision, tiny filter elements are placed in front of each pixel in order to obtain a measurement for a certain wavelength. The most common type of filter is RGB (Red, Green, Blue), therefore it is possible to obtain three individual images, one for each primary color also known as "channels" of an image. Each pixel is coded with 8 bits (i.e., 256 values) which allows for 16.7 million different color combinations.

The most common way of arranging the RGB filters is called a *Bayer pattern*, which has alternating rows of red-green and green-blue filters. The human eye is more sensitive to green, therefore the Bayer array has twice the amount of green filters. See figure 21 for more details of the Bayer pattern.

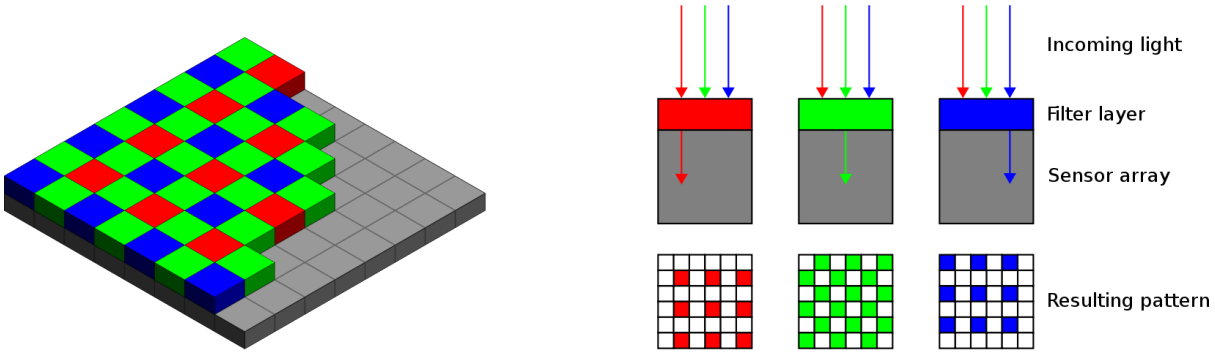

Figure 21. Bayer Pattern and Filter Layers [11]

Software Tools

ROS and Gazebo

The software developed in this project was built under the ROS framework. ROS (Robot Operating System) provides a framework for building robot applications, as well as providing a set of common software tools available as modular packages for any project. This allows the foundations for building a modular project and easy scalability for all levels of complexity.

Some of its main features and advantages are:

- Popular Open-Source robotics framework
- Built on top of Ubuntu Linux
- Hardware Abstraction
- Visualization
- Low-Level Device Control
- Message passing between processes
- Package Management
- Modular and Flexible
- Support code reuse

ROS is frequently updated and over the years many distributions have been launched, including improvements and new features. This project was developed under the **melodic** distribution, being the latest version at the time of starting the robot development.

ROS uses a node base architecture that allows different processes to run asynchronously and exchange predefined messages through what are called topics. A node can then be configured to subscribe to a certain topic, process the message, and then publish another topic for another node to use. As a result, a network of interconnected nodes is formed as shown on figure 22 (Circles are Nodes and Squares are topics).

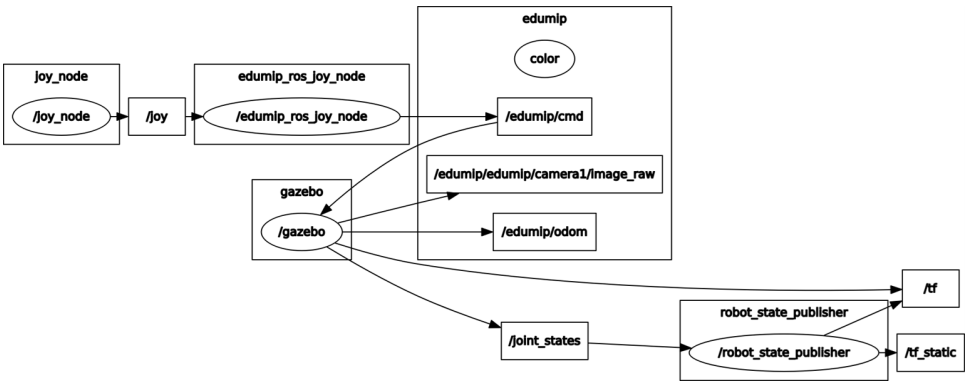

Figure 22. ROS Node Network [12]

Gazebo, on the other hand, is an open-source robot simulator that is well integrated with ROS. It allows the user to simulate various sensors such as a Lidar, a camera or IMU, between others. Thanks to its integration with ROS, it is possible to get sensor data through ROS topics and therefore test your software in a controlled environment without the need of real sensors.

Simulator

A simulator developed by ITBA was used as the start point for this project [13]. This simulator served as an entry to ROS framework and usage of Gazebo, as well as a benchmark for testing and validating the possibility of driving between cones using only a monocular camera. New features were added such as a track editor, control of the vehicle using a joystick and basic tools for developing Imitation and Reinforcement Learning algorithms.

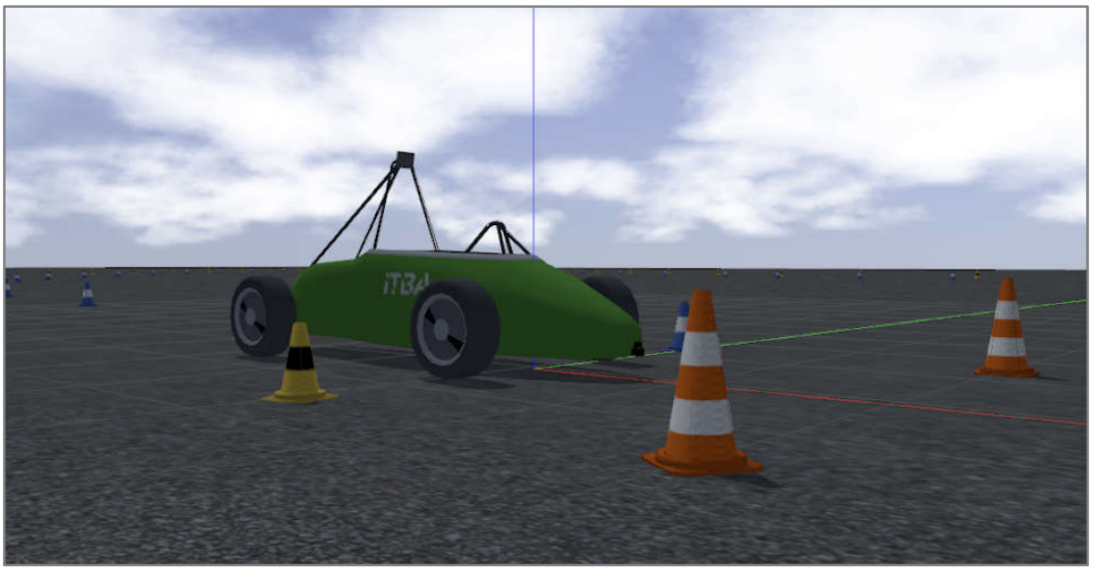

Figure 23. ITBA Simulator [13]

Keras and TensorFlow

During the project, various Machine Learning related tasks were performed. For this, an API named Keras [14] was used, which is built on top of TensorFlow [15] and uses it as the backend. Keras facilitated the development, training and implementation of ML models. As the project was developed under ROS Melodic, which is only compatible with Python 2.7, the first version of TensorFlow was used.

Control

Introduction

To autonomously drive the formula student, a Machine Learning approach was elected. The main idea is to map the vehicle state (s) to the corresponding action (a), via a policy (π).

$$\pi(s) = a$$

In this work, the state s consists of an RGB image obtained by a monocular camera containing the track ahead, the action a consists of the commanded vehicle velocity and steering, and the policy π_Θ is parametrized via a Neural Network with the corresponding parameters (weights) Θ.

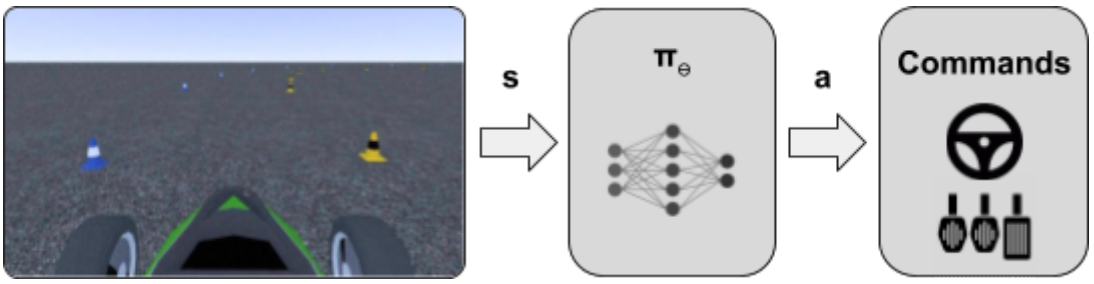

Figure 24. Simple control block diagram.

The objective is to optimize the parameters of the Neural Network, that successfully map the actions from the corresponding state. For this, the proposed path was first to train (optimize) the policy via Imitation Learning to validate the workflow, and later by Reinforcement Learning, to further improve the driving controller.

Imitation Learning

Introduction

Imitation Learning (IL) is a technique that aims to mimic an expert's (usually a human) behavior in a specific task, via demonstration. The idea is that the expert provides the algorithm with a set of "optimal" state-action pairs, from which the policy can be later trained. There are different IL methods to train the policy, some of which are Behavioral Cloning, Direct Policy Learning and Inverse Reinforcement Learning, among others [16].

In this first part of the work, Behavioral Cloning [17] was used, as being the simplest, but still effective, approach. The method consists of learning the expert's "optimal" policy (π^*) using Supervised Learning.

As from a set of human demonstrations $\xi = \{\xi_1, \xi_2, \dots\}$, a dataset is created. Each ξ_i is a trajectory taken by the expert, consisting of N_i consecutive states s_t, paired with their corresponding optimal action a_t^*, being t the respective step:

$$\xi_i = \left\{(s_1, a_1^*), (s_2, a_2^*), \dots, (s_N, a_N^*)\right\}$$

Where:

$$a^* = \pi^*(s)$$

The objective is to train the driving policy to imitate the human behavior over the whole dataset:

$$\pi_\theta^*(s) = a \approx a^*$$

$$s, a^* \in \xi$$

To do this, a loss function that compares the difference between $\pi^{\Theta}(s)$ and a^* is minimized.

$$\pi_{\theta}^{*} = \underset{\theta}{argmin}\ Loss(a^{*}, \pi_{\theta}(s))$$

Being:

$$Loss(a^{*}, \pi_{\theta}(s)) = \underset{s, a^{*} \sim \xi}{E} f(a^{*}, \pi_{\theta}(s))$$

Where the f function represents any metric, such as Mean Squared Error (MSE) or Mean Absolute Error (MAE), between others.

Expert

As mentioned earlier, expert demonstrations are necessary to train the agent. These are supplied by a human, who drives the simulated formula around different tracks. Each lap around a track corresponds to a different trajectory.

Therefore, by having *n* tracks, the dataset is composed by:

$$\xi = \{\xi_1, \xi_2, \ldots, \xi_n\}$$

During data collection, the expert commands the vehicle with a joystick, while the corresponding state-action pairs (s_t, a_t*) are being recorded at a frequency of 10 Hz.

As mentioned above, the state is composed of a single 640×320 RGB image, which shows a snapshot of the track ahead of the vehicle. It was assumed that this was sufficient for the policy to be capable of self-driving at low speeds, if at least one cone is always present in the image. This has various limitations, but was a first instance election.

On the other hand, the action consists of the steering angle and velocity of the formula:

$$a = (steer, velocity)$$

The steering accepts values from -1 to 1, being -1 full steer to the left, and 1 full steer to the right.

$$steer \in [-1, 1]$$

The velocity accepts values from $-V_{Max}$ to V_{Max}, where V_{Max} is the maximum desired speed, which was $10\ \frac{m}{s}$ in this case. A negative velocity implies the vehicle moving backwards, while a positive velocity implies a forward direction.

$$velocity \in [-10, 10]$$

To date, Machine Learning algorithms are highly data inefficient, which means they require big amounts of data to perform as expected. Therefore, in order to facilitate data acquisition, a track generator script, created by AMZ, was used. The used track editor is shown in the image below.

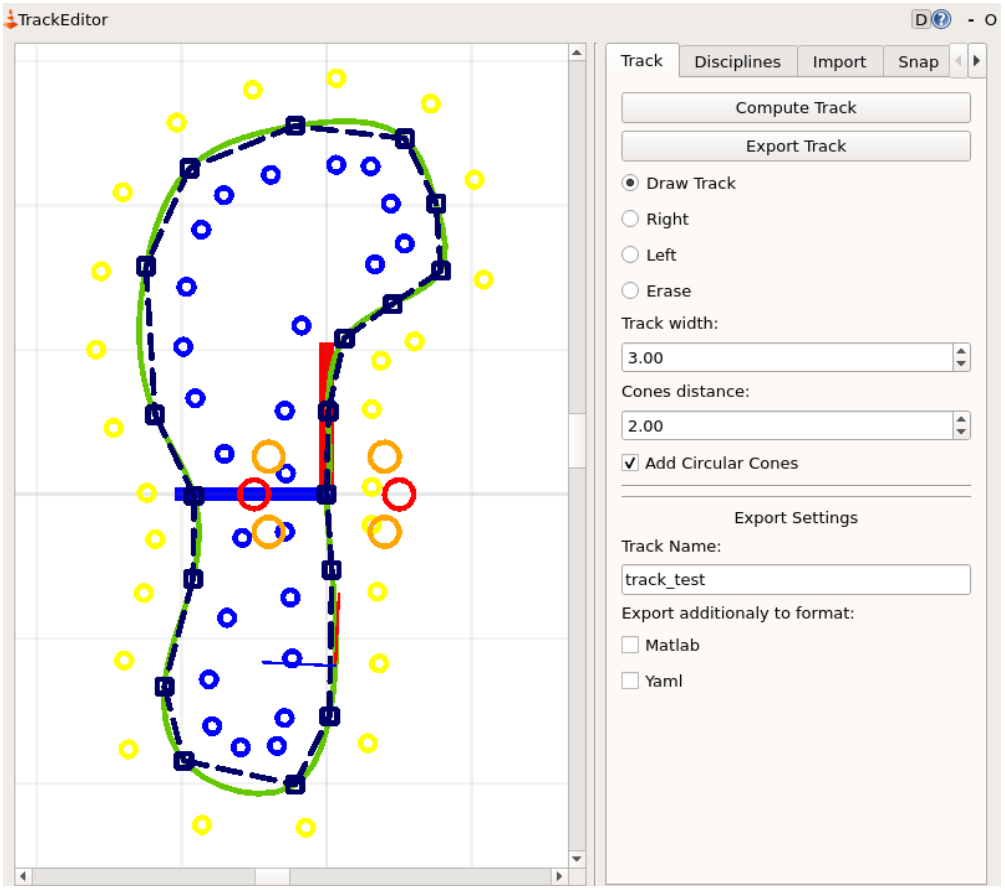

Figure 25. GUI used to generate and edit tracks.

It is presented in an intuitive GUI, which allows changing the track shape, and both the longitudinal and transversal distance between cones. This allowed the creation of multiple tracks, which increased variety in the dataset.

Policy

The policy (π_θ) is modeled with a Neural Network, which is mainly based on Nvidia's end-to-end self-driving car network architecture [18]. It is primarily composed of convolution blocks (c_n) and multilayer perceptron blocks (m_n).

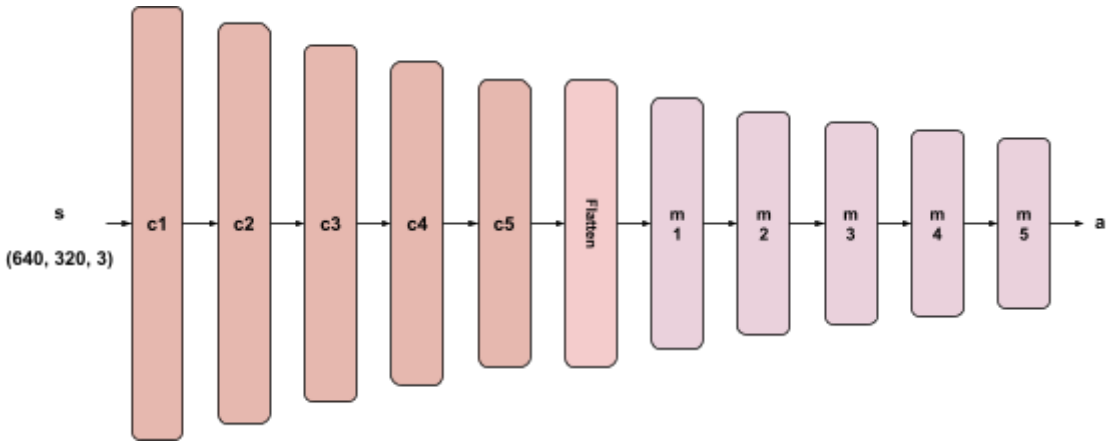

Figure 26. Policy Neural Network architecture.

Each convolutional block (c_n), consists of a convolutional network layer (with its respective hyperparameters as shown in the table below), followed up with a Rectified Linear Unit activation function (ReLU) and a 2D max pooling layer. The max pooling kernel has a size of 2 in both horizontal and vertical directions, and is used in all blocks except for the last (c_5).

Block	N° Filters	Kernel Size	Kernel Stride	Max Pooling?
$\mathbf{c_1}$	4	(5, 5)	2	Yes
$\mathbf{c_2}$	8	(5, 5)	2	Yes
$\mathbf{c_3}$	16	(3, 3)	1	Yes
$\mathbf{c_4}$	32	(3, 3)	1	Yes
$\mathbf{c_5}$	64	(3, 3)	1	No

Table 1. Hyperparameters for each convolutional block.

Each m_n block consists of a perceptron layer with different amounts of units, followed up with a ReLU activation function, except for the last block, which has a linear one.

Block	N° Units	Activation Function
$\mathbf{m_1}$	512	ReLU
$\mathbf{m_2}$	256	ReLU
$\mathbf{m_3}$	64	ReLU
$\mathbf{m_4}$	10	ReLU
$\mathbf{m_5}$	2	Linear

Table 2. Hyperparameters for each dense block.

Training

The first step before training was to normalize the dataset. This improves stability and convergence during optimization. As mentioned above, the state consists of an RGB image. Each pixel from each channel of the image has an 8-bit precision, which represents its color intensity from 0 to 255. Therefore, each image is then standardized to take values from -1 to 1. Being s_{ijk} a pixel from the i^{th} row, j^{th} column and k^{th} channel, and $s_{ijk}^{Norm.}$ the corresponding normalized pixel:

$$s_{ijk}^{Norm.} = 2\frac{s_{ijk}}{255} - 1$$

On the other hand, the actions are also normalized between -1 and 1. Being the normalized steering command:

$$steer^{Norm.} = steer$$

and the normalized velocity command:

$$velocity^{Norm.} = \frac{velocity}{10}$$

The minimization objective during training was the Mean Squared Error (MSE) between the predicted action and the expert's action.

$$Loss(a^*, \pi_\theta(s)) = MSE(a^*, \pi_\theta(s)) = E_{s, a^* \sim \xi}\left[(a^* - \pi_\theta(s))^2\right]$$

An Adam optimizer was used, with a learning rate of 0.0002.

The dataset was split randomly in 2 sets: A training and a validation set. The training set was used to optimize the policy parameters directly, while the validation set was used to monitor training and avoid overfitting.

As shown in the figure below, TensorBoard [15] was used to monitor the respective metrics during training. As can be seen, the left plot shows the mean squared error over the training data, along the different training epochs. The right plot, on the other hand, shows the mean squared error over the validation set. As from the 8th epoch, the mean squared error over the validation dataset begins to increase, whereas the one over the training set keeps decreasing. This indicates that the policy is starting to overfit the data, which means that if training continues, it will not generalize well over new, unseen scenarios. Therefore, the training is early stopped, and the final policy is the one with the lowest validation MSE.

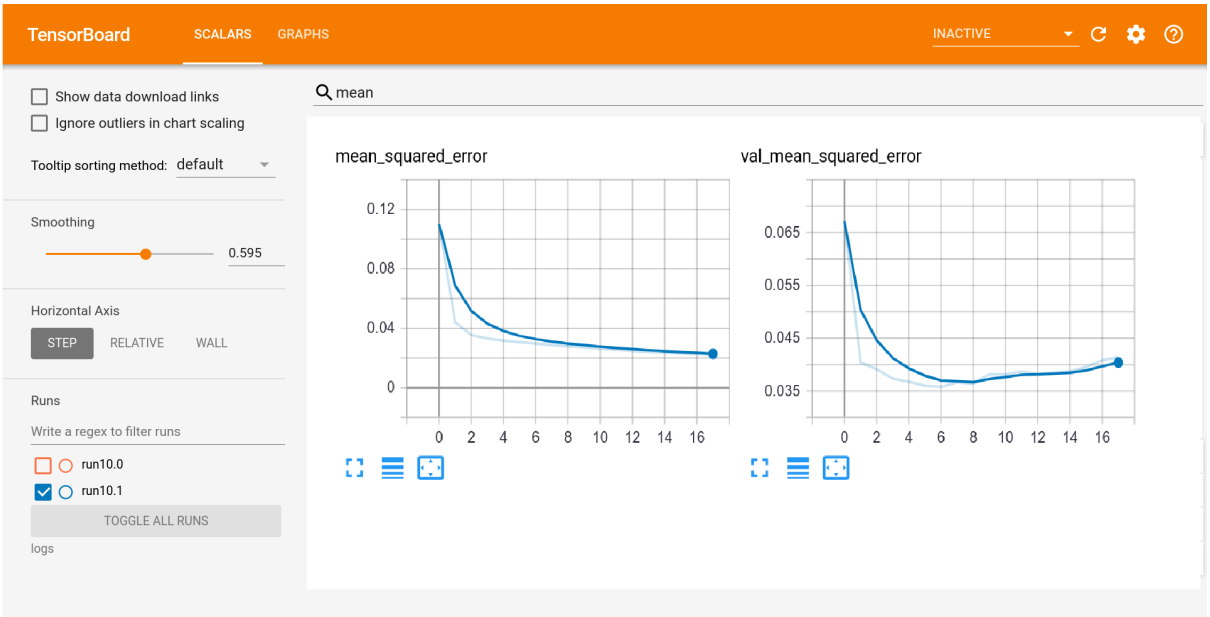

Figure 27. TensorBoard plots displaying the evolution of the metrics during training. The left plot ('mean_squared_error") shows the mean squared error over the training dataset for the different epochs. The right plot ('val_mean_squared_error") shows the mean squared error over the validation dataset for the different epochs.

Simulation Results

The final evaluation of the policy was performed using the simulation, in tracks that were not present in the dataset. As will be explained in more detail in the following section, the policy yielded promising results, although certain limitations were observed. More specifically, the policy was evaluated in 10 different unseen tracks. In 7 of these tracks, the controller was able to complete a full lap without going off the track; whereas in the remaining 3 tracks, the vehicle ended up off the track before completing the lap. It is interesting to note that the policy increases the vehicle's speed in the straight parts of the track, while decreasing it in the entrance of a curve, mimicking the human behavior.

Supplementary material showing the performance of the policy is available at the project's Git repository [*19*].

Limitations

Even though Behavioral Cloning yielded satisfactory results, they are far from being optimal. This is mainly because the expert's demonstrations are not uniformly sampled across the entire state space, as they are collected via different trajectories. In other words, the dataset does not contemplate every possible scenario. Therefore, when the agent visits a state that is not present in the dataset, the policy is prone to work poorly, leading to small errors. This is amplified by the nature of this type of sequential decision problems, where a bad action a_t in an unknown state s_t, leads to a new (usually unknown) state s_{t+1}. This generates a sequence of compound errors, which could lead to an unrecoverable state, as shown in the figure below.

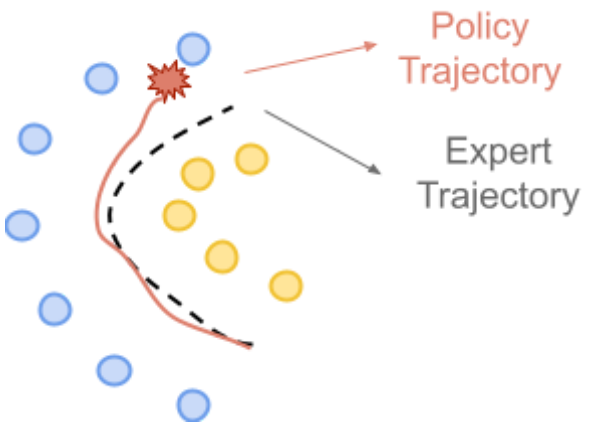

Figure 28. Scheme showing the difference between the trajectory taken by the expert, and the one taken by the policy.

The underneath cause is that Behavioral Cloning is basically a direct implementation of Supervised Learning over a set of sequential state-actions. Supervised Learning assumes that the data is sampled independently of each other, from an identical distribution (i.i.d). In this work, the data is not independent, as each state s_{t+1} depends on the previous state s_t and action a_t. This breaks the i.i.d assumption.

To overcome this issue, a Reinforcement Learning approach was proposed, which models the environment as a Markov Decision Process (MDP).

Reinforcement Learning

Introduction

Reinforcement Learning (RL) is an area of Machine Learning, in which an agent (the learner) interacts sequentially with an environment to achieve a goal. The agent, present in a specific state, continually selects a specific action, and in consequence, transitions into a new state. In each transition, the agent is given a reward by the environment, which the former tries to maximize over time [20].

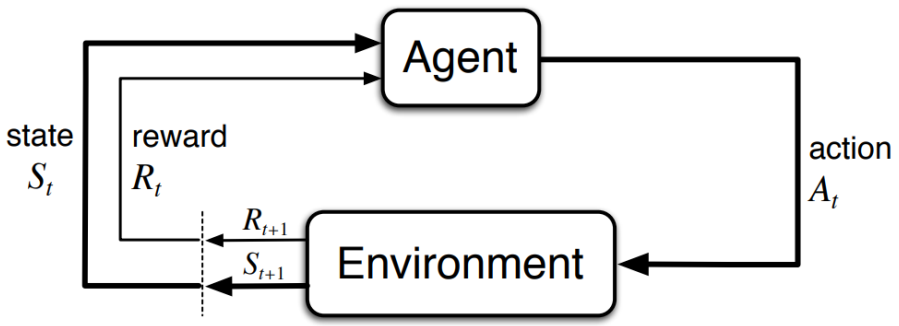

Figure 29. RL block diagram [20]

More specifically, the problem is modeled as a Markov Decision Process (MDP). An MDP consists of a set of states S, a set of actions A, a transition probability matrix $P_{s,a} = Pr(s_{t+1} \mid s_t, a_t)$ and a reward function $R(s_t, a_t)$. For each time step t, the agent present in a state $s_t \in S$ performs an action $a_t \in A$, transitioning into a new state $s_{t+1} \in S$ and receiving a reward R_{t+1}. The transition dynamics are described by the transition probability matrix $P_{s,a}$.

The objective is to find a policy $\pi(s) = a$, that maximizes the expected discounted sum of rewards G, over the episode's time horizon T. Being γ the discount factor:

$$G = \sum_{k=0}^{T} \gamma^k R_{k+1}$$

Environment

To implement the different Reinforcement Learning algorithms, an environment following OpenAI gym's [21] convention was created. This facilitates the integration of different RL libraries with our project.

The gym environment consists basically of a set of different functions that wraps around the formula simulator, standardizing the use of the environment.

Due to time constraints, it was not possible to train a policy using this method. Therefore, it is left as an open line of research for the following working groups.

Perception (GAN)

Introduction

Transferring knowledge from simulation to reality is one of the biggest challenges in the application of simulation-based Machine Learning algorithms, such as Reinforcement Learning, to robotics. Even though one could implement this type of learning directly in the real robot, the advantages of using a simulation are notorious:

- Faster training, as the simulation can be "fast-forwarded" and one can even run many episodes in parallel. This is important, because at today's date, RL algorithms are highly data inefficient, requiring thousands of examples to yield a performant model.
- Automated training routine.
- Cost reduction, as the real robotic platform is not at risk.
- Mechanical design and software development can be done in parallel.

In the present work, the difference between simulation and reality is present mainly in two different aspects of the robot: A visual and a dynamical one. We will focus primarily on the first one, as the dynamic is not a dominant factor in this case. This is mainly due to the low-speed capability of the robot.

The image below shows the notorious gap between simulation and reality.

Figure 30. Visual gap between the simulation (Left) and reality (Right).

To overcome this visual reality gap, a technique known as Domain Randomization (DR) [22] was used. The simulation domain (*S*) is different from the real domain (*R*), leading to a policy trained in simulation (π_S) not working as expected in the real robot.

$$\pi_S(x_S) \neq \pi_S(x_R)$$

The idea behind the visual Domain Randomization approach is to add perturbations to the simulated domain *S*, so as to extend it into a new domain *S'*, which includes the real domain *R*.

$$\pi_{S'}(x_{S'}) \approx \pi_{S'}(x_R)$$

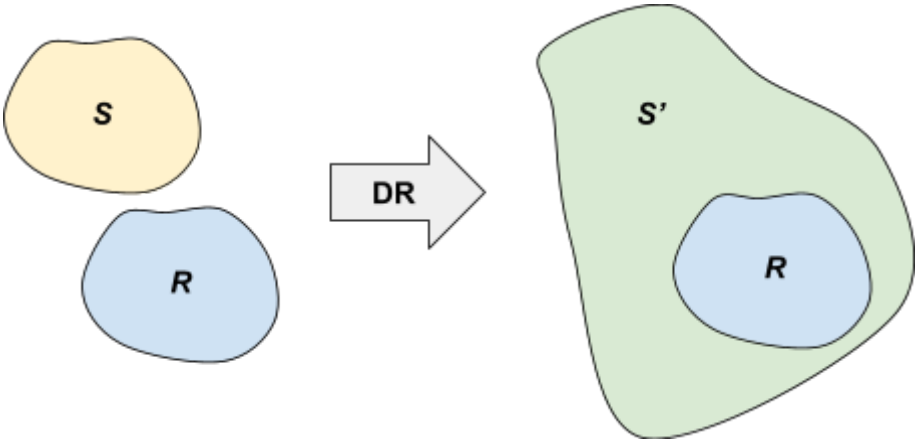

Figure 31. Diagram representing the effect of Domain Randomization (DR) over the different domains (S, R and S').

The image below shows a comparison between the three mentioned domains. Both S and *S'*, are obtained from simulation. The difference between them, is that in the case of *S'* different visual randomizations are applied.

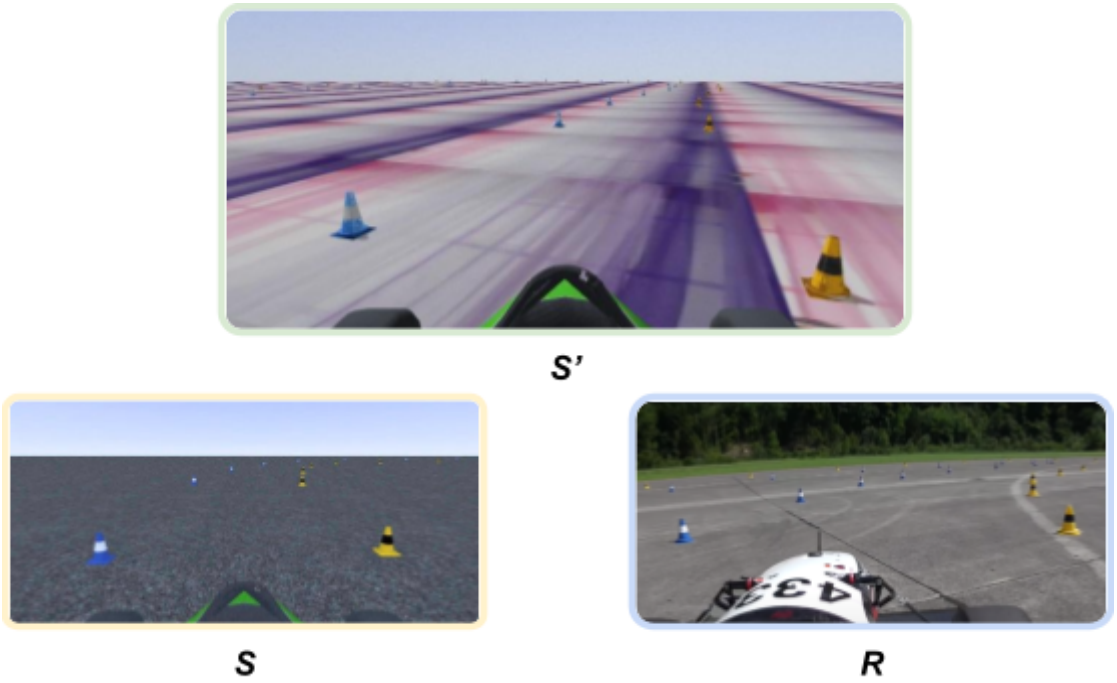

Figure 32. Effect of DR in this particular work.
Top: Randomized domain S'. Bottom Left: Simulation domain S.
Bottom Right: Real domain R.

Instead of applying this method directly to an end-to-end policy, an indirect implementation was preferred.
The direct approach consists of training a driving policy ($\pi_{S'}$) directly from randomized data (*S'*).

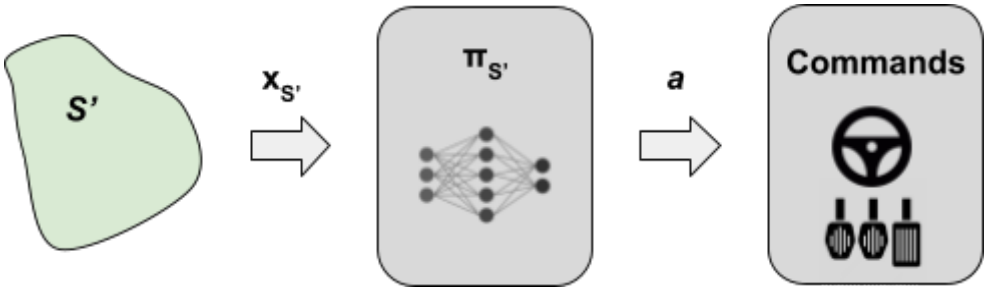

Figure 33. Control block diagram of the direct implementation of DR.

In this case, the policy $\pi_{S'}$ must extract the relevant features of the input image, and from these, choose the right commands (*a*).

On the other hand, the indirect method yields an intermediate, intuitive image, from which the policy can learn from [23]. This intermediate image (new domain *M*) is generated using a GAN, and it simplifies the task of the driving policy.

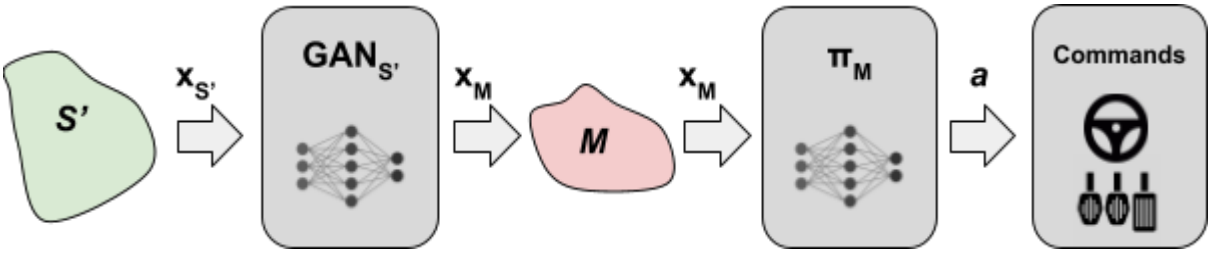

Figure 34. Control block diagram of the indirect implementation of DR.

The $GAN_{S'}$ is trained using randomized simulated images (*S'*), to create a simplified mask (*M*) of the input image.

$$GAN_{S'}(x_{S'}) \approx GAN_{S'}(x_R) = x_M$$
$$\pi_M(x_M) = a$$

Therefore:

$$\pi_M(GAN_{S'}(x_{S'})) \approx \pi_M(GAN_{S'}(x_R)) = a$$

The indirect method has the following advantages:

- Interpretable simplified image
- Guided training, enhancing relevant features
- Simplifies training of driver policy
- Faster training

Method Overview

To implement this indirect Domain Randomization, we use an image conditioned variant of Generative Adversarial Networks (cGAN).

Simple GAN's [24] consists of two main components: a Generator (G) and a Discriminator (D). The objective of the Generator is to create images that resemble the ones present in the dataset (x), based on the input of a random vector Z. In the other hand, the Discriminator tries to distinguish between the fake images created by the Generator, and the real ones present in the dataset. In this manner, during training, both parts compete against each other optimizing a MINMAX Binary Cross Entropy loss.

$$min_G max_D \; Loss(G, D) = E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

Being:

- $D(x)$: The probability that x come from the dataset (real)
- $G(z)$: The generated (fake) image

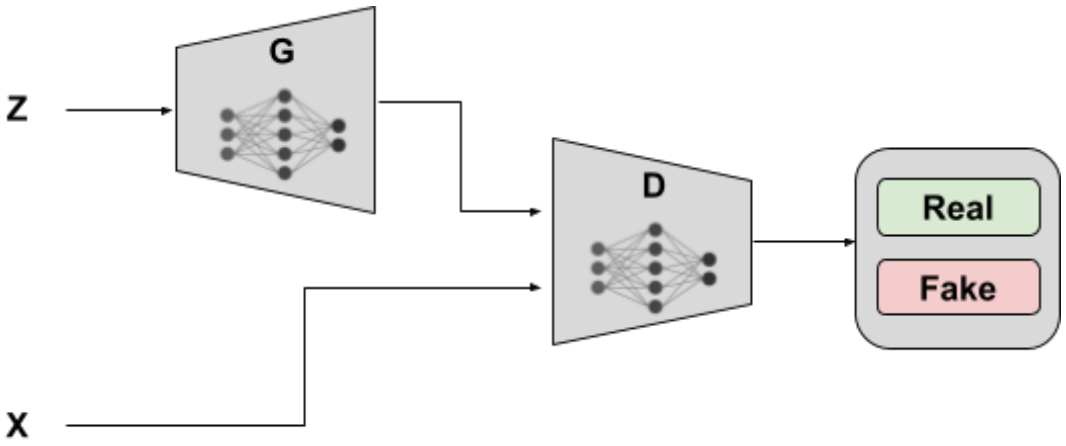

Figure 35. Simple GAN block diagram.

Conditional Generative Adversarial Network (cGAN's) add an extra requirement to the Generator, as they not only have to create similar Images to the ones of the dataset, but also must follow certain requirements (conditions) specified to it.

In particular, image conditioned GAN's (such as pix-to-pix [25]) create an output image based on an input image. Therefore, the generated image must follow specific semantics of the input image. Even though this can be done with a simpler supervised approach, recent literature has shown that adding an adversarial term leads to much sharper images.

In this work, the main objective of the cGAN is to obtain a segmentation mask (x_M) from the inputted image ($x_{S'}$ or x_R). This segmentation filters noise and irrelevant objects present in the image, keeping only the present cones, while also highlighting and normalizing their respective color. In this way, no matter the source of the image (simulation or reality), the outputted one will always correspond to the same domain (*M*). This segmentation is later fed into the driving algorithm (policy). This leads to the direct implementation of a simulation-trained model directly into the real world.

In addition to the segmentation mask, the cGAN also outputs a pixelwise depth estimation (x_D) based on the inputted image. This estimation is said to help improve the segmentation, while also extending the amount of relevant features extracted from the original image.

In summary, the cGAN is trained to output both an estimated segmentation (G^M) and depth (G^D), similar to the respective ground truth (x_M and x_D).

$$G(x_{s'}) = \left(G^M(x_{s'}),\ G^D(x_{s'})\right) \approx \left(x_M,\ x_D\right)$$

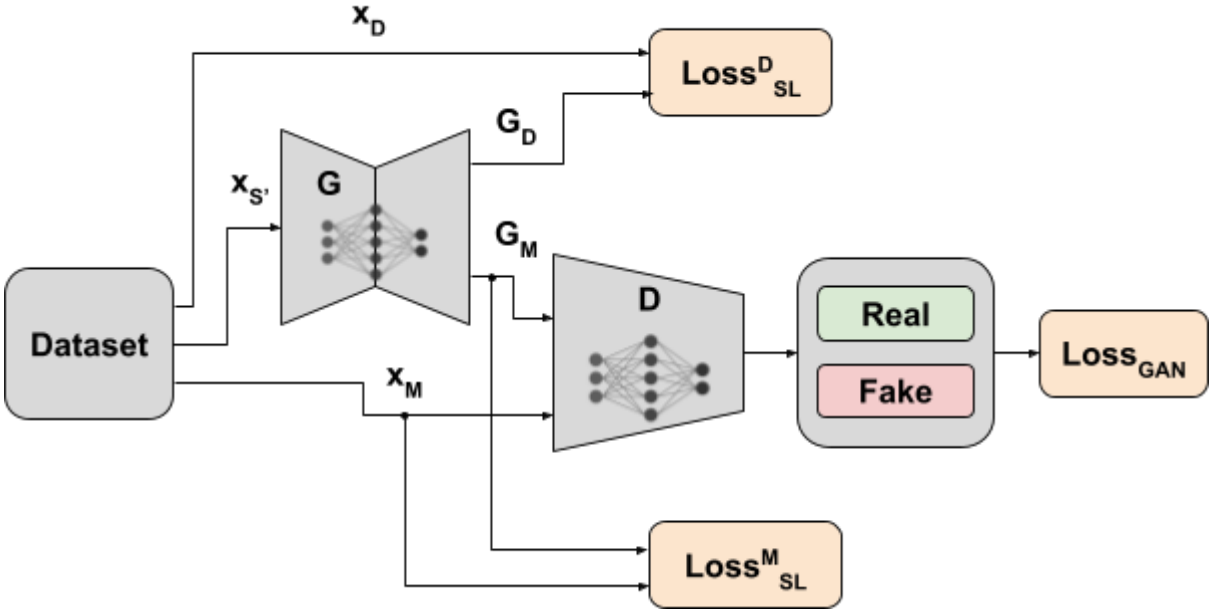

Figure 36. Block diagram showing the computation of the different losses in the implemented conditional GAN.

As with the traditional GAN, we search for the parameters of both G and D that optimizes the loss function. In this case, the loss function is a combination of a Binary Cross Entropy term for the GAN component, and a Mean Squared Error (MSE) term for the Supervised Learning components.

Being:

$$Loss^{\alpha}_{SL}(G) = E_{(x_{S'}, x_{\alpha})}[MSE(G_{\alpha}(x_{S'}), x_{\alpha})]$$

$$Loss_{GAN}(G, D) = E_{x_M}[log(D(x_M))] + E_{x_{S'}}[log(1 - D(G_M(x_{S'})))]$$

The loss function is given by:

$$min_G\, max_D\; Loss(G, D) = Loss_{GAN}(G, D) + Loss^{M}_{SL}(G) + 100\, Loss_{SL}^{D}(G)$$

Implementation

As mentioned earlier, for the method to work in the real platform, various randomizations were applied. In particular:

- The numbers of cones present in the image.
- The position of the cones relative to the camera.
- The type of cones present (blue, yellow or orange).
- The texture and color variation of the floor and cones.
- The color, intensity and direction of lighting.
- Gaussian noise present in the image.

The following image shows the mentioned randomizations implemented in simulation. In this manner, a dataset of approximately 15000 image trios was created, from which the cGAN is later trained. Each trio consists of a randomized image (*S'*) with its corresponding segmentation (*M*) and depth (*D*).

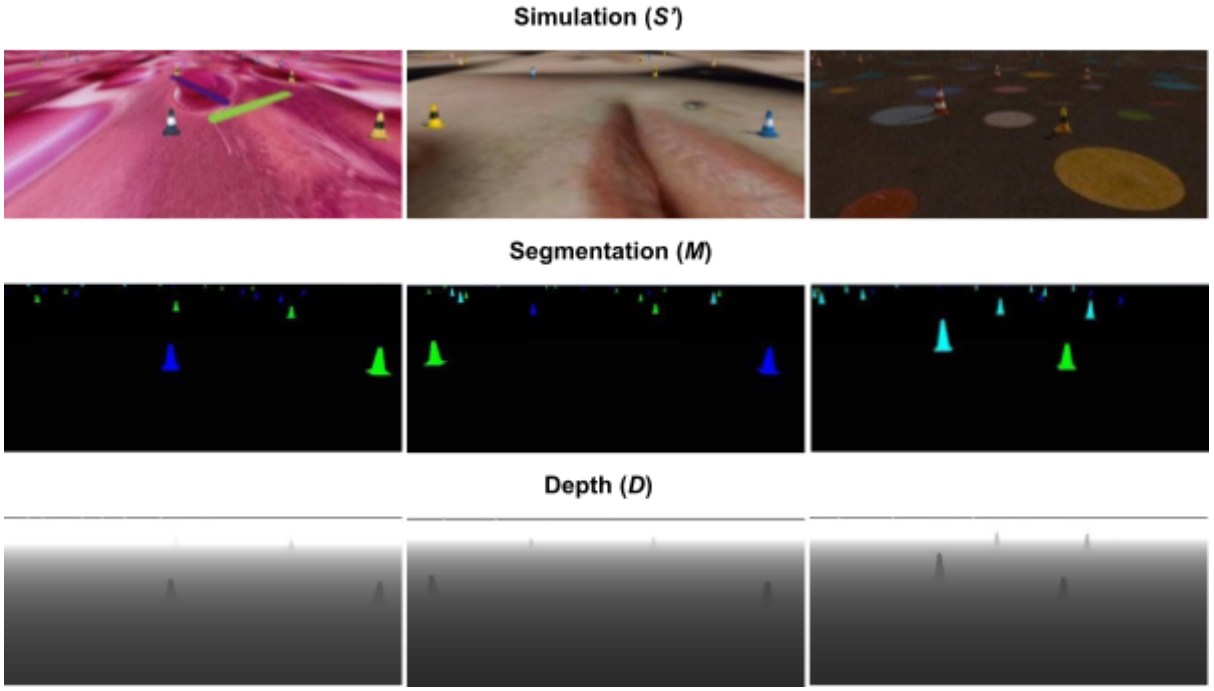

Figure 37. Samples of the dataset used during training of the cGAN.
Top Row: Randomized simulated images. Middle Row: Target segmentation of each corresponding image.
Bottom Row: Target depth of each corresponding image.

The segmentation is targeted to the cones present in the scene. Blue cones are highlighted in blue (RGB = [0, 0, 255]), yellow cones in green (RGB = [0, 255, 0]), and orange cones in cyan (RGB = [0, 255, 255]). The rest of the irrelevant objects (floor, formula, trees, etc) are left in black (RGB = [0, 0, 0]). These colors were selected to maximize the difference between object types, using only two channels, the green and blue. Reducing the number of channels from three (RGB) to two (GB), simplifies the driving policy.

It is interesting to notice that to create a segmentation model, as shown in the picture above, using only real images would require a huge amount of human labor. But instead, the simulation was used to automate the process.

The generator model follows of a U-Net architecture. It mainly consists of two parts, an encoder and a decoder, connected via skip connections.

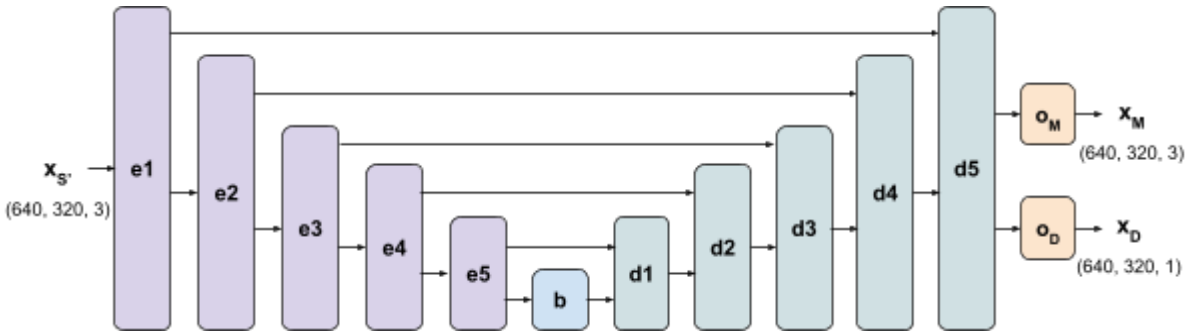

Figure 38. Generator's U-Net architecture.

The encoder blocks (e_n) consist of a 2D convolutional layer, followed up with a Batch Normalization layer (in most blocks) and a Leaky Rectified Linear Unit (Leaky ReLU) activation function. The convolution is done with a 4×4 kernel, stride of 2 in both directions and zero padding. The number of filters varies between blocks.

Block	Number of filters	Batch Normalization
e1	64	No
e2	128	Yes
e3	256	Yes
e4	512	Yes
e5	512	Yes

Table 3. Hyperparameters for each encoder block.

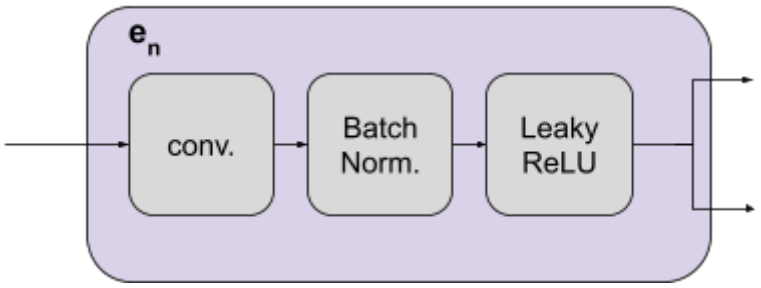

Figure 39. Internal block diagram of the encoder block.

The bottleneck block (b) has a convolutional layer with 4×4 kernel, 512 filters, stride of 2 in both directions and zero padding, followed with a ReLU activation function.

In the other hand, the decoder blocks (d_n) consist of a transposed convolutional (deconvolution) layer, followed up with a Batch Normalization layer, dropout (in some blocks), a channel-wise concatenation with skip connections and a ReLU activation function. The deconvolution is done with a 4x4 kernel, stride of 2 in both directions and zero padding. The number of filters varies between blocks.

Block	Number of filters	Dropout
d1	512	Yes
d2	512	Yes
d3	256	Yes
d4	128	No
d5	64	No

Table 4. Hyperparameters for each decoder block.

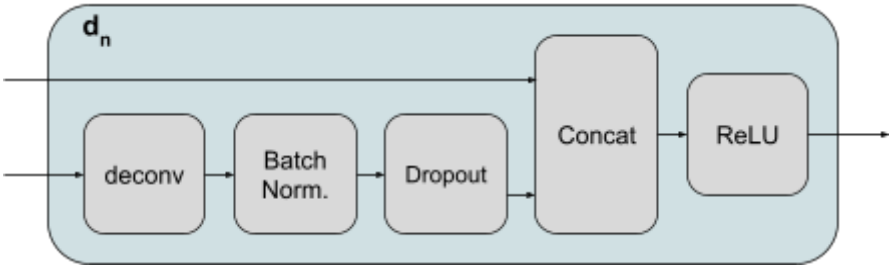

Figure 40. Internal block diagram of the decoder blocks.

Lastly, the output blocks, O_M and O_D, have both a deconvolution layer followed with a hyperbolic tangent (tanh) activation function. The O_M block, which outputs the segmented image, have a kernel with 3 filters. The O_D block, which outputs the depth estimation, has a kernel with 1 filter.

The discriminator follows an architecture similar to the encoder section of the U-Net. It consists of the e_1 to e_5 blocks followed by a convolution with a 4×4 kernel, stride of 1 in both directions, 1 filter, zero padding and a sigmoid activation function. After this patch-based classification, a mean is computed to output a single value. The fifth encoder block (e_5), has a convolution with stride of one, instead of a stride of 2, as it is used in the generator.

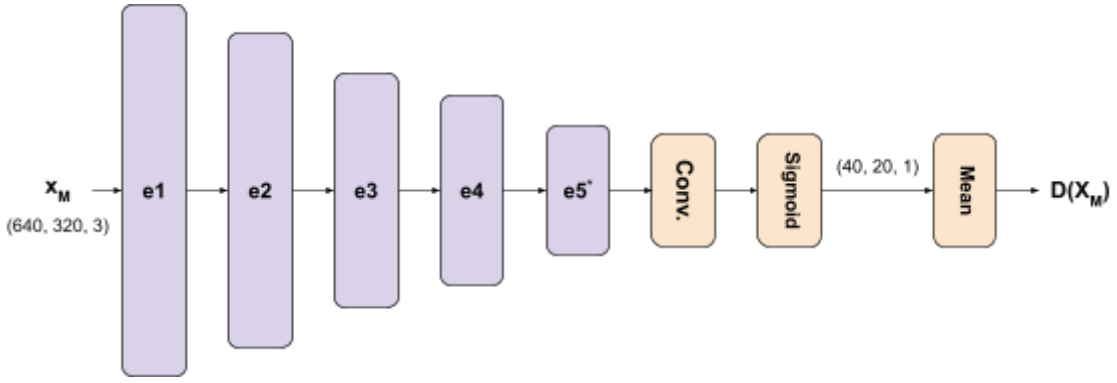

Figure 41. Discriminator's Neural Network architecture.

As it can be seen with the dimension of the output of the sigmoid layer (40, 20, 1), the discriminator follows a patch-based format. This means that it classifies between "real" and "fake" different parts (patches) of the image, and then averages all this outputs.

Partial results

The feasibility of the approach was evaluated using open-source material of real formula student platforms on track. In particular, a video of the FLÜELA driverless from AMZ was used [26].

The image below shows a single shot of the FLÜELA on track (*R* domain). This shot will be used to demonstrate the performance of the cGAN approach.

Figure 42. Image taken from the on-board camera of the FLÜELA on track.

It is important to highlight that the following segmentation and depth estimation was done using the Generator (*G*), which was trained using only simulated images. The generator has never seen any real images during training.

Figure 43. Generator's segmentation of the sample image.
Left: Sample of the FLÜELA on track (R domain). Right: Segmentation (M domain) of the left image, created by the Generator of the cGAN.

The top figure shows the segmentation of the relevant features of the sample image. These are the different present cones, with their respective color. It is interesting to see how the yellow line present in the floor is avoided, this shows that the Generator not only discriminates over color, but also discriminates shapes. It is even more interesting to notice how the Generator succeeds in the identification of the yellow cone that is over the yellow line.

The image below shows an (almost) top view of the current shot. This top view was created using a point cloud, which was made using the depth estimation of the generator. Supplementary material showing the performance of the Generator over the whole track is available at the project's Git repository [19].

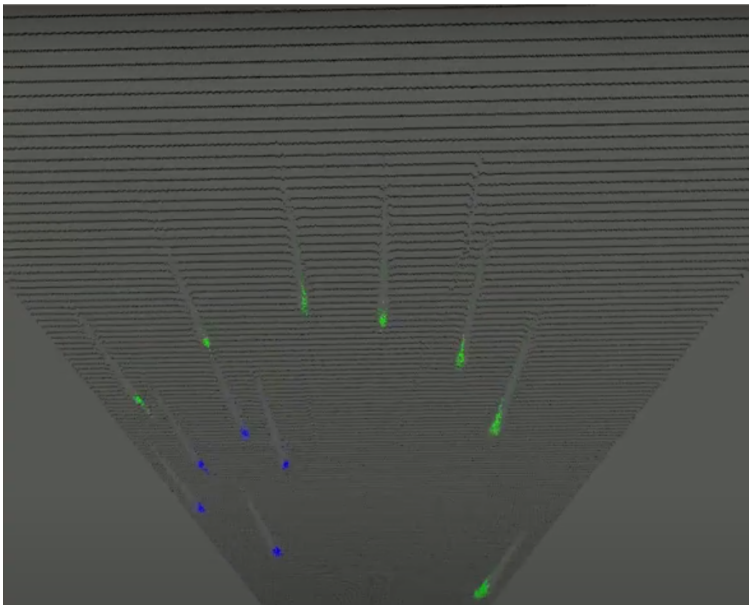

Figure 44. Point Cloud created using the Generator's depth estimation.

In the following sections, it will be shown the implementation of this method on the custom robot, with the following concatenation of the driving policy, which will be trained over simple segmented images (*M* domain).

Robot

Design

As the actual electric formula that should compete in the driverless category was not available, a small vehicle was designed that would serve as a modular platform capable of multiple sensor configurations and as a workbench for future autonomous projects. The vehicle consists of two levels, the bottom one holds the two motors (one for each back wheel) in charge of driving the vehicle and a servo to steer the front wheels. A microcontroller is employed, as well as motor drivers that take care of the low-level control of all three actuators. The top level on the other hand holds the computer in charge of the high-level control and connectivity of the vehicle, as well as the one in charge of getting information from the different sensors. Also in the top level, the power source is installed. Finally, sensors can be mounted on both levels depending on the user's need, allowing for new sensors to be included with their corresponding mounting support. Details regarding construction, hardware employed and software are detailed in the following sections.

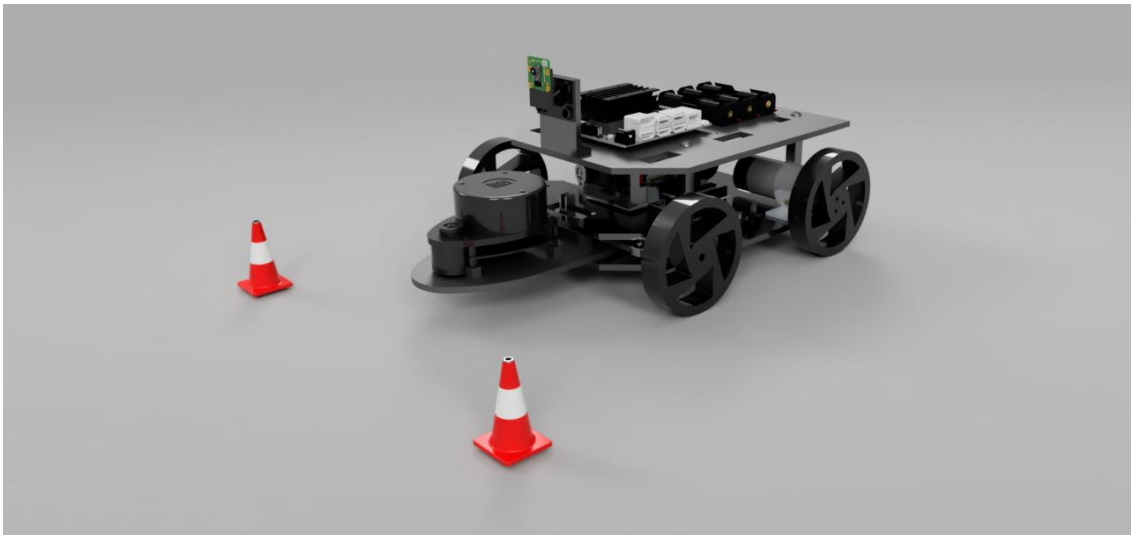

Figure 45. Robot Design

Construction

The vehicle was designed considering that most of the parts will be 3d printed. Mounting elements (e.g., screws, nuts) were employed to assemble the parts as well as helping in rigidizing the structure. On the other hand, bearings were used to allow a smoother movement between parts, such as for the steering mechanism. On figure 49 examples of the STL parts are shown, and on figure 50 the result of the 3d printed parts. See annex for a complete list of the mechanical elements used to assemble the vehicle.

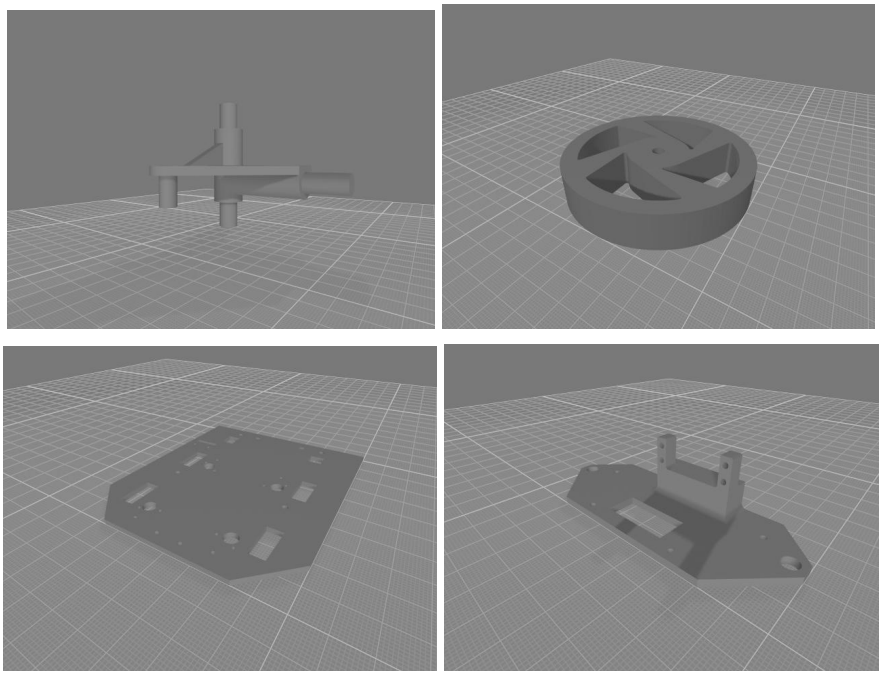

Figure 49. STL Parts

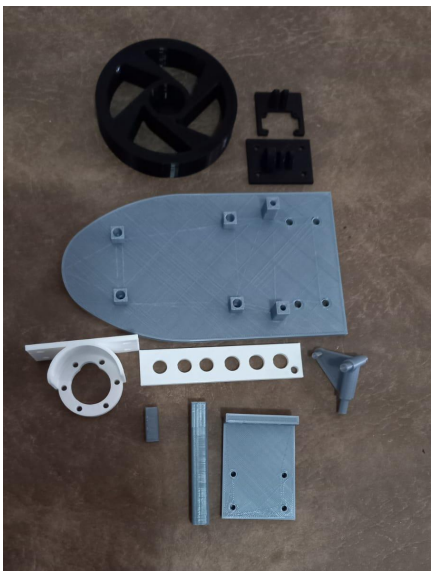

Figure 50. 3D-Printed Parts

On figure 51 the robot is shown in its early stages of construction, control of actuators was tested as well as the parallel steering.

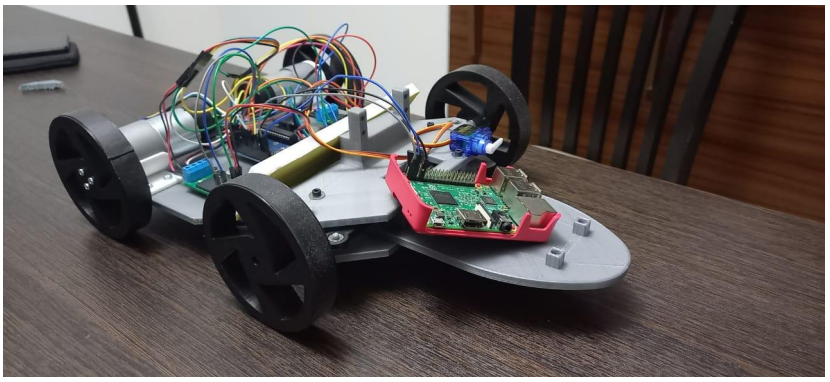

Figure 51. Early Stages of Construction and Hardware Testing

On figure 54 all remaining parts were assembled as well as the complete top level. The Jetson Nano was mounted as well as sensors, power source and cables.

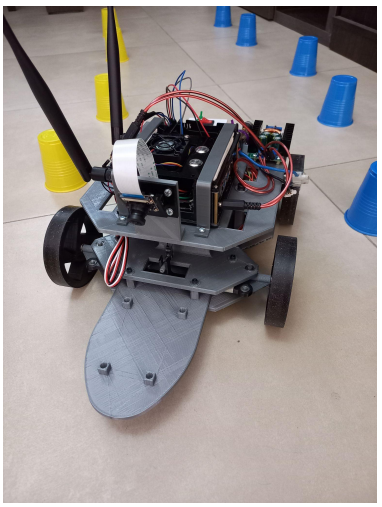

Figure 52. Robot Fully Built

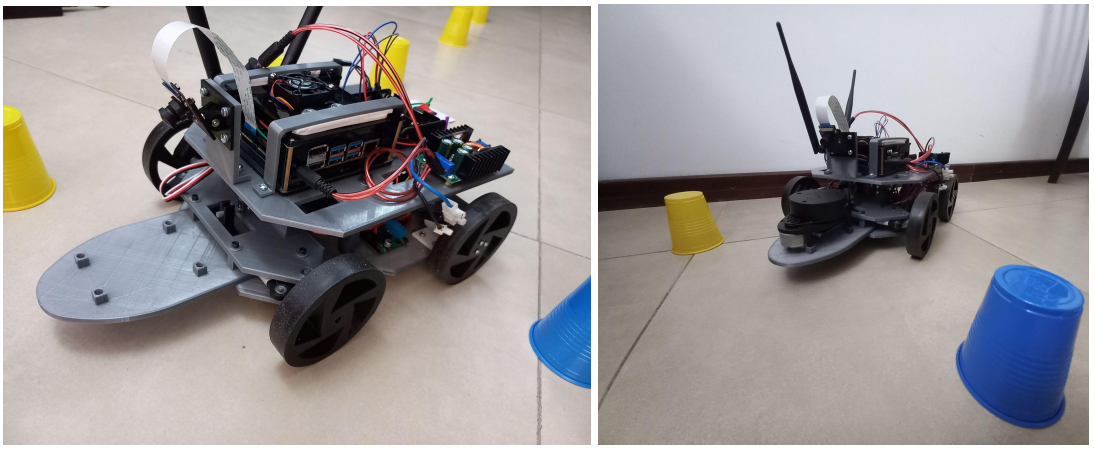

Figure 54. Sensors and Hardware on Robot

Vehicle Dynamics and Control

Most common vehicles seen on the road employ an Ackerman steering that ideally makes the wheel angles all have a common turning circle. In this way, front wheels tend to have minimum slip in curves and reduce tire wear. For this design, and for the sake of simplicity and minimum number of parts, a parallel steering was chosen. In this configuration, front wheels do not have a common turning circle and will tend to slip in curves, but as the robot has very slow dynamics this is not an issue.

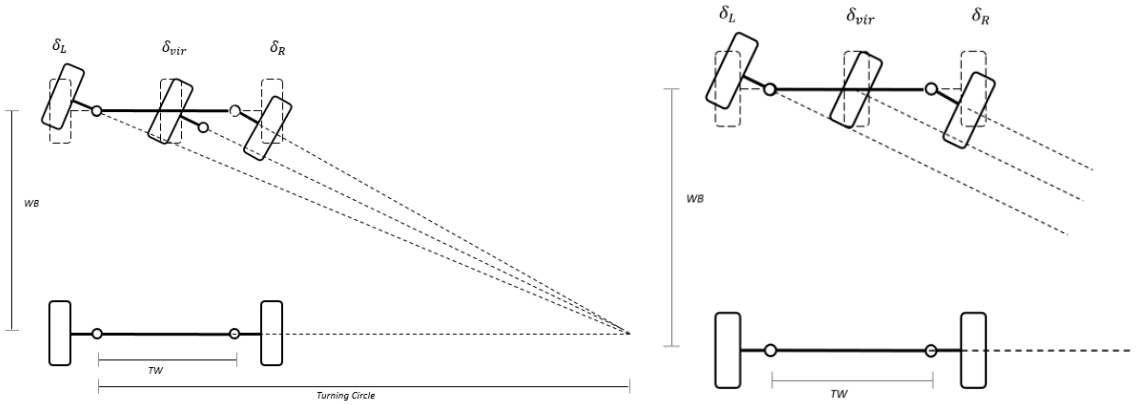

Figure 56. Ackermann vs Parallel Steering [27]

Our vehicle, as mentioned in the design section, has two motors, one on each back wheel to drive the vehicle. Therefore, a differential drive controller is required to set different wheel speeds and allow the vehicle to turn. A servo is used to control the steering mechanism of the front wheels. For this project the robot accepts as input a steering angle and a velocity which is compatible with the way we drive common vehicles with the slight difference that the "braking" and "accelerating" are both combined into a single velocity control variable.

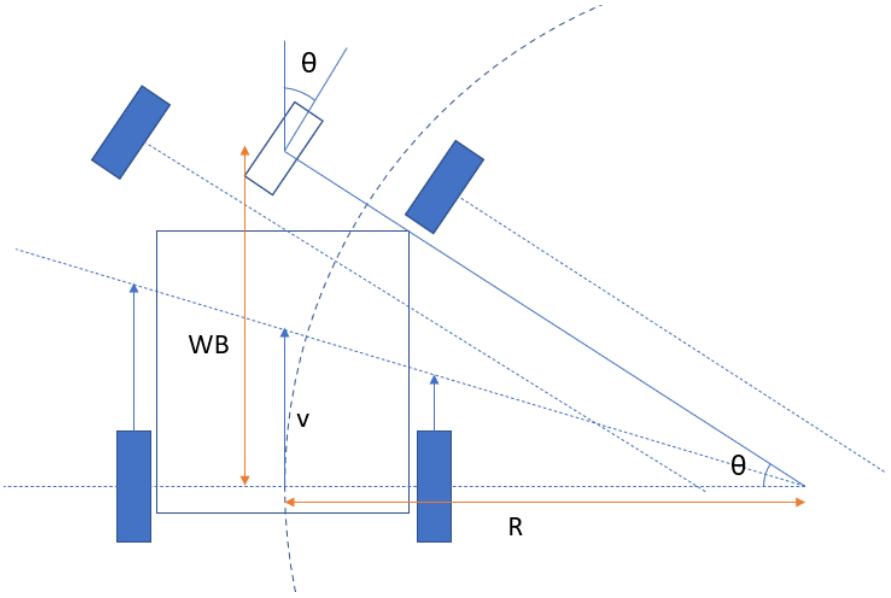

Figure 57. Vehicle Dynamics

Taking as input steering angle θ and velocity v, it is possible to calculate the angular velocity w of the vehicle using the following equations.

$$v = w \cdot r \qquad \tan(\theta) = \frac{WB}{R}$$

$$w = \frac{v \cdot \tan(\theta)}{WB}$$

The differential drive controller takes as input v and w and by adding geometric parameters such as the track width (TW) and the size of wheels, calculates the angular speed of each back wheel. The input steering angle θ on the other hand is mapped directly to an angle in the servo, taking into account the geometry of the steering mechanism.

One interesting thing to notice from the equations shown above is that just like real cars, the vehicle is not "allowed" to turn if the command velocity is zero ($v = 0 \Rightarrow w = 0$) . In other words, our vehicle is programmed to behave as a real car even though it has the complete freedom of sending opposite angular speeds to each back wheel, which in the case of a mechanical differential drive will be impossible.

Hardware

Regarding the hardware used in our robot, the diagram shown below shows the location of each one for each level. See the annex for a complete list and specifications of the hardware employed, including motors, servo, microcontrollers, sensors, drivers, and communication interfaces.

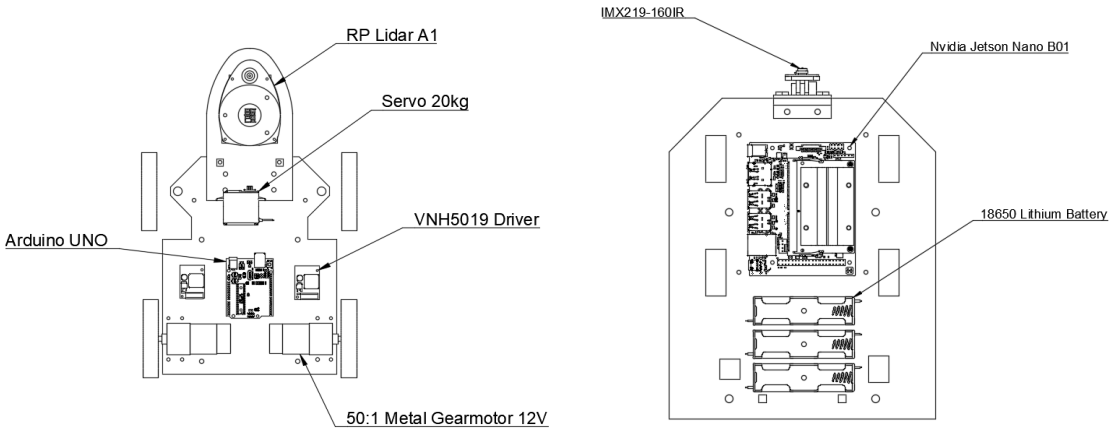

Figure 59. Hardware Layout
Left: Bottom Level Right: Top Level

Bottom Level	
Name	Quantity
50:1 Metal Gearmotor 37Dx70L mm 12V with 64 CPR Encoder (Spur Pinion)	2
Servo 20KG	1
Arduino UNO	1
VNH5019 Motor Driver Breakout	2
RP Lidar A1	1

Table 5. Hardware on Bottom Level

Top Level	
Name	Quantity
Nvidia Jetson Nano B01	1
XL4016 Step Down (12V ⇒ 5V)	1
Panasonic NCR 2900mAh (18650)	3
Adaptor Level I2C (5V - 3.3V)	1
IMX219-160IR	1
IMX219-83 Stereo	1
MPU 9250	1

Table 6. Hardware on Top Level

Camera Calibration

In order to perform a calibration, a camera calibration tool already provided as a ROS package was used. This tool takes as input the size of the chessboard, the size of a single square and the name of the camera topic the user would like to calibrate. As soon as the tool starts, the user is required to move the chessboard to different locations, with the objective of obtaining images with variation in x and y location, size and skew. See figure 61 for examples. Once enough images are taken, the tool begins the calibration, similar to the process explained in the Camera Technology Overview section. Once it's finished the tool outputs the results containing the intrinsic parameters of our camera, examples of the results are shown below in figure 62.

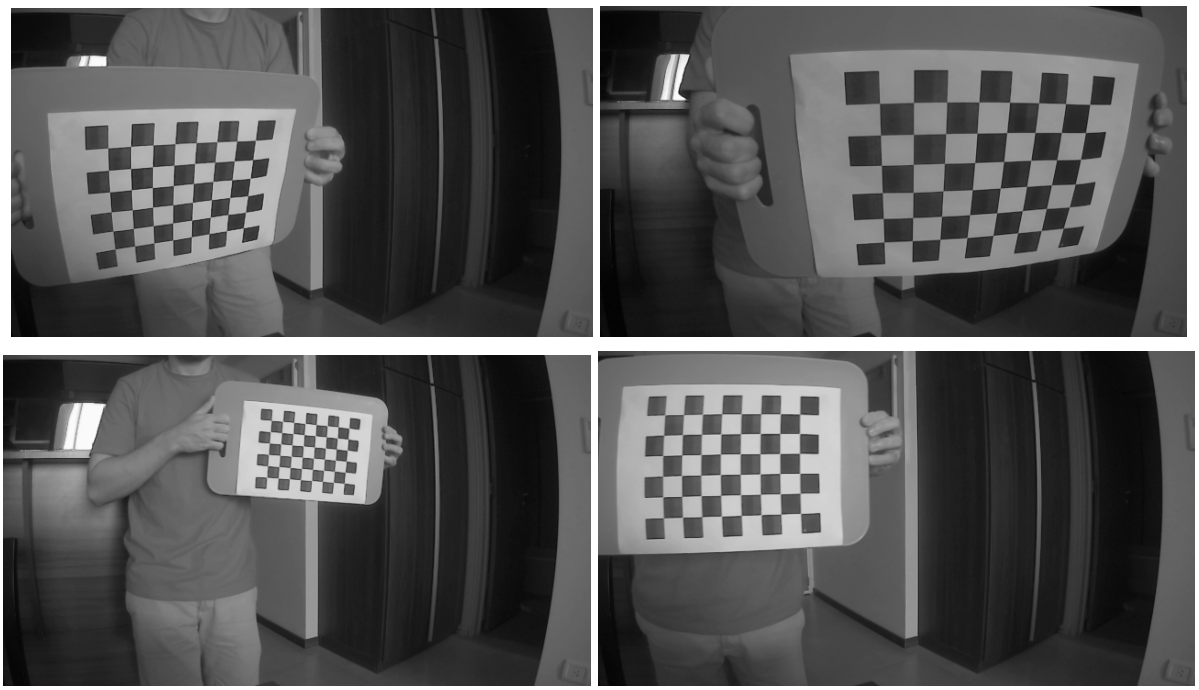

Figure 61. Camera Samples for Calibration

```
image_width: 640
image_height: 360
camera_name: narrow_stereo
camera_matrix:
  rows: 3
  cols: 3
  data: [ 401.20988,    0.     ,  315.55324,
            0.     ,  400.9087 ,  166.39583,
            0.     ,    0.     ,    1.     ]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5
  data: [-0.322257, 0.095476, -0.002642, -0.001059, 0.000000]
rectification_matrix:
  rows: 3
  cols: 3
  data: [ 1.,  0.,  0.,
          0.,  1.,  0.,
          0.,  0.,  1.]
projection_matrix:
  rows: 3
  cols: 4
  data: [ 305.4006 ,    0.     ,  311.37576,    0.     ,
            0.     ,  371.82794,  163.34566,    0.     ,
            0.     ,    0.     ,    1.     ,    0.     ]
```

Figure 62. Output of Camera Calibration

Once the parameters of the camera are obtained, it is possible to rectify the raw image. This rectified image will then be used as input for any deep learning algorithms, as in most cases they are trained with free-distortion images. On figure 63 the results of running the camera rectification are shown below. Notice how straight edges from cones become straight again on the right image.

Figure 63. Difference between the distorted and the rectified image.
Left: Raw image. Right: Rectified image.

Simulation

The robot designed for this project was also included in the simulation in order to have a more accurate model representation of the real platform regarding vehicle dynamics and sensor information. Models of cones and objects used in real testing were also added to the simulator.

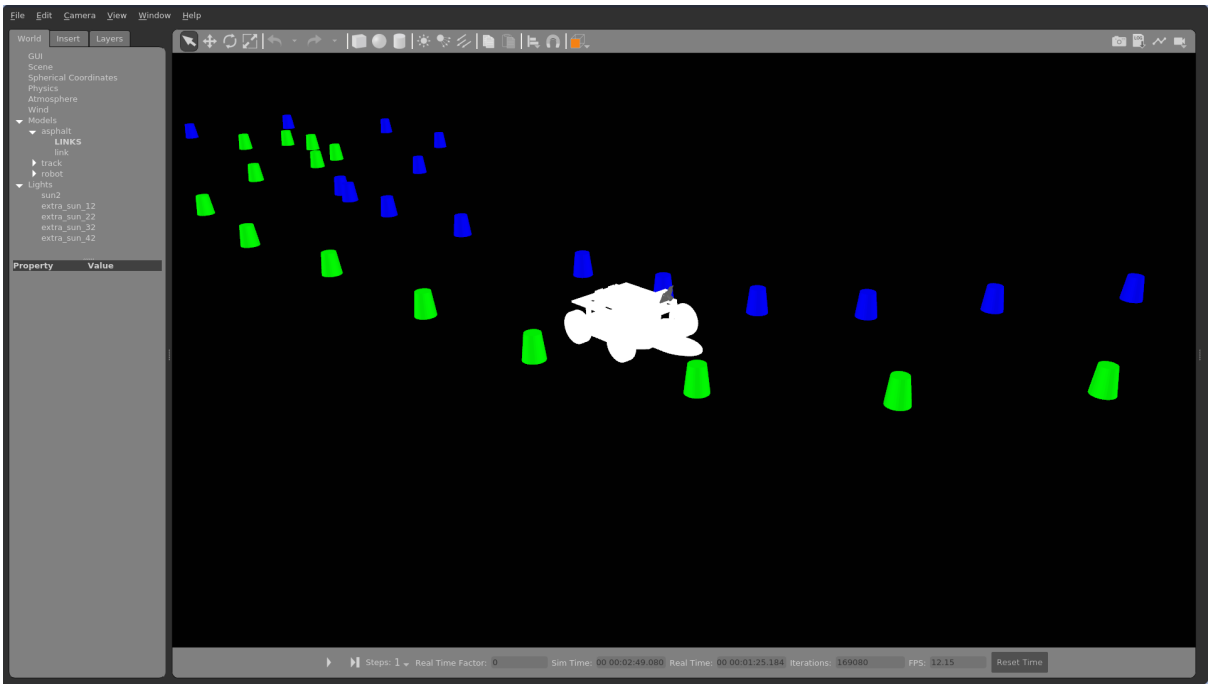

Figure 64. Robot in Simulation

Furthermore, the camera employed on the real robot was also simulated using the intrinsic parameters found in the previous section. Results can be seen on figure 65. It is important to recall for the case of following the indirect method that the driving policy is trained in the segmentation domain M. Hence the reason our new robot simulator has drifted apart from reality.

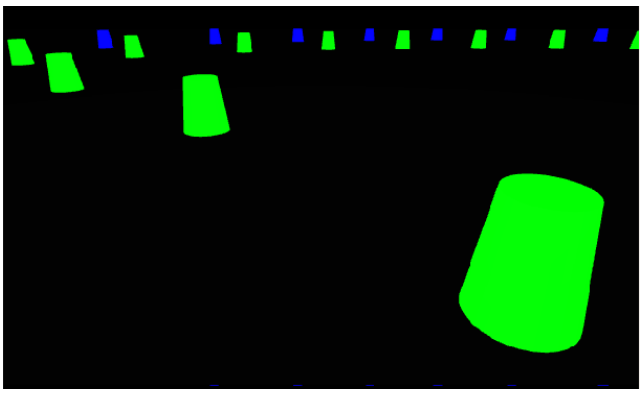

Figure 65. Simulated Camera

Software and Control

As ROS uses the local network to communicate different nodes via messages (a.k.a. topics), it is possible to set a laptop as master and connect to the Nvidia Jetson Nano (i.e., our robot) through ssh. The application starts by running a launch file in the master device. In this launch file, it is possible to specify the nodes that will be running and where they will be running, either on the master or on our robot. Processes (or nodes) regarding drivers that get sensor information as well as the differential drive run on the robot. This architecture provides the flexibility to run visualization tools on the master or for example the ability to control the vehicle all through the same ROS application that runs on the same network. It is up to the user to choose where certain nodes will be running, considering that messages transmitted through the network may not satisfy real time applications.

To control the actuators of the robot, the architecture provided by the ROS Control package was implemented [28][29]. This architecture allows the user to select from a set of available controllers, including diff drive controllers, velocity controllers, position controllers and effort controllers to control the actuators of a robot. See figure 66 for a visual representation of how the ROS Control package works. In this case, a diff drive controller was chosen to control the back wheel motors and a position controller to control the servo in charge of the steering. On top of the selection of the controllers, a hardware interface was required to communicate with the Arduino. I2C communication protocol was used for its simplicity and already available pins supporting this protocol in both the Jetson Nano and the Arduino. This hardware interface includes methods to communicate through I2C and the possibility to read and write messages with the Arduino. The diff drive controller running in the Jetson Nano sends velocity set points of each back motor to the Arduino. On the other hand, the position controller sends to the Arduino the desired steering angle of the servo. The Arduino then receives these set points and controls each back motor's velocity using a PID. The motors are controlled using a PWM signal and the velocity is measured using the encoders already integrated in the motor. The angle of the servo is controlled using a PWM signal directly, as the servo runs its own position control loop.

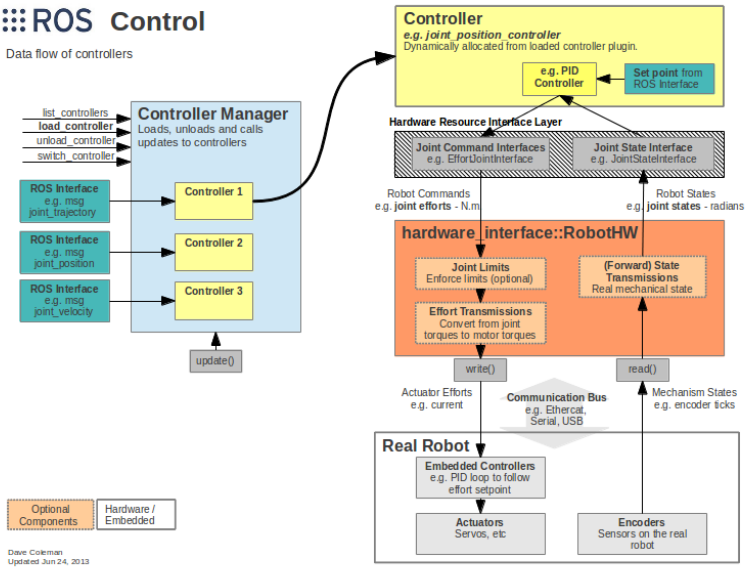

Figure 66. ROS Control Architecture [28]

Manual Mode

In Manual Mode, the robot receives the commands from a joystick connected to the laptop. The laptop reads the joystick through a USB connection and then sends the commands of angle and velocity through a ROS topic. The differential drive running on our robot receives those messages and through the hardware interface node sends the commands to the Arduino to control all three actuators. The RViz (i.e., visualization tool) runs on our laptop and takes care of receiving and visualizing all sensor data as well as the odometry and joint state of our robot. An overview of the communication between robot and computer is shown on figure 67. Nodes running in manual mode can be seen on figure 68. There is an option to record sensor data through the use of Rosbags, which lets you replay data as if it was coming from the real sensor regarding timestamps. These Rosbags can also be used as a dataset for training Deep Learning models.

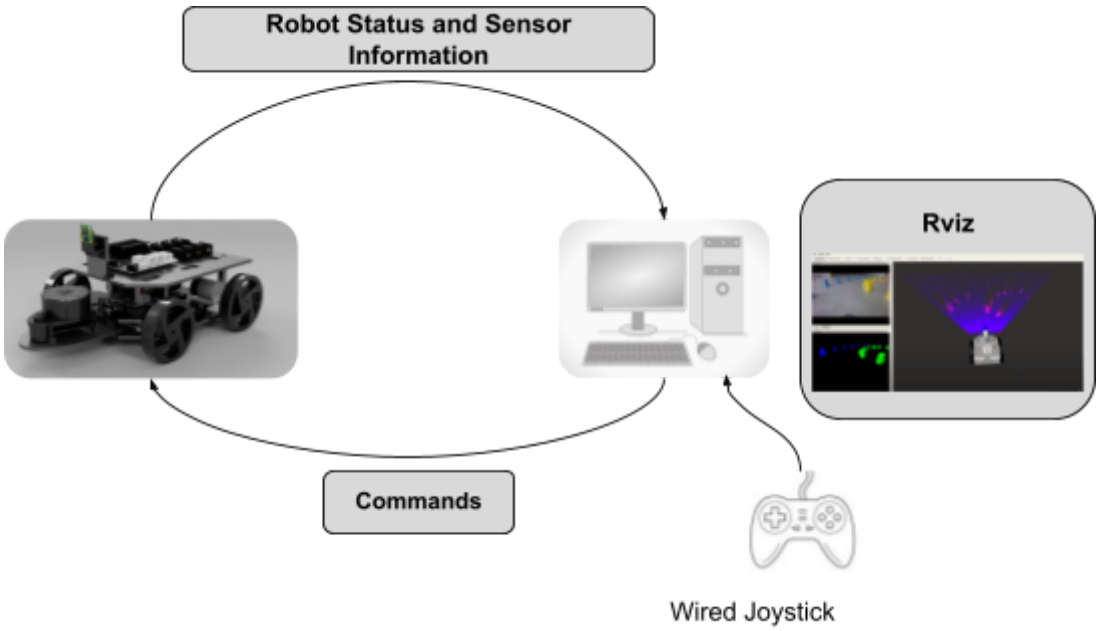

Figure 67. Robot in Manual mode

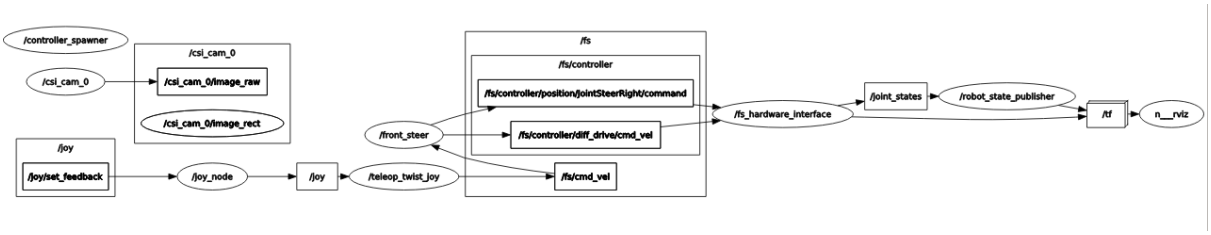

Figure 68. Manual Mode RQT Graph

Driverless Mode

In this mode, the only difference with respect to manual mode is that a joystick is no longer required and the control of the vehicle is replaced by a series of nodes that take care of processing the image and output a control command. Due to the fact that TensorFlow 1.15 was not available as a Python package for Python2.7 in the Jetson Nano as seen on [30], both neural networks were run on a laptop where this Python package is still supported for that type of architecture (i.e., x86). In order to achieve real time performance, images sent over the network must be compressed. ROS image transport package [31] was used to take care of this issue and receive images from the robot with minimum delay and at a high frequency. Finally, the output from our policy is sent back to the robot as ROS topic to command the vehicle. An overview of how and where the image is processed is shown on figure 69.

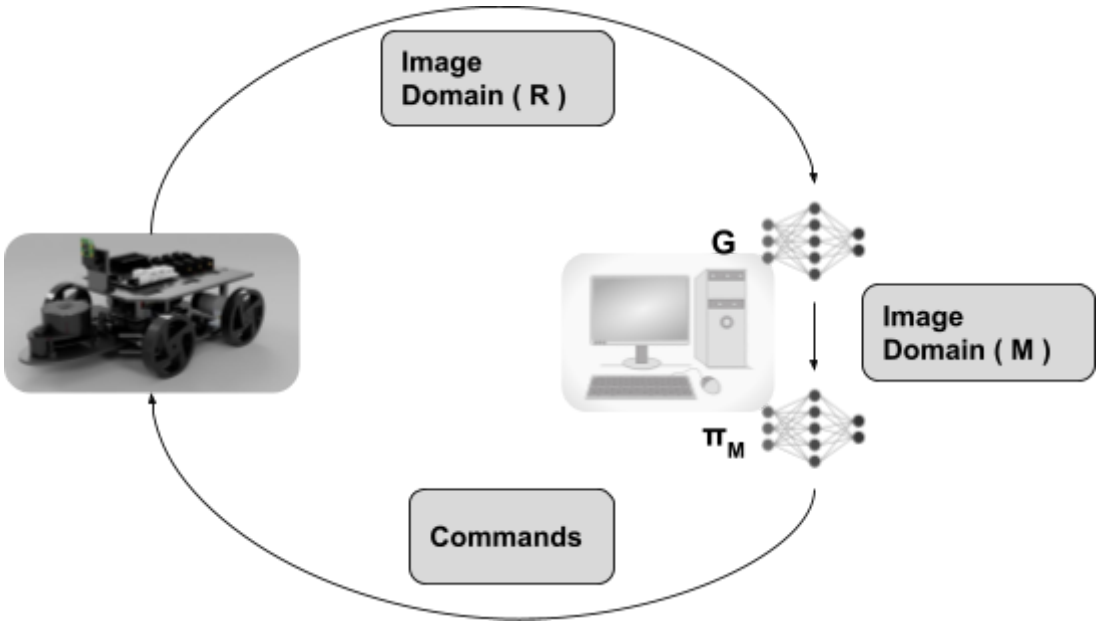

Figure 69. Robot in Driverless Mode

Simulation and Reality Comparison

Software run in simulation and reality are very similar, the only thing that changes between them is that in the case of the real robot a hardware interface is required (including drivers and communication interfaces) to communicate with sensors and actuators. For example, images from a camera are provided through the same standard message regardless of using Gazebo or the actual camera driver in the Jetson Nano. The ROS Control package used to control the actuators also works in similar manner in simulation and reality, with the addition of an I2C communication interface to communicate with the Arduino. See figure 70 for more details.

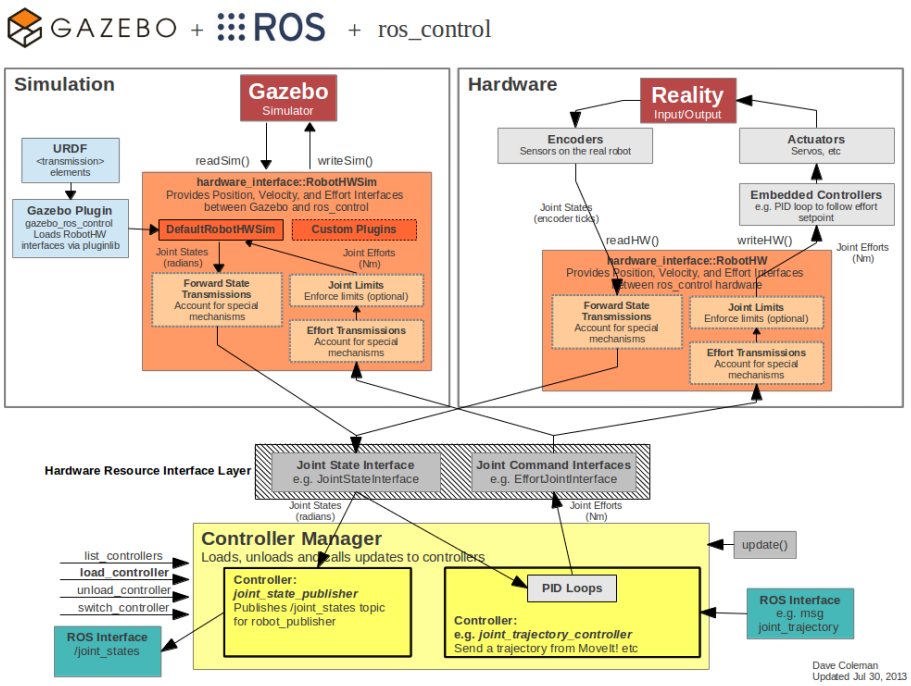

Figure 70. ROS Control (Gazebo and Reality Comparison) [28]

Results

As shown in the previous sections, the feasibility of the approach was evaluated in separated modules. In the first place, the driving policy trained via Behavioral Cloning was tested in the simulation of the formula Student. In second place, the perception module (cGAN) was evaluated using on-board images of the AMZ's FLÜELA driverless.

The last step was to concatenate both modules, while first segmenting the image received from the robot's camera, and later feeding this segmentation to the driving policy, which outputs the corresponding commands. To accomplish this, both the driving policy and the cGAN had to be retrained, using the robot's simulation. This is because the previous Neural Networks were trained on a different environment, where the dynamics of the car, and both the shape and size of the cones were different.

It's important to highlight that, in contrast to what was shown in the 'Imitation Learning' section, the driving policy was trained in the segmentation domain *M*. This is because the images that will receive the policy belong to this domain (cGAN's Generator output).

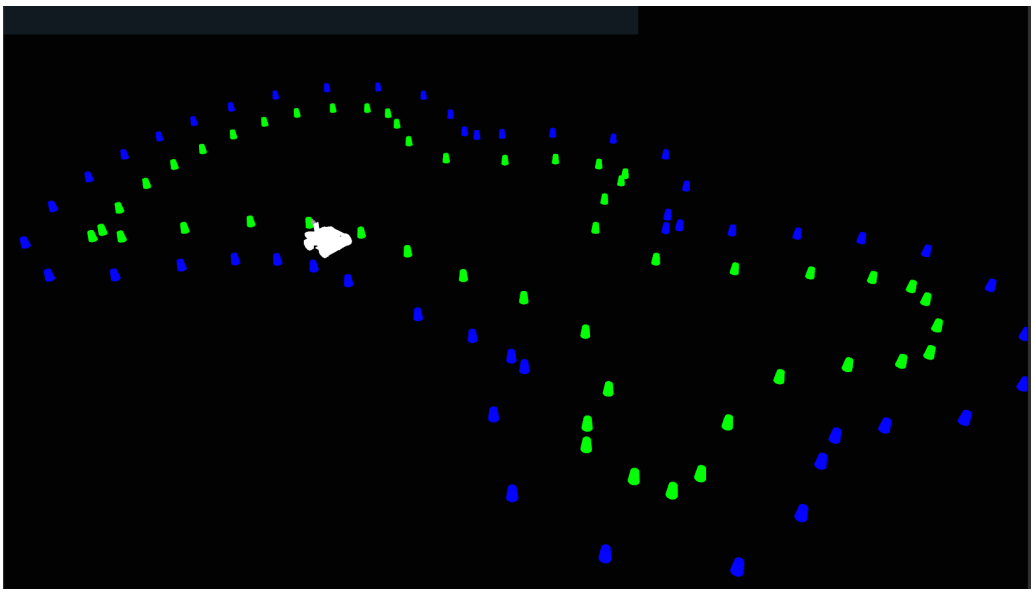

Figure 71. Robot Simulator in Domain (M)

As shown in the image below, the cGAN segmentation and depth estimation yielded promising results in the robotic platform.

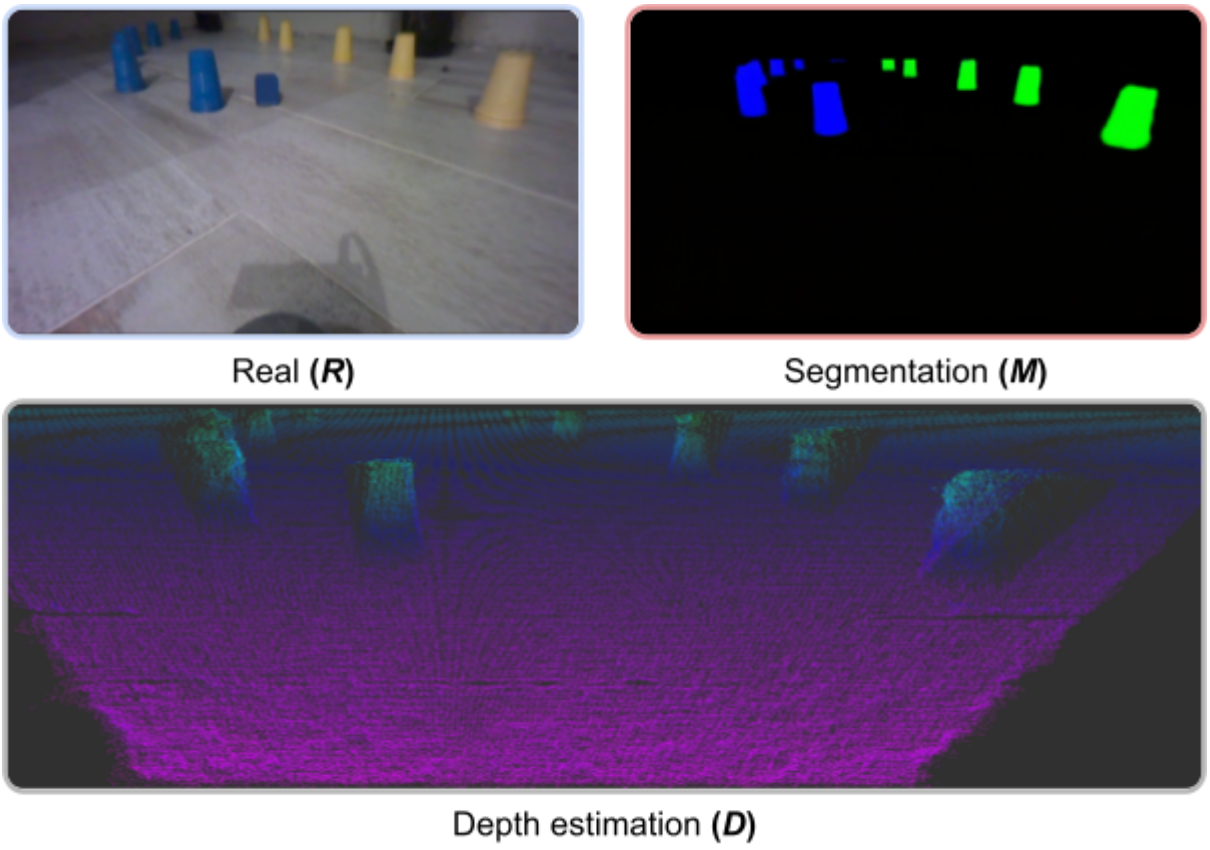

Figure 72. cGAN segmentation and depth estimation.
Top Left: Image captured by the on-board camera. Top Right: Generator's segmentation. Bottom: Point Cloud created by the Generator's pixel-wise depth estimation.

It's interesting to see how accurate the segmentation is, even though the last blue cones are in the shade, making them less visible. It is also important to reemphasize that the generator does not only segment over color, but both by color and shape. This can be seen in the image, as the blue cube, present in the center of the image, is ignored by the generator.

The final evaluation was to test the whole approach in the robotic platform, driving around a random track. To do this, both trained Neural Networks were run in series:

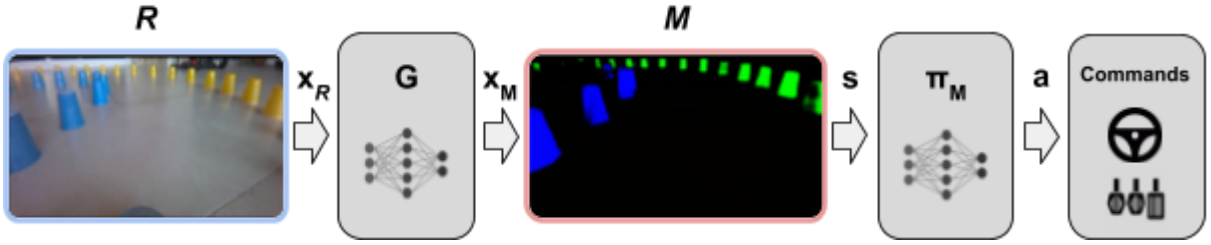

Figure 73. Driving controller execution flow.

As explained in the "Software Control" section, in the case of the driverless mode, Neural Networks were run in an offboard computer due to an issue in the installation of TensorFlow 1.15 in the Jetson Nano. A complete overview of the control architecture for the robot can be seen on figure 74.

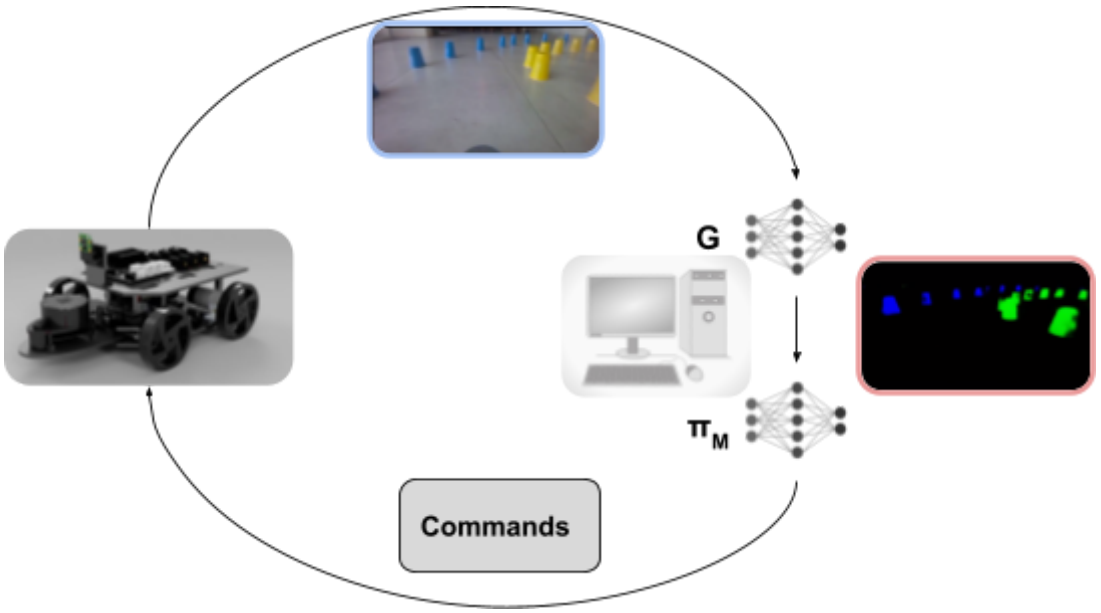

Figure 74. Robot in Driverless Mode

Similarly to how it was done in the "Imitation Learning" section, the full control architecture was evaluated in 6 different unseen tracks. In 3 of these tracks, the controller passed the test, while in the 3 remaining it failed. The test was considered successful if the vehicle completed a full lap without running out of the track. Supplementary material showing the performance of the controller is available at the project's Git repository [19].

It is important to recall that both the policy and the cGAN Generator were both trained using synthetic data, which was obtained exclusively using the simulation. This means that none of the Neural Networks saw any real image prior to the evaluation. The evaluation results show how the method

was able to greatly counteract the difference between simulation and reality, as the policy yielded a 70% of effectiveness in simulation and a 50% in reality.

It is hypothesized that this difference in performance is mainly due to an amplification of the limitations of Behavioral Cloning, given by existing segmentation errors. In other words, errors in the Generator's segmentation add noise to the driving's policy input image, which augments the shift between the actual state and the states present in the policy's training dataset. This creates a sequence of magnified compound errors, which increases the probability of collision.

This problem can be addressed mainly by improving the robustness of the driving policy. One way of doing this is to add image noise and perturbations during training, to augment the dataset, and in consequence diminish the difference between the training set and the evaluation set. Another way could be to replace Behavioral Cloning with Reinforcement Learning, or other Imitation Learning method, to directly avoid the mentioned limitations. On the other hand, the Generator could be further analyzed to detect weaknesses and improve segmentation, to further decrease the reality gap. In particular, it appeared to be susceptible to shiny surfaces. This may be due to the fact that glossy surfaces were not modeled in the simulation, and as a result the dataset did not contemplate this characteristic.

Conclusion

This project aimed to develop a self-driving algorithm capable of navigating through an unknown track delimited by cones using only a monocular camera. Based on the results mentioned above, it can be concluded that a model (cGAN + Driving Policy) developed fully in simulation performed well in a real-life track. The architecture proposed, following further analysis and testing, could serve as a viable approach for the actual formula competition. On the other hand, the use of simulation in this project clearly showed the advantages in data acquisition, as well as model validation before testing in the real platform. Furthermore, the robot designed and engineered for this research proved to serve as a simple modular platform capable of future sensor additions and features.

Future Projects/Improvement

Many approaches regarding self-driving algorithms have been left for the future to test and compare their performance.

- Taking into account the API developed for the simulator as an OpenAI gym, Reinforcement Learning could be an area of research and development for more complex and smarter driving policies with fewer limitations than Behavior Cloning.
- The depth estimation obtained from the cGAN could be added as an extra channel in order to train models with more information on top of 2D information coming from a monocular camera.
- An approach with a Stereo Camera could be addressed, as depth estimation can be easily computed via triangulation to sense the proximity of obstacles.

- Sensor Fusion could be an interesting approach, following the idea of taking the strengths from different sensors and combining them into a single representation of the environment. This information could then be fed to any self-driving algorithm.

Besides the control algorithm used to navigate through an unknown track, mapping of cones and its location could be useful to plan a trajectory for future laps and decrease times. This could be done using depth estimation from the cGAN, a LIDAR or stereo camera. Furthermore, vehicle location could be improved by fusing odometry and IMU.

The robot designed for this project should serve as a development platform for future work, taking into account that ROS Melodic was employed. In the case of requiring new updated software, it will require porting to newer versions of ROS, such as Noetic or ROS2.

Bibliography

[1] IMECHE. Formula Student [Online]. Available from: https://www.imeche.org/events/formula-student

[2] FSG. Formula Student Rules [Online]. Available from: https://www.formulastudent.de/fileadmin/user_upload/all/2022/rules/FS-Rules_2022_v1.0.pdf

[3] FSG. Formula Student Germany [Online]. Available from: https://www.formulastudent.de/fsg/

[4] AMZ. AMZ Racing [Online]. Available from: https://electric.amzracing.ch/en/home

[5] Juraj Kabzan , Miguel I. Valls , Victor J.F. Reijgwart, et al. AMZ Driverless: The Full Autonomous Racing System [Online]. Available from: https://arxiv.org/pdf/1905.05150.pdf

[6] Kieran Strobel, Sibo Zhu, Raphael Chang, Skanda Koppula. Low Latency Visual Perception for Autonomous Racing Challenges Mechanisms and Practical Solutions [Online]. Available from: https://static1.squarespace.com/static/5b79970e3c3a53723fab8cfc/t/5e5dd142f31d6249a33bb70b/1583206732393/Accurate__Low_Latency_Visual_Perception_for_Autonomous_Racing__Challenges__Mechanisms__and_Practical_Solutions_.pdf

[7] EUFS. EUFS Team [Online]. Available from: https://www.imeche.org/events/formula-student

[8] Formula Technion. Formula Student Technion Driverless [Online]. Available from: https://github.com/Microsoft/AirSim/wiki/technion

[9] First Principles of Computer Vision. Camera Calibration [Online]. Available from: https://www.youtube.com/playlist?list=PL2zRqk16wsdoCCLpou-dGo7QQNks1Ppzo

[10] Radial Distortion Correction [Online]. Available from: http://www.baspsoftware.org/radcor_files/hs100.htm

[11] Wikipedia. Bayer Filter [Online]. Available: https://en.wikipedia.org/wiki/Bayer_filter

[12] ROS. RQT Graph [Online]. Available from: http://wiki.ros.org/rqt_graph

[13] Gabriel Torre. Kinodynamic Simulator of the Autonomous Formula Student and Sensor System Design.

[14] Chollet F, others. Keras [Online]. 2015. Available from: https://github.com/fchollet/keras

[15] Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al. TensorFlow: Large-scale machine learning on heterogeneous systems [Online]. 2015. Available from: tensorflow.org

[16] Hussein A, Gaber M, Elyan E, Jayne C. Imitation learning: a survey of learning methods [Article]. 2017.

[17] Torabi F, Warnell G, Stone P. Behavioral Cloning from Observation. 2018. Available from: https://arxiv.org/pdf/1805.01954.pdf

[18] Bojarsk M, Del Testa D, Dworakowski D, Firner B, et al. End to End Learning for Self-Driving Cars. 2016. Available from: https://arxiv.org/pdf/1604.07316v1.pdf

[19] Valentin M, Grillo A. Development of an image-based autonomous driving system for an e-FSAE [Online]. 2022. Available from: https://github.com/AgustinGrillo/image-based-fsd

[20] Sutton RS, Barto AG. Reinforcement learning: An introduction. MIT press; 2018.

[21] Brockman G, Cheung V, Pettersson L, Schneider J, Schulman J, Tang J, et al. Openai gym. arXiv preprint arXiv:160601540. 2016;

[22] Tobin J, Fong R, Ray A, Schneider J, Zaremba W and Abbeel P. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World [online]. 2017. Available from: https://arxiv.org/abs/1703.06907

[23] James S, Wohlhart P, Kalakrishnan M, Kalashnikov D, Irpan A, Ibarz J et al. Sim-to-Real via Sim-to-Sim: Data-efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks [online]. 2019. Available from: https://arxiv.org/abs/1812.07252

[24] Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S et al. Generative Adversarial Networks [online]. 2014. Available from: https://arxiv.org/abs/1406.2661

[25] Isola P, Zhu J, Zhou T, Efros A. Image-to-Image Translation with Conditional Adversarial Networks [Online]. 2018. Available from: https://arxiv.org/abs/1611.07004

[26] AMZFormulaStudent. Autonomous Racing: AMZ Driverless with flüela. [video file]. 2017. Available from: https://www.youtube.com/watch?v=FbKLE7uar9Y

[27] Mathworks. Kinematic Steering [Online]. Available from: https://la.mathworks.com/help/vdynblks/ref/kinematicsteering.html

[28] ROS. ROS Control Wiki [Online]. Available from: http://wiki.ros.org/ros_control

[29] Robotics and ROS Learning. ROS Control [Online]. Available from: https://www.rosroboticslearning.com/ros-control

[30] Nvidia Forums. Install tensorflow 1.15 for Python 2.7 [Online]. Available from: https://forums.developer.nvidia.com/t/install-tensorflow-1-15-for-python-2-7/174625

[31] ROS. Image Transport [Online]. Available from: http://wiki.ros.org/image_transport

Annex

List of Materials

	Name	Quantity
Hardware	50:1 Metal Gearmotor 37Dx70L mm 12V with 64 CPR Encoder (Spur Pinion)	2
	Servo 20KG	1
	Arduino UNO	1
	Nvidia Jetson Nano B01	1
	XL4016 Step Down (12V => 5V)	1
	Panasonic NCR 2900mAh (18650)	3
	Adaptor Level I2C (5V - 3.3V)	1
	IMX219-160IR	1
	IMX219-83 Stereo	1
	MPU 9250	1
	VNH5019 Motor Driver Breakout	2
	RP Lidar A1	1
STL Parts	Back_Wheel v2	1
	Back_Wheel v2_M	1
	Front_Wheel v1	1
	Front_Wheel v3_M	1
	Bracket	1
	Camera Mount A	1
	Camera Mount B	1

	Front_Hub	1
	Front_Hub_M	1
	FS_support_rod v3	4
	FS_Support_Steer v2	2
	Mobile_robot_Motor_case	2
	SupportLidar	2
	Rod	1
	SteerSupport	1
	FS_New_Base_case v5	1
	FS_2nd_base_case	1
	Steer_screw	1
	Sup_Nvidia	1
	Sup_Nvidia_2	1
Mounting Elements	M3 x 12mm (Phillips)	8
	M3 x 12mm (Phillips)	12
	M5 x 90mm (Allen)	4
	M3 x 14mm (Phillips)	4
	M2 x 14mm (Allen)	4
	M3 x 16mm (Phillips)	4
	M3 x 35mm (Phillips)	4
	M4 x 16mm (Phillips)	4
	M3 x 12mm (Phillips)	4
	M3 x 20mm (Phillips)	1
	M2 x 6mm (Allen)	4
	M3 x 14mm (Phillips)	2
	Bearing (4mmx9mmx4mm)	2
	Bearing (5mmx10mmx4mm)	6

Table 7. List of Materials