MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Autonomous vehicle simulation in Gazebo

Bachelor's Thesis

LUKÁŠ POUL

Brno, Spring 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Autonomous vehicle simulation in Gazebo

Bachelor's Thesis

LUKÁŠ POUL

Advisor: Mgr. Oldřich Pecák

Brno, Spring 2024


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools:
- Grammarly to improve my writing and for grammar check,
- ChatGPT to improve my writing.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Lukáš Poul

Advisor: Mgr. Oldřich Pecák

Acknowledgements

I want to thank my advisor Mgr. Oldřich Pecák for the consulting and his suggestions for the project. I would also like to thank the faculty's NXP Cup team members for helping with understanding the NXP Cup racecar. Finally, I need to thank my family and girlfriend for their support.

Abstract

The main objective is to create a fairly accurate simulation with tools to enhance the development of controlling algorithms of miniature autonomous racecars for the NXP Cup with methods to adjust the simulation precision compared to its tangible counterpart. The research methodology involves the utilization of the Robot Operating System (ROS) in conjunction with the Gazebo simulation environment. This implementation includes an API enabling testing of existing real-life algorithms on simulation, enabling precise and reproducible tests. The outcome is reasonably accurate to the tangible counterpart and helps design new, reliable, faster driving algorithms. Future research could use machine learning to optimize the driving algorithm or make the simulation more accurate.

Keywords

NXP Cup, Simulation, Autonomous car, Self-driving, Gazebo, ROS, Robotics

Contents

List of Figures

Introduction

Robotics simulation is a digital representation of a real-life robot to test its algorithms [1].

Simulation brings many advantages. Firstly, simulation is cost-effective for several reasons. While it is true that initially developing a simulation incurs an upfront expense, this cost is relatively minor when compared to the alternative [2, 3]. Testing on actual robots poses a risk of physical wear and tear or even catastrophic failure, which can lead to significant expenses for repair or replacement, downtime, and lost productivity [4]. In contrast, a simulation allows for extensive testing and iteration without the risk of damaging expensive equipment. This capability to refine and troubleshoot in a virtual environment saves substantial money in the long run, making the initial investment in simulation development highly cost-effective [1, 2]. Secondly, it brings massive advantages for developing robots in complex scenarios that might be challenging to recreate. For example, robots can be simulated in space, underwater, or other hazardous environments without exposing the robot or human operators to risk [4]. Thirdly, simulations are crucial in developing artificial intelligence (AI) for robotics. They provide a platform for training and testing AI algorithms [5]. Lastly, multiple people can simultaneously work on the robot's software and algorithms, even on different parts of the robot.

Simulation is not without its drawbacks. One of the main limitations of simulation is its accuracy, which is heavily dependent on how well it replicates real-world situations. The accuracy of a simulation may be compromised by inaccuracies in the physical properties or sensor behaviors it is based on, leading to significant discrepancies between simulated and real-world performance. Additionally, creating a simulation can be a time-intensive process that requires significant computational power [2]. Also, relying too heavily on a simulation can create a false sense of security for the operators.

This bachelor's thesis is motivated by the issues and inefficiencies of developing, debugging, and testing driving algorithms for the autonomous NXP Cup racecar. NXP Cup is a competition to determine the fastest autonomous miniature car on a simple track. Those problems included a lack of debugging data, guessing why it made wrong

decisions, wondering why it detected a finish line, chasing the car on foot, and taking the car back to the start. The Gazebo simulation API created in this paper allows quicker prototyping, quicker testing and provides debugging data. It also allows the creation of a track scenario, which is hard to create in real life. For example, last year's final track can be recreated, and lap times compared to the winners.

1 Software for robotic simulation

1.1 Gazebo

Gazebo is as a robotics simulation software within the robotics community, offering a sophisticated platform for designing, testing, and training robots across diverse indoor and outdoor settings, thereby mitigating the risk of physical damage. This software facilitates the simulation of an array of robotic scenarios, ranging from autonomous vehicles and drones to legged robots, employing highly accurate and physics-based modeling to achieve this end [6].

Distinctively, Gazebo can simulate robots interacting with various environmental elements under assorted conditions, including but not limited to changing weather, varying lighting, and different terrains. This capability is indispensable for the advancement of robotics research and development. The platform's support for multiple physics engines, such as ODE, Bullet, and Simbody, ensures dynamic and realistic interactions with the environment. Furthermore, Gazebo's ability to generate sensor data, including data from cameras, lidar, and GPS, is critical for robotics research, facilitating sensor fusion experiments and AI model training [6, 7].

A key strength of Gazebo lies in its extensibility, offering a plugin architecture that enables developers to broaden their capabilities and integrate with other software tools, rendering it a versatile option for many robotic applications. Additionally, as an open-source platform, Gazebo fosters a collaborative environment for developers globally to share, modify, and enhance the software, thereby contributing significantly to the advancement of robotic technologies [6].

1.1.1 Approach to simulation

In terms of operation, Gazebo commences with the setup of a virtual simulation environment, where users can either design their scenarios using the tools within Gazebo or import pre-built world files that encompass specific settings and models. This highly customizable environment includes detailed terrains, multiple objects, and dynamic lighting changes [6].

At its core, Gazebo conducts simulations that accurately mirror real-world physics laws, supported by its compatibility with multiple physics engines. These engines play a crucial role in simulating the motion and interaction of objects within the simulation, thus providing insights into how robots would execute tasks in real-world scenarios, considering factors such as friction, collisions, and material properties [6].

Gazebo emerges as a comprehensive tool for the robotics community, providing a safe, controlled, and cost-effective environment for developing and testing the next generation of robots and their components [6].

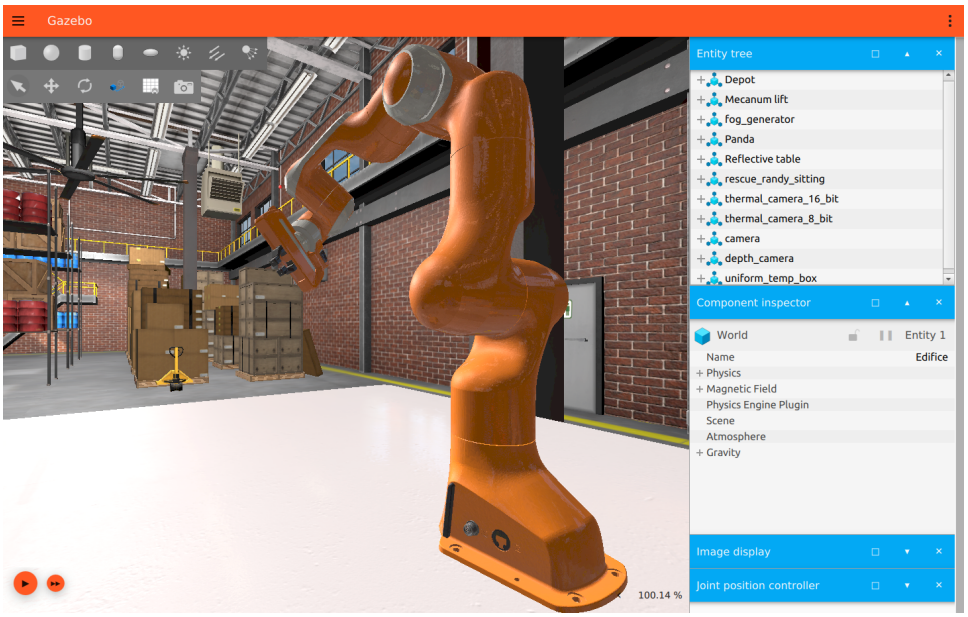

Figure 1.1: Example simulation in Gazebo [8]

1.1.2 Sensors

Another critical functionality of Gazebo is its capacity to simulate sensor outputs encountered by robots in real-life operations, encompassing data from visual, depth, motion, and environmental sensors. By delivering realistic sensor feedback, Gazebo enables roboticists to test algorithms and systems designed for data processing, essential

for navigation, object recognition, and environment mapping tasks [6, 8].

1.1.3 Control

Robots within the Gazebo environment can be controlled using external programming scripts or through direct interaction within the simulator, facilitated by integrating with popular robotic middleware, such as ROS (Robot Operating System). This middleware is instrumental in programming robotic behaviors, processing sensor data, and implementing control systems, thereby supporting complex robotic operations within the simulated environment [6, 8].

Gazebo offers real-time feedback on the simulations, which is crucial for iterative testing and development. Users can modify robot designs, adjust environmental parameters, and refine control algorithms based on the outcomes observed in the simulation, optimizing robot performance before their final deployment in real-world applications [6, 8].

1.2 ROS

The Robot Operating System (ROS) is a flexible framework for writing robot software, primarily designed to enhance the reusability and interoperability of software components across diverse robotic platforms. In the context of its relationship with Gazebo, a prominent simulation software, ROS serves as a crucial conduit for simulating, testing, and optimizing robotic algorithms and behaviors before their real-world deployment. [9]

1.2.1 Integration with Gazebo

ROS and Gazebo communicate in this symbiotic relationship via a publish-subscribe and service-call mechanism. ROS nodes (the basic execution units in ROS) can publish and subscribe to topics (TCP/IP sockets) and invoke simulated services within Gazebo. For example, a ROS node can publish commands to control a virtual robot's movement in Gazebo or subscribe to sensory data simulated by Gazebo that reflects environmental interactions [6, 8, 10].

A critical aspect of the ROS-Gazebo integration is the simulation feedback loop. As ROS sends commands to Gazebo to manipulate the robot model, Gazebo provides real-time feedback to ROS, simulating how sensors would react and how physical forces would affect the robot's movement. This feedback is crucial for tuning controllers and developing fault-tolerant systems under diverse operational conditions, such as different terrains or weather scenarios [6, 8, 10].

1.2.2 Toolchain

ROS enhances this relationship by providing tools and libraries that simplify configuring and running simulations in Gazebo. This includes visualization tools like RViz, which integrates with Gazebo to provide real-time visualization of the robot's state and the surrounding environment, as well as diagnostic tools that help analyze and debug the interaction between the robot's software and its simulated hardware [11].

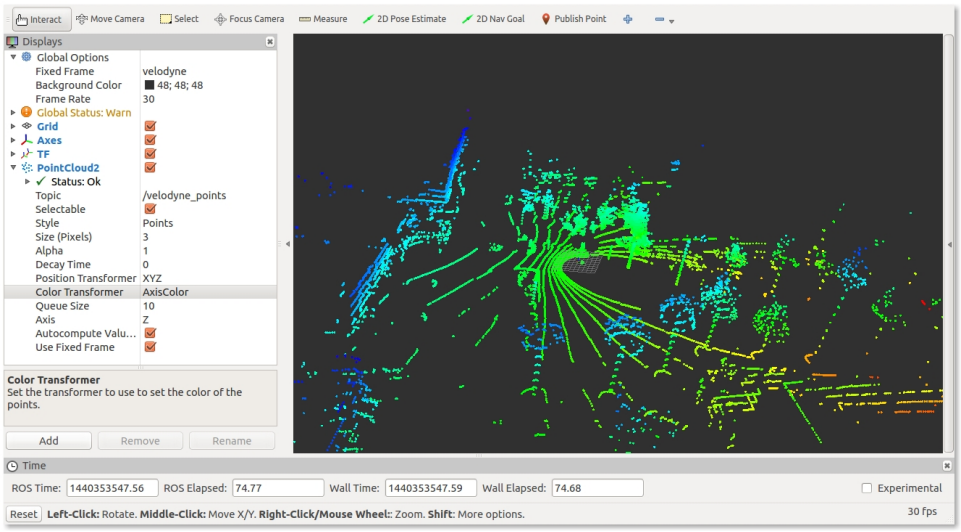

Figure 1.2: RViz example [12]

2 NXP Gazebo

The decision was made to leverage the Gazebo models and simulations provided by NXP [13] and customize them to meet the specific needs of the faculty's car. NXP Gazebo is also simulating Pixy Camera [14], which has its own chip and detects lines in the image, as can bee seen in bottom-left corner in Figure 2.1.

NXP had initially recommended that the setup be implemented on Ubuntu 20, utilizing versions of Gazebo and ROS that are now considered outdated. Initial attempts to install everything on Ubuntu 22 were met with failure. Consequently, acquiring a different computer with Ubuntu 20 was arranged, and numerous hours were invested in the project. Progress was made, even though slowly, it was getting close to making it work. However, the encounter with a lack of documentation and the presence of unresolved errors, for which solutions seemed missing on the internet, led to a key decision.

The majority of the resources provided by NXP were abandoned, a switch to Ubuntu 22 was made, and, importantly, upgrades to newer versions of Gazebo and ROS were made. The project was then tackled almost from the ground up.

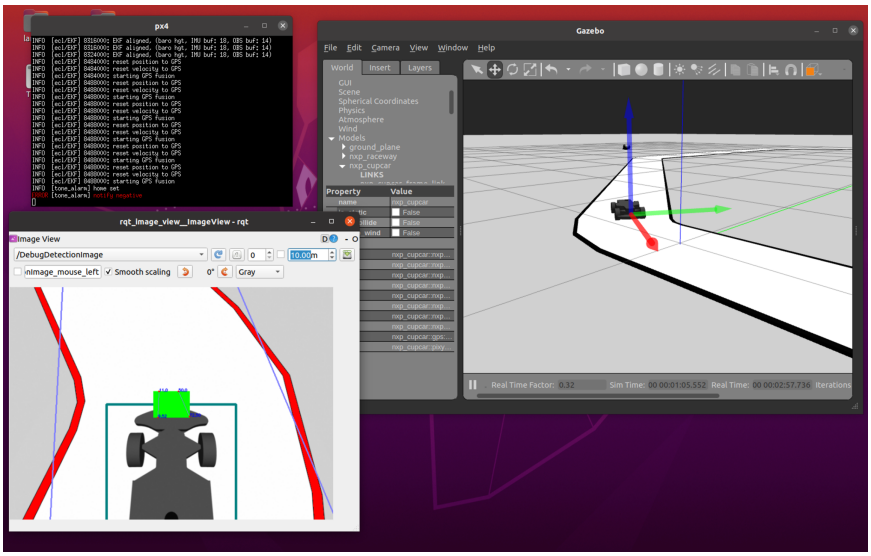

Figure 2.1: NXP Gazebo example [13]

3 Implementation in Gazebo and ROS

3.1 Car model

As the base of the simulation car model provided by NXP, MR-Buggy3 [15] was used. Multiple parameter adjustments were needed, which will be mentioned later. Unfortunately, the model is larger than the faculty's car (around 130%), and cannot be easily scaled down, because of that everything else has to be scaled up to match the size. The model itself uses Gazebo plugins for control. It used an outdated version of Ackermann steering plugin, which have been utilized for overall car control - driving and steering. Ackermann steering is a geometric arrangement in vehicles that allows the front wheels to turn at different angles, ensuring that all wheels follow concentric circles during a turn to minimize tire wear and improve handling [16], as shown in Figure 3.1. The plugins communicate with ROS through topics. They either subscribe or publish information. The API needs an accurate control of steering and driving. The plugin has been updated to a newer version and changed to control only the steering of the front wheels, which allows proper control over the steering angle, just like controlling a servo in real life.

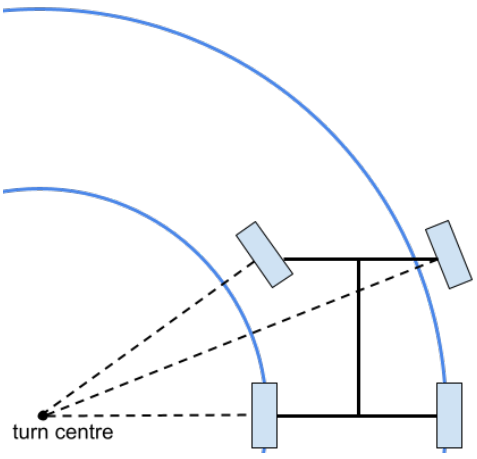

Figure 3.1: Ackermann Steering

The plugin is used in the car model's .sdf file. There are important parameters so that the steering works as expected, as well as a topic set to which commands are sent. The important parameters include wheelbase, steering angle, wheel radius and topic to subcribe to.

The next part was accurate control of the rear wheels. I have used another plugin to simulate rear differential and drive rear wheels. A car's rear-wheel differential is a mechanical device that allows the rear wheels to rotate at different speeds, which is essential for smooth turning and improved traction on varying road surfaces. Even though this plugin is meant to control Differential Drive vehicles (driving without steering any wheels). Important parameters include maximum velocity, acceleration, braking and two topics: one to receive commands to control the wheels and one to publish the odometry. The plugin configuration can be seen in Figure 3.2 with its parameters.

```
<plugin
  filename="gz-sim-diff-drive-system"
  name="gz::sim::systems::DiffDrive">
  <left_joint>MR-Buggy3/RearLeftWheelJoint</left_joint>
  <right_joint>MR-Buggy3/RearRightWheelJoint</right_joint>
  <wheel_separation>0.2</wheel_separation>
  <wheel_radius>0.044</wheel_radius>
  <min_linear_velocity>-4</min_linear_velocity>
  <max_linear_velocity>4</max_linear_velocity>
  <min_linear_acceleration>-4</min_linear_acceleration>
  <max_linear_acceleration>4</max_linear_acceleration>
  <topic>/cmd_vel</topic>
  <odom_topic>/diffodom</odom_topic>
</plugin>
```

Figure 3.2: DiffDrive plugin configuration

3.2 ROS Connection

The connection between Gazebo and ROS is done using the so-called ROS bridge. ROS bridge is a communication interface facilitating interaction and synchronization between these two platforms. When running the bridge, the types of messages have to be stated for both platforms and have to be interchangeable [17].

When first trying to connect these platforms after installing the newest versions of both Gazebo and ROS, there was a problem with the bridge; the messages did not go through because there was no version for the newest Gazebo and ROS (at the time of development). A step-down by one version was needed. Luckily, this did not cause problems with the plugins used for the controlling of the car, as there were no changes in the plugin between these versions.

In the Figure 3.3 is command to run the ROS bridge.

```
ros2 run ros_gz_bridge parameter_bridge /cmd_vel@
   geometry_msgs/msg/Twist]gz.msgs.Twist
```

Figure 3.3: Bash command to run a single bridge

In the command from Figure 3.3, we can see the topic, highlighted in blue, and also the type of message ROS sends, highlighted in green, and is converted to Gazebo variant, highlighted in orange. Also a direction of the bridge has to be specified, which in this case is by the square bracket between the types highlighted in red, or if two-way bridge is needed an at-sign can be used. Alternatively, when multiple bridges need to be run simultaneously, a config file in YAML format can be used, as shown in Figure 3.4, and a different command can be run. It is also more readable and the direction is easy to understand.

The configuration file can be run using the command from Figure 3.5. At this point, the Gazebo simulation is ready to be controlled using ROS.

3.3 Controlling API

ROS supports two programming languages: Python and C++. The decision was straightforward, as the racecar code is written in C++. NXP Cup racecar relies mostly on the image of its line camera, and the algorithm controls the car according to the camera data. The API's objectives are to provide camera image retrieval, current speed retrieval, drive and turn command transmission, debugging data storage, and easy extension for other methods

```
- topic_name: "/cmd_vel"
  ros_type_name: "geometry_msgs/msg/Twist"
  gz_type_name: "gz.msgs.Twist"
  direction: "ROS_TO_GZ"

- topic_name: "/NPU/image_sim"
  ros_type_name: "sensor_msgs/msg/Image"
  gz_type_name: "gz.msgs.Image"
  direction: "GZ_TO_ROS"

- topic_name: "/steer_angle"
  ros_type_name: "std_msgs/msg/Float64"
  gz_type_name: "gz.msgs.Double"
  direction: "ROS_TO_GZ"
```

Figure 3.4: Part of bridges configuration file

```
ros2 run ros_gz_bridge parameter_bridge --ros-
   args -p config_file:=./bridges-conf.yaml
```

Figure 3.5: Bash command to run a configuration file with multiple bridges

The first task was getting the camera image. Using ROS C++ libraries, mage subscriber has ben created that, when started, continuously updates the last retrieved image in its thread. In this case the image is also converted to OpenCV type Mat. Then, when the API method getImage gets called, the last image retrieved is returned.

In this fashion, another subscriber classes can be created. Another classes are available to extend the API, such as for retrieving simulation time, car speed, and car position.

These classes do not work on their own. They are used in the main API class - Spinner. The class is the manager, and its primary objective is to connect all the subscribers and publishers into a simple-to-use API.

Spinner creates a thread for each subscriber class, as they retrieve the data continuously. Publishers are easier to manage because they do not require their own thread. A simple diagram from Figure 3.6 shows how the Spinner is structured.

Control methods also allow for controlling normalization; for example, the real racecar accepts values from -512 to 512 to control the servo, but the simulation only accepts values from -0.5 to 0.5, and turn directions are switched. The methods also allow for preprocessing the data; for example, noise is added to the camera image to reflect real cameras more and to force the image processing algorithm to be more reliable.

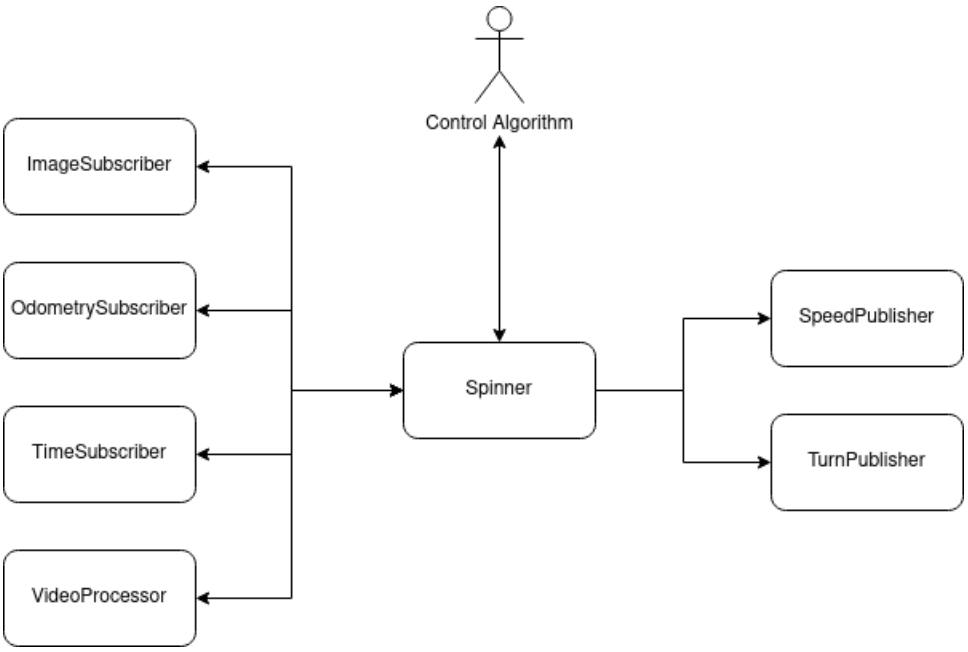

Figure 3.6: API Diagram

The diagram from Figure 3.6 reveals that the API class Spinner creates an abstract layer above the complex parts of the control using ROS.

3.4 Debugging data

Comprehensive data is essential for creating and optimizing autonomous driving algorithms, including understanding the decision-making process of autonomous cars. The API provides a solution by offering access to valuable data and leverages OpenCV[1] to create videos showcasing all the relevant information.

Initially, the idea was to generate a single image of the video each time the *getImage* method was called. However, implementing this would slow the method's response time and negatively impact the simulation. Since generating a video is a computationally expensive task, it's crucial to ensure that the API control methods required for the simulation aren't slowed down.

The video is being processed in a new thread that collects the data approximately 60 times per second. This solution should not slow the control methods. The video footage is initialized in the constructor of Spinner and saved in the destructor, so it does not require any control from the user.

An example image of the output video is in Figure 3.7. The top left corner contains a full image of the camera, and next to it on the right is the line (stretched) used for the control algorithm. The API itself fills the left column of text data as it has access to this data. The control algorithm also has essential data that the API cannot access. Other API methods were created to add data to the video for better debugging. One method is to add processed image, and another is for text data. Optionally, under the camera line is another camera line received by the algorithm, and under that are all received text data that can be easily extended. As mentioned before, there is an added camera noise.

The API also provides another form of visualization of debugging data - it saves valuable data to a CSV file, which the user can visualize using the provided Python script leveraging the matplotlib[2] library. An example is shown in Figure 3.8.

1. `https://opencv.org/`
2. `https://matplotlib.org/`

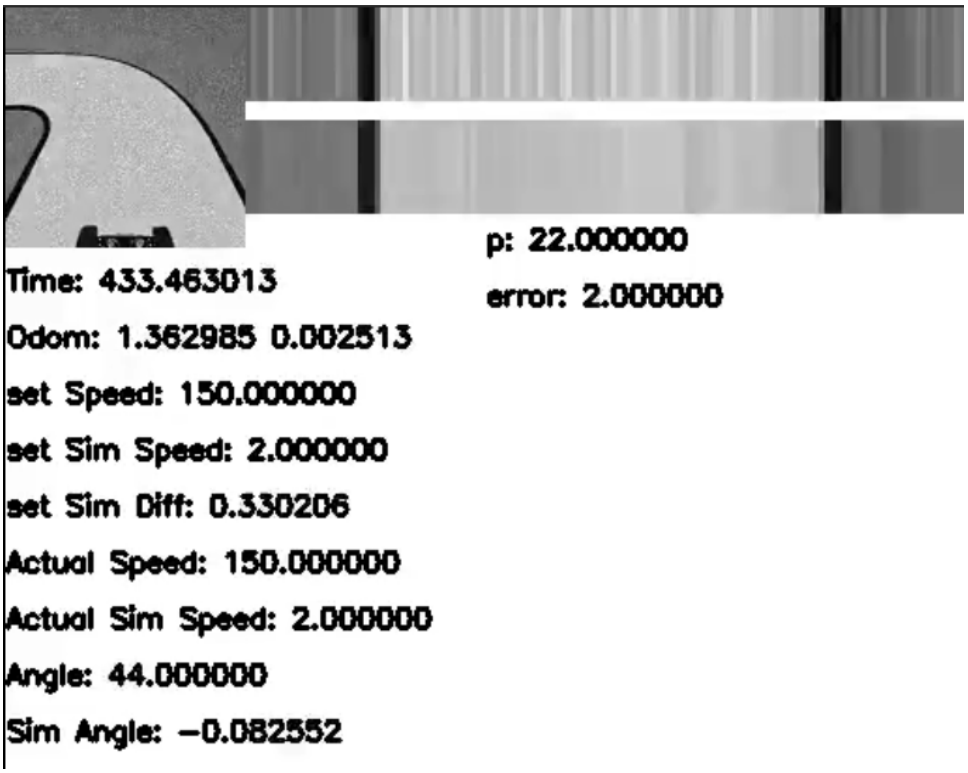

Figure 3.7: Example image of the video output

4 Track Generation

Racetrack options are crucial for proper NXP Cup car simulation algorithm testing. Although NXP has developed some track models, the options are minimal. Therefore, a track-generating script was necessary to have the ability to test various track scenarios to ensure the algorithm's speed and reliability.

4.1 Gazebo model format

Gazebo models are digital versions of physical objects, machines, and even robotic systems in the Gazebo simulator. These models range from simple shapes like boxes and spheres to highly intricate and complex robotic systems with sensors, actuators, and control systems.

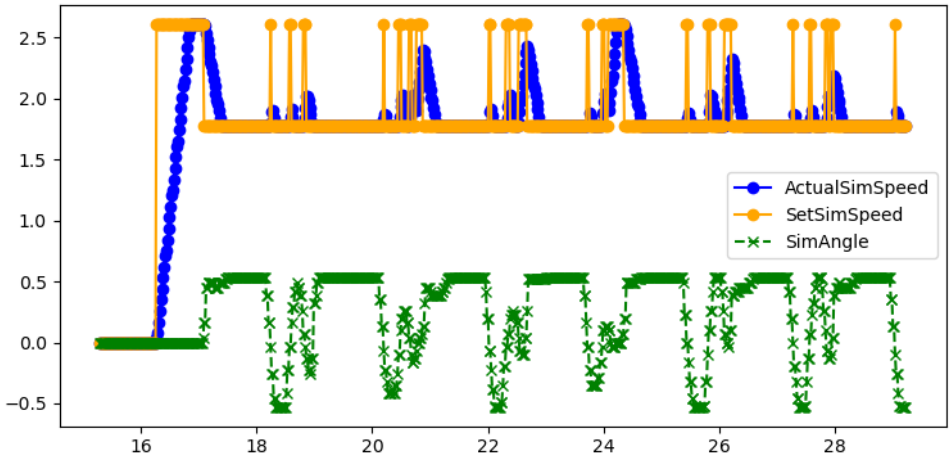

Figure 3.8: Data visualization

Models in Gazebo are not just static representations but dynamic entities with physical properties such as mass, friction, and texture [6].

The key features of Gazebo models include modularity, physical properties, sensors, actuators, programming, and control. Models in Gazebo can be constructed from smaller components like links and joints, allowing users to create highly detailed and articulated structures that behave realistically. Models in Gazebo simulate real-world physics, including gravity, inertia, collision, and other dynamics that affect how objects interact. These interactions are calculated using the DART physics engine [18].

Models in Gazebo can be equipped with various sensors like LIDAR, cameras, and GPS, as well as actuators such as motors and servos to mimic real robotic systems. These elements are essential for developing and testing robotics algorithms, including navigation, perception, and environmental interaction. Models can be controlled programmatically through APIs in popular languages such as C++ and Python. This allows for the simulation of autonomous systems and testing control algorithms under various conditions.

Standard model formats in Gazebo include SDF, URDF (Unified Robot Description Format), and mesh formats. SDF is the native configuration for Gazebo, providing detailed specifications for physics

properties, sensor configurations, and more [19]. URDF is commonly used with ROS (Robot Operating System) and defines the mechanical structure of robots, which can be imported into Gazebo [20]. Mesh formats like COLLADA (.dae) and STL are used to create detailed visual representations of the models.

The STL format is a widely used file type in 3D printing that describes the surface geometry of a three-dimensional object without any representation of color, texture, or other common CAD model attributes, using a series of triangular facets [21].

COLLADA (.dae) is an XML-based file format that facilitates the exchange of detailed 3D models, including their visual aspects, animations, and physical properties, among various digital content creation applications [22].

4.2 NXP Cup track

The NXP Cup track is build out of 6 different piece types. These include straights, short straights, intersections, curves, chicanes, and start/finish. You can create any track using these pieces. The middle of the track is white with black borders, and the high contrast helps the cars distinguish the track's edge.

4.3 Script for track generation

There was a need to create more diverse tracks to enhance the testing of driving algorithms for the NXP Cup, leading to the development of a new tool for track creation.

The decision was made to use the STL format for designing tracks because of a familiarity with this format, even though it presented a unique challenge compared to using something like COLLADA. The challenge was increased due to the need to create separate STL models for the insides and borders of the track pieces to allow them to have different colors in the simulation, making the task more complex.

Initially, the work started with using Tinkercad[1] for its simplicity and accessibility, allowing for the quick creation of most track pieces.

1. `https://www.tinkercad.com/dashboard`

However, the complexity of creating a chicane track piece required changing the CAD software for a more sophisticated, FreeCAD[2], to meet the specific geometric needs. The chicane piece's complexity is shown in Figure 4.1.

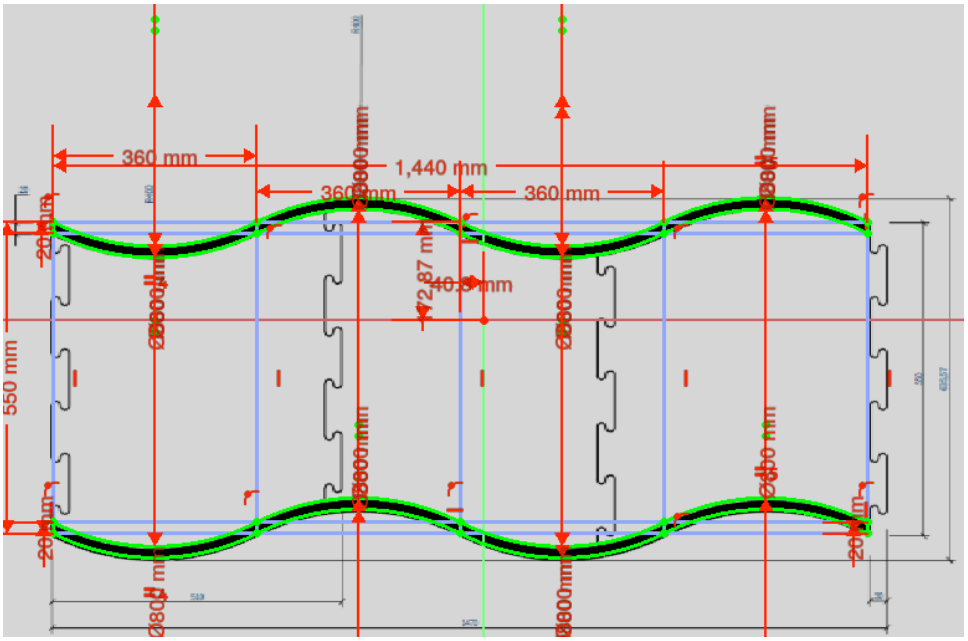

Figure 4.1: Chicane track piece sketch in FreeCAD

The effort then focused on making the track generator user-friendly, aiming to replicate the real-life process of adding track pieces sequentially to complete a circuit. An example of the input in the Python script and the resulting track layout are shown in Figures 4.2 and 4.3.

Initially, the plan was for the furthest edge of the current track to be identified using an algorithm and aligned with the right edge of the piece. The mesh would then be rotated, and its position would be adjusted accordingly. However, this method would have needed a complex algorithm to identify the furthest edge accurately. Instead, the decision was made to store this point in memory and calculate it relative to the measurements of the previous piece.

To create the script, Python was chosen for its ease of use and range of libraries, specifically stl and numpy. The script processes each

2. `https://www.freecad.org/`

```
gen.add_piece(CROSSROADS)
gen.add_piece(STRAIGHT_SHORT)
gen.add_piece(TURN_LEFT)
gen.add_piece(TURN_LEFT)
gen.add_piece(STRAIGHT)
gen.add_piece(STRAIGHT)
gen.add_piece(TURN_LEFT)
gen.add_piece(TURN_LEFT)
gen.add_piece(TURN_LEFT)
gen.add_piece(TURN_LEFT)
gen.add_piece(STRAIGHT)
gen.add_piece(STRAIGHT)
gen.add_piece(STRAIGHT_SHORT)
gen.add_piece(CROSSROADS)
gen.add_piece(STRAIGHT_SHORT)
gen.add_piece(TURN_RIGHT)
gen.add_piece(TURN_RIGHT)
gen.add_piece(TURN_RIGHT)
gen.add_piece(TURN_RIGHT)
gen.add_piece(TURN_RIGHT)
gen.add_piece(TURN_RIGHT)
gen.add_piece(STRAIGHT_SHORT)
```

Figure 4.2: Example of user input in the Python script

piece type the user enters, loads the mesh and rotates it if necessary to ensure the closest point on the right edge can be accurately identified. It then calculates the translation vector between the furthest point on the right edge of the track and the piece and rotates the mesh around the corresponding point to maintain continuity. The rotation is centered around the found point to ensure the translation vector is precise. After that, the mesh of the last piece is translated, and the furthest right point on the edge is calculated. The same logic works for the border. The exact process is repeated for every piece entered.

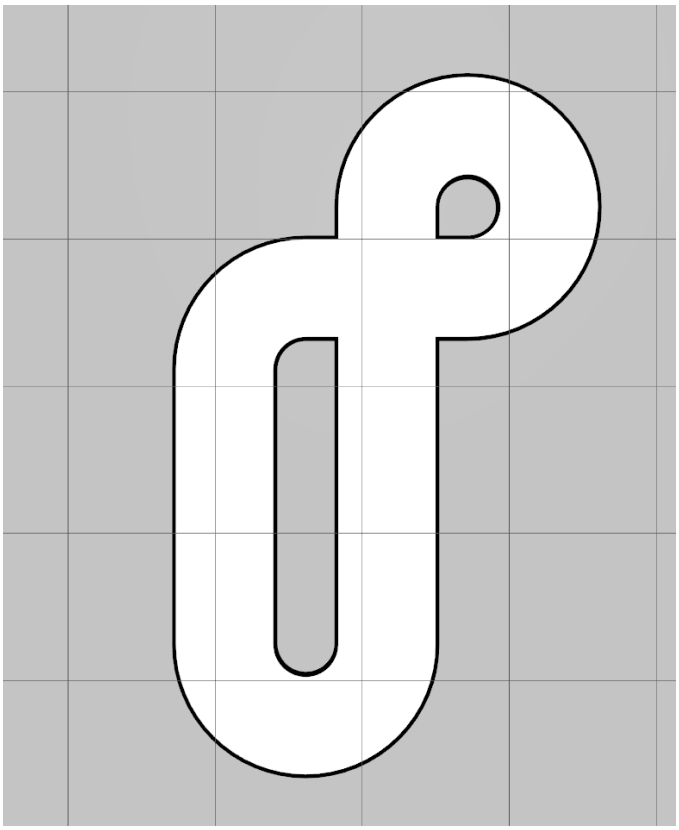

Figure 4.3: Example track generation result

5 Verification and calibration

An essential part of the simulation is its accuracy compared to its rear counterpart. However, it is vital to note that achieving perfect accuracy is not the primary goal of creating this simulation. Instead, the focus is on striking a balance between realistic representations and computational feasibility, ensuring the simulation remains a valuable tool for development and testing without being bogged down by the pursuit of perfection. The main goal is to create an accurate simulation sufficient for quicker algorithm prototyping, finding mistakes, and making the car faster and more reliable.

5.1 Camera

The camera sensor is an essential part of the simulated car, but luckily it was quite simple to get accurate. The crucial parameters were position, FOV, framerate, resolution, and the camera format.

5.1.1 Framerate

The framerate should precisely match the real car's expected framerate. It can be hard to determine the framerate of a real car's camera as it depends on current lighting conditions. Minimizing the difference between the two framerates is important to avoid significant deviations in algorithm behavior.

5.1.2 Camera image

Image width and height represent the camera's resolution. The real car's line camera has 128 pixels of grayscale image. Even though the real car and its control algorithm only use one line of the image, seeing the 2D image for debugging was informative. However, only one line has to be picked to use. The faculty's NXP Cup team uses the

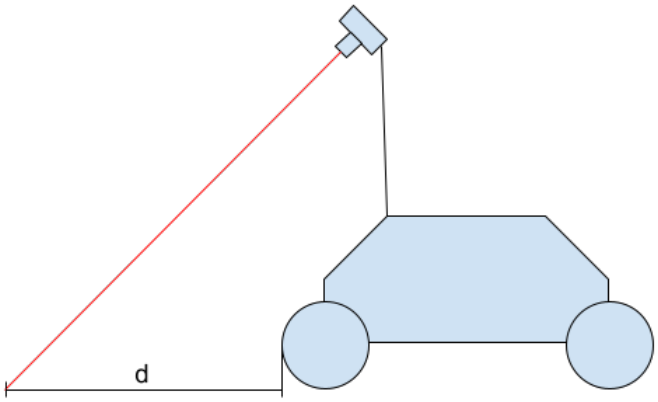

Figure 5.1: Representation of line selection

distance from the front bumper to set the correct line camera angle as represented in Figure 5.1. To unite with this idea, the simulated car

was manually moved back by the camera's desired distance (distance d in Figure 5.1, adjusted for the scale) from the finish line, and then, using the API, every line was checked.

```
int main(int argc, char* argv[]) {
    rclcpp::init(argc, argv);
    Spinner s;
    for (int i = 0; i < 128; i++) {
        s.addData({"image␣row:␣" + std::
           to_string(i)});
        s.getImage(i);
        usleep(100000);
    }
    return 0;
}
```

Figure 5.2: Test case for camera angle using API

After running code in Figure 5.2, the automatically generated video clearly shows the desired line index, this can be seen in Figures 5.3 and 5.4. This means the algorithm should get the image line on the index 60 to stay accurate to the real car.

Figure 5.3: Part of the debugging video - finish line is not seen

Figure 5.4: Part of the debugging video - finish line is seen

5.2 Velocity

Another essential part of the simulation is velocity accuracy. There are multiple simple tests for all the parts of controlling the velocity. The API is used, which makes it quite simple.

5.2.1 Driving speed

When going a constant velocity, the velocity has to be normalised to be the same for both the real car and simulated car. The tests are simple:

First, the real car is accelerated to a certain speed, after reaching the speed it is kept at the same speed for a certain amount of time. The distance covered in the time window is measured. The distance d in figure 5.6 represents the distance measured in this case, *s1* represents start of the timer, when reaching constant speed and *s2* stop of the timer, and braking. This test is repeated multiple times with different speeds set.

At this point it is necessary to find the speed function according to the data measured previously. In the case of faculty's NXP Cup car the function was linear.

Secondly, the test case has to be created in the simulation using the API.

Using the function in Figure 5.5 the case can be tested, pay attention to the *norm* function which adjusts for the scale difference of the simulated car and the real car. All that is left to do is to adjust the normalization function for the distance values to be the same with real car, possibly even maximum velocity in the model's .sdf file.

```
float norm(float dist) {
    return dist * 0.774193548;
}

void test_distance(Spinner& s, int speed) {
    auto time = s.getTime();
    s.setServo(0);
    s.setSpeed(speed);
    while(s.getTime() - time < 1){}
    time = s.getTime();
    auto [sx, sy, sz] = s.getPosition();
    while(s.getTime() - time < 2.0){}
    auto [ex, ey, ez] = s.getPosition();
    s.setSpeed(0);
    auto distance = norm(std::sqrt(std::pow(ex -
        sx, 2) + std::pow(ey - sy, 2)));
    std::cout << "speed: " << speed << ",
       distance: " << distance << std::endl;
}
```

Figure 5.5: Velocity test case

5.2.2 Acceleration

Real car acceleration calibration is done on such surface, which offers the best traction to eliminate wheel slip. In simulation, the traction of all wheels is set to high values.

This calibration case assumes that the previous cases are already calibrated. The car is accelerated from a stationary position, and the distance covered in a certain amount of time is measured. The distance measured is represented by a and d in Figure 5.6, where the car starts to accelerate, and the timer starts at $s0$; when the timer hits $s2$, the car slows down. The distance of constant velocity is also used because it is hard to know when the real car stops accelerating and has already reached the desired velocity.

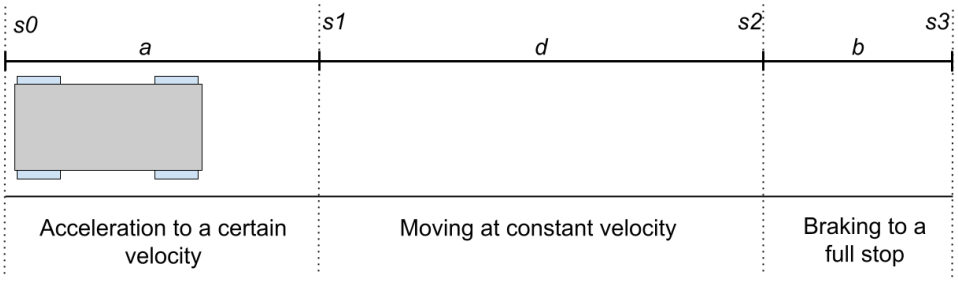

Figure 5.6: Test cases representation

This calibration case code for the simulation is created by analogy to the previous calibration. The test is run multiple times while adjusting the acceleration value in the model's .sdf file.

5.2.3 Braking

This calibration test is going to be done under the same circumstances as the previous case - with the best amount of traction possible to eliminate wheel slip. The car is accelerated to a certain speed and then brakes to come a to stationary position. Again the distance covered is measured, represented as *b* in Figure 5.6, and calibrated. Calibrating is done by adjusting the minimal linear acceleration in the model's .sdf file.

5.3 Turning

This section talks about how to set up a car in a simulation so that its turning dynamics are similar to the real car. It's important to get this right to make the simulation feel real. We look at how to make the car's turning very accurate.

5.3.1 Turn Angle

When using the Ackermann steering plugin in Gazebo, amongst other important parameters there is a steering limit. Steering limit is the maximum angle the wheels can reach relative to the car's body. The calculation depends on the car's wheelbase and turning radius.

$$\delta = \arctan\left(\frac{L}{R}\right) \tag{5.1}$$

The steering limit is calculated using the formula 5.1 [23], where R represents the inner turning radius, and L represents the car wheelbase - the distance between the rear and front wheels. It is important to adjust for the scale difference when making these calculations; otherwise, the resulting turning dynamics will differ. The result is an angle in degrees, which needs to be converted to radians for the Ackermann steering plugin in Gazebo. Once set, the steering limits of both the simulated and real cars should be similar.

5.3.2 Rear Differential

The rear wheel differential significantly affects the car's turning capabilities. The rear wheel differential is crucial in a vehicle's handling, especially during turning. This mechanism allows the two rear wheels to rotate at different speeds, essential for smooth and stable turns. Without a differential, the wheels would be forced to spin at the same rate, causing friction and potentially leading to tire wear or even losing control on tight turns. In a turn, the outside wheel has to travel a greater distance than the inside wheel. The differential compensates for this difference, distributing torque to the wheels in a way that allows them to rotate at speeds proportional to the turn radius. This improves the car's handling, enhances traction, and reduces strain on the vehicle's drivetrain [24].

Creating a reasonably accurate simulation of a car's rear-wheel differential is crucial. In Gazebo, the DiffDrive plugin is used to control the rear wheels, even though it was originally designed for differential-drive vehicles that do not have steering wheels. This is because no plugin was available to simulate a car's rear-wheel differential. The plugin requires linear and angular velocity as its input. If the angular velocity is not controlled, the car's dynamic would simulate having no rear differential.

$$\mathrm{R} = \frac{\mathrm{L}}{\sin(\delta)} \tag{5.2}$$

$$\omega = \frac{v}{R} \tag{5.3}$$

Using the vehicle's wheelbase (L) and its current turning angle (δ), the turn radius according to the current turning angle can be calculated using formula 5.2 [25]. Then using the current velocity (v) and the turning radius calculated in previous step, the angular velocity ω can be calculated using formula 5.3 [26].

Even though the outcome is fairly accurate, it is just an approximation of the behavior of a vehicle with differential steering. This method does not account for factors such as wheel slip, suspension dynamics. This calculation has to be done everytime the turning angle or velocity changes for the differential to stay accurate. This calculation is set by default using the *diff_norm* function, but can be changed by passing a different calculation method to the Spinner's constructor.

5.4 Traction

In the development of simulations, it is crucial to ensure the accuracy of traction parameters. Traction significantly influences a vehicle's handling, acceleration, and deceleration characteristics. A substantial deviation in traction values can lead to markedly different behavior in the simulated vehicle. Consequently, it is essential to establish methodologies that enhance the precision of traction representation within the simulation to mirror real-world dynamics accurately.

Simulating traction accurately can be a challenging task. However, a simplified methodology has been developed which provides reasonably accurate results.

The process begins with an educated guess. The car is placed on an NXP Cup track piece with its wheels stopped. The track piece is then lifted from one end so that the car is facing down the slope. The angle between the piece and the ground is measured as it is slowly lifted until the car starts slipping down the slope. This angle measurement provides an indication of the car's traction on the track. This measurement is depicted in Figure 5.7.

$$\mathrm{m} = \tan\theta \tag{5.4}$$

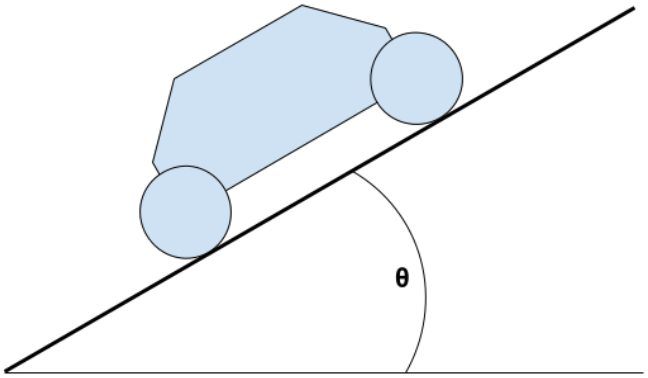

Figure 5.7: Traction measuring representation

The friction *mu* parameters for the wheels in the model's .sdf file are calculated based on the angle measurement obtained using formula 5.4 [27]. To determine the perpendicular friction *mu2* parameter, the measurement process is repeated with the car rotated 90 degrees from the previous test [28]. This helps make an educated guess about the friction parameter.

After making those educated guesses, the simulation and the real car are run, and the parameters are iteratively adjusted according to the dynamic difference between the two. The wheels' friction can be adjusted separately, for example, when the rear wheels are wider than the front wheels.

5.5 Weight

Weight is also essential for the accuracy of the simulated car dynamics. It is simple to get accurate.

$$m = w \cdot s^3 \tag{5.5}$$

Using 5.5, where w is weight of the real car, s is the scale, the simulated car weight is calculated.

6 Final evaluation

The simulation provides reasonably accurate simulation. Thanks to the simple test cases to enhance precision, the simulation is accurate enough for the needs of the faculty's NXP Cup team. This level of accuracy is key for developing and refining algorithms without needing real-world testing. The evaluation scenario entailed replicating an identical track in both real life and simulation, with a smaller track due to limitations such as the restricted number of track pieces and the feasibility of capturing the desired camera angle. A comparative analysis of the paths driven by the car, as depicted in Figure 6.1 , highlights the algorithm's current challenge with oscillation on the track, a phenomenon attributed to the camera's low frame rate. This issue, observable in both real and simulated environments, underlines the fidelity of the simulation.

However, the simulation has its limitations; one such limitation is the absence of an engine for accurate tire simulation, which would account for grip, slip, and wear. Additionally, the simulation lacks aerodynamic modeling, as the Gazebo platform does not support aerodynamic simulations.

Despite these limitations, the reliability of the simulation opens up opportunities for refining existing algorithms and pioneering new ones. Moreover, any modifications to a real car must also be mirrored in the simulation, including but not limited to enhancements in camera frame rate, alterations in wheel types, or motor modifications. This meticulous approach ensures that the simulation remains a valuable tool for algorithm development, offering a balance between practical feasibility and accuracy.

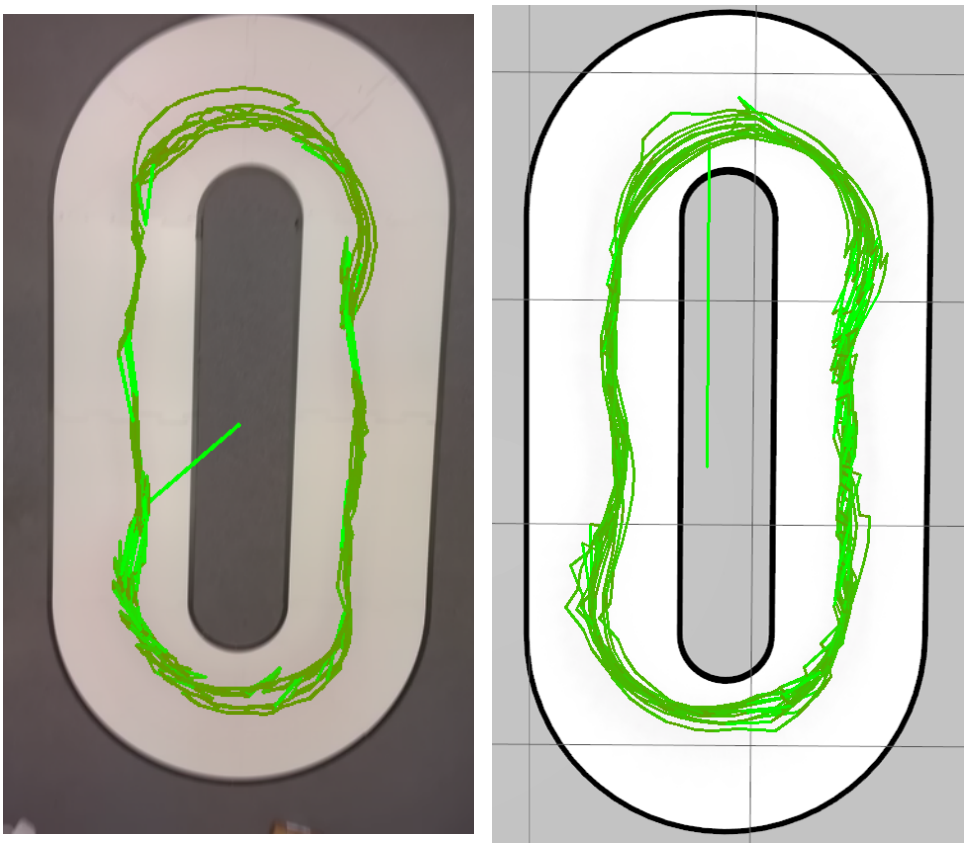

Figure 6.1: Path comparison - real on the left, simulated on the right

7 Conclusion

This thesis presents a detailed exploration and development of a Gazebo-based simulation environment specifically tailored for the NXP Cup autonomous miniature racecar. By integrating Gazebo with the Robot Operating System (ROS), the research successfully creates a robust platform replicating real-world conditions to test, refine, and speed up driving algorithms.

The setup of the simulation environment is a significant improvement for the development of control algorithms. The environment not only supports testing the vehicle's response under various conditions but also enables the adjustment and refinement of algorithms in a risk-free setting. This aspect of the research is crucial, as it allows for iterative development without the high costs or potential risks associated with physical prototypes.

A key feature of the thesis is the development of an advanced control API within ROS. This tool provides precise control over the simulated vehicle. It facilitates real-time debugging and data analysis, which is essential for deepening the understanding of autonomous driving behaviors and improving algorithm reliability.

Additionally, generating custom tracks and scenarios is beneficial, as it greatly expands the testing capabilities beyond what can typically be achieved in physical setups.

Despite these advancements, the thesis acknowledges several challenges and limitations. The accuracy of the simulation in mimicking real-world physics and vehicle dynamics, though high, is not without discrepancies. These differences highlight the perennial challenge in simulation work: ensuring that successful simulations accurately translate into real-world efficacy.

The research opens up several fascinating avenues for further investigation. By incorporating machine learning strategies, there is a significant potential to boost the adaptability and efficiency of algorithms designed for autonomous racing. Furthermore, enhancing the realism of the simulations could yield more precise data, which would be invaluable for development objectives.

The simulation helps the faculty's team to improve and stay competitive.

Bibliography

1. CHOI, HeeSun; CRUMP, Cindy; DURIEZ, Christian; ELMQUIST, Asher; HAGER, Gregory; HAN, David; HEARL, Frank; HODGINS, Jessica; JAIN, Abhinandan; LEVE, Frederick; AL., et. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*. 2020, vol. 118, no. 1. Available from DOI: 10.1073/pnas.1907856118.
2. AFZAL, Afsoon; KATZ, Deborah S; GOUES, Claire Le; TIMPERLEY, Christopher S. A study on the challenges of using robotics simulators for testing. *arXiv preprint arXiv:2004.07368*. 2020.
3. COLLINS, Jack; CHAND, Shelvin; VANDERKOP, Anthony; HOWARD, David. A review of physics simulators for robotic applications. *IEEE Access*. 2021, vol. 9, pp. 51416–51431.
4. ŽLAJPAH, Leon. Simulation in robotics. *Mathematics and Computers in Simulation*. 2008, vol. 79, no. 4, pp. 879–897.
5. EL-SHAMOUTY, Mohamed; KLEEBERGER, Kilian; LÄMMLE, Arik; HUBER, Marco. Simulation-driven machine learning for robotics and automation. *tm-Technisches Messen*. 2019, vol. 86, no. 11, pp. 673–684.
6. *Gazebo: Robot simulation made easy*. [N.d.]. Available also from: https://gazebosim.org/. [Accessed 27-04-2024].
7. STAN, Alexandru; OPREA, Mihaela. Applied learning of artificial intelligence techniques by using the Gazebo simulator and Turtlebot3 multi-robot system. [N.d.].
8. *Github - Gazebo*. [N.d.]. Available also from: https://github.com/gazebosim. [Accessed 27-04-2024].
9. *ROS: Home — ros.org* [https://www.ros.org/]. [N.d.]. [Accessed 29-04-2024].
10. *Topics - ROS Wiki — wiki.ros.org* [http://wiki.ros.org/Topics]. [N.d.]. [Accessed 1-05-2024].
11. *rviz - ROS Wiki — wiki.ros.org* [http://wiki.ros.org/rviz]. [N.d.]. [Accessed 3-05-2024].

12. JOSEPH, Lentin. *ROS Robotics Projects — oreilly.com* [https://www.oreilly.com/library/view/ros-robotics-projects/9781838649326/1850d8b2-00b7-4715-853d-f871521fcec4.xhtml]. [N.d.]. [Accessed 03-05-2024].

13. *NXP Gazebo*. [N.d.]. Available also from: https://nxp.gitbook.io/nxp-cup/gazebo/. [Accessed 03-05-2024].

14. *PixyCam* [https://pixycam.com/]. [N.d.]. [Accessed 6-05-2024].

15. *MR-Buggy3* [https://app.gazebosim.org/RudisLaboratories/fuel/models/MR-Buggy3]. [N.d.]. [Accessed 04-05-2024].

16. VENERI, M; MASSARO, M. The effect of Ackermann steering on the performance of race cars. *Vehicle system dynamics*. 2021, vol. 59, no. 6, pp. 907–927.

17. *Integration between ROS and Gazebo simulation* [https://github.com/gazebosim/ros_gz]. [N.d.]. [Accessed 3-05-2024].

18. *DART: Dynamic Animation and Robotics Toolkit — dartsim.github.io* [https://dartsim.github.io/]. [N.d.]. [Accessed 20-05-2024].

19. OSRF. *SDFormat Home — sdformat.org* [http://sdformat.org/]. [N.d.]. [Accessed 8-05-2024].

20. TOLA, Daniella; CORKE, Peter. Understanding URDF: A survey based on user experience. In: *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2023, pp. 1–7.

21. *STL (file format) - Wikipedia*. Wikimedia Foundation, 2024. Available also from: https://en.wikipedia.org/wiki/STL_(file_format). [Accessed 19-05-2024].

22. BARNES, Mark. Collada. In: *ACM SIGGRAPH 2006 Courses*. 2006, 8–es.

23. MINH, Vu. VEHICLE STEERING DYNAMIC CALCULATION AND SIMULATION. In: 2012.

24. *How Differentials Work — auto.howstuffworks.com* [https://auto.howstuffworks.com/differential.htm]. [N.d.]. [Accessed 20-05-2024].

25. *Turning radius - Wikipedia*. Wikimedia Foundation, 2023. Available also from: https://en.wikipedia.org/wiki/Turning_radius. [Accessed 9-05-2024].

26. *Angular velocity - Wikipedia*. Wikimedia Foundation, 2024. Available also from: https://en.wikipedia.org/wiki/Angular_velocity. [Accessed 12-05-2024].

27. ARIEL BALTER, Ph.D. *How to calculate the coefficient of friction*. 2020. Available also from: https://sciencing.com/calculate-coefficient-friction-5200551.html. [Accessed 12-05-2024].

28. OSRF. *Gazebo : Tutorial : Friction — classic.gazebosim.org* [https://classic.gazebosim.org/tutorials?tut=friction]. [N.d.]. [Accessed 13-05-2024].

A Appendix

A.1 List of attachments

- *implementation.zip*
- *example-video.mp4*
- *debug-output.mp4*

A.2 API method summary

```
Spinner(
    std::optional<std::function<float(float,
       float)>> speed_norm = std::nullopt,
    std::optional<std::function<float(float,
       float)>> speed_denorm = std::nullopt,
    std::optional<std::function<float(float,
       float)>> turn_norm = std::nullopt,
    std::optional<std::function<float(float,
       float)>> diff_norm = std::nullopt
    )
```

Spinner is the API Class, that optionally accepts functions to normalize the controls. They are optional, because there are already defined default functions. The reason for the functions is that the values from the algorithm might not be correctly mapped to the simulation's controls. For example real racecars algorithm uses values between -512 and 512 for steering, but the simulation uses values between -0.5 and 0.5 and the direction is switched. All the functions have speed as the first argument and turn angle as second, they do not necessarily use both. Important function is also diff_norm, because it controls how much the rear differential is used, diff_norm takes already normalised speed and angle. Using the differential function angular velocity is calculated, which represents the rate at which the car must turn to achieve the desired turn angle at the given speed. The Spinner constructor also starts all the publishers, subscribers and initializes the video processing thread.

```
void setServo(float angle, std::optional<float>
    speed = std::nullopt)
```

Method used to control the turning. Takes both angle and optionally speed, because the turn normalization function needs both. If speed is not provided, the speed from last call of *setSpeed* is used. After normalizing the message is published.

```
void setSpeed(float speed, std::optional<float>
    angle = std::nullopt)
```

Method used to control the rear wheels. Takes both speed and optionally angle, because the turn normalization and differential function need both. If angle is not provided, the angle from the last call of *setServo* is used. It is recommended to provide the angle for better simulation accuracy. This method also controls the differential.

```
rightWheelSpeed = (linearVelocity +
    angularVelocity * wheelSeparation / 2.0) /
    wheelRadius;
leftWheelSpeed = (linearVelocity -
    angularVelocity * wheelSeparation / 2.0) /
    wheelRadius;
```

This is how each wheel's speed is calculated where linearVelocity is the speed given to the method and angularVelocity is calculated using the differential function.

```
void setSpeedAndServo(float speed, float angle)
```

Method recommended to use instead of *setSpeed* and *setServo* methods. The method ensures effective control.

```
cv::Mat getImage(int line = -1, bool add_noise =
    true)
```

Method used to retrieve image out of the simulation car's camera. Returns image in OpenCV format - cv::Mat. Has two optional parameters - line index and an flag, if to add noise to the camera's image. If line index is not relevant than the whole image is returned. Camera noise

is added by default to simulate real cameras. This method also saves last retrieved image, to be used for the debugging video.

```
float getSpeed()
```

Method used to retrieve current car speed, the speed is not always equal to set speed, as the car also needs to accelerate and break. This method uses speed denormalizing function from the constructor and thus returns relevant speed for the control algorithm.

```
std::tuple<float, float, float> getPosition()
```

Method for retrieving cumulative position of the simulation car.

```
float getTime()
```

Method for retrieving current simulation time, as the simulation can run slower than real time.

```
void addData(const std::vector<std::string>&
   data)
```

Method for adding text Data to the debugging video, each vector member is on one line, in the right column of the debugging image.

```
void addData(const cv::Mat& img)
```

Method for adding processed image line by the algorithm to the debugging video. Method expects one image line.

```
void addNoise(cv::Mat& img)
```

Method for adding artificial camera noise to the image passed as a reference. Is used by default in method *getImage*

A.3 How to run

A.3.1 Recommended: Ubuntu 22

Install using `bash install.sh`.

Open 3 terminals:

1. *bash run.sh* runs Gazebo
2. *bash run-bridges.sh* runs bridges between Gazebo and ROS
3. *mkdir -p cpp/build && cd cpp/build && cmake .. && make* builds the example algorithm.

After building, run `./example`, the car in the gazebo window should start moving

The simple example algorithm in cpp/example_control.cpp can be changed and optimized.

A.3.2 Docker

It is not recommended due to unpredictable bugs in OpenCV in docker.

```
sudo xhost +local: # used to allow local
   connections to the X server
cd docker && sudo docker compose up
sudo docker exec -it ros_gazebo_container bash #
    has to be run in 3 different terminals
cd /gazebo # in each docker container terminal
```

In 3 different docker terminals:

1. *bash run.sh* runs Gazebo
2. *bash run-bridges.sh* runs bridges between Gazebo and ROS
3. *mkdir -p cpp/build && cd cpp/build && cmake .. && make* builds the example algorithm.

After building, run `./example`, the car in the gazebo window should start moving

The simple example algorithm in cpp/example_control.cpp can be changed and optimized.

A.4 Essentials

Track generator generates the files in the current directory, they are not immediately used. The files have to be moved to *models/generated_raceway/meshes* and the Gazebo has to be restarted.

After making changes to any Gazebo model's .sdf file, the Gazebo has to be restarted, for it to load the changes made.