RoboRacer: LiDAR to Predict Autonomous Vehicle Behavior

Maile Campbell, Sean Cooper, Julian Tamayo
Course: Machine Learning for Engineering Applications I
Instructor: Dr. Megan Chiovaro
Date: December 4, 2025

Table of Contents

1. Introduction

2. Dataset Description
   2.1. Data Collection
   2.2.1 Variables
   2.2.2 Constants

3. Data Cleaning and Preparation
   3.1. Missing variables
   3.2. Outliers
   3.3. Feature Elimination and Extraction

4. Model Training and Evaluation
   4.1. Model Training: Phase I
   4.2. Model Training: Phase II, Improvement
   4.3. Model Training: Phase III, Gradient Boosting
   4.4 Model Evaluation

5. Results

6. Conclusion and Future Work

7. References

8. Appendix A

9. Appendix B

1. Introduction

As three electrical engineering students interested in robotics and autonomous systems, we aimed to incorporate machine learning principles that used autonomous driving ideas in our project. Autonomous vehicles in general rely on their ability to interpret sensor data and convert it into safe driving actions, and understanding how much of this behavior can be learned directly from perception is an important question in robotics.

In this project, we investigate whether a machine learning model can predict two core vehicle control outputs (steering angle and vehicle speed) using only LiDAR data collected from an F1-Tenth–style robot in simulation. Our goal is to determine how effectively a 'shallow' machine learning model can reconstruct this driving simulated policy, what kinds of feature engineering improve performance, and what these results reveal about the learnability of autonomous vehicle behavior from LiDAR data alone.

2. Data Description

2.1 Data Collection

Our dataset was generated with ROS2 (Robot Operating System 2)[1], which runs a simulation environment recommended by the official RoboRacer platform. The simulator creates a F1-Tenth-style mobile robot placed on a simple map, modeling a car on a track. To prepare our dataset, we created a Python script to autonomously drive the car around the track using the Follow The Gap method–a reactive path planning algorithm that keeps the car from hitting obstacles without using Machine Learning principles.

A custom ROS2 node sampled all 1080 LiDAR beams at each timestep and wrote every 20th ray into a CSV, producing 54 LiDAR distance values per row. Each row also contained the robot's corresponding:

- steering angle
- vehicle speed
- heading angle
- angular velocity

and other state variables and constants at any given timestep.

Since the robot followed a fixed controller, we obtained a clean paired dataset of LiDAR to (steering angle, vehicle speed) suitable for supervised learning.

1. Appendix A – Information on ROS2

2.2.1 Variables

Below is a list of all columns present in the original dataset, regardless of whether they were incorporated into the model training process.

- `timestamp`: The time at which each variable was recorded
- `scan_number`: The number of scans
    - starting from 0
- `vehicle_linear_y`: Vehicle's linear speed in the vehicle's y direction
    - always 0 because our vehicle can't move in it's y coordinate frame
- `vehicle_linear_x`: Vehicle's linear speed in the vehicle's x coordinate frame
    - same as vehicle_speed
- `vehicle_speed`: Combination of linear x and y value
    - because y is always 0 in our case, vehicle_speed = vehicle_linear_x
- `vehicle_angular_z`: Turning speed of the vehicle with respect to the vehicle's z axis.
- `pos_x`: Position of the vehicle along the x-axis of the simulation's coordinate frame
- `pos_y`: Position of the vehicle along the y-axis of the simulation's coordinate frame
- `steering_angle`: The vehicle's intended orientation in relation to its zero axis
- `heading_angle`: The vehicle's actual orientation in relation to its zero axis
- `range_r_angle_a`: The Lidar ray index: range r and angle a
    - r is an integer from 0-1080, a is an angle from angle_min to angle_max; each range increment represents a ray as its own feature

2.2.2 Constants

- `angle_min`: The minimum angle that the Lidar scanner can reach (in Radians)
- `angle_max`: The maximum angle that the Lidar scanner can reach (in Radians)
- `angle_increment`: The angle between each Lidar ray that is recorded
- `range_min`: The minimum distance that the Lidar scanner can record
- `range_max`: The maximum distance that the Lidar scanner can record

All constants were removed from the dataset because they would not provide any useful training information to the model.

3. Data Cleaning and Preparation

3.1 Handling Missing and Unimportant Data

Since we got our data from simulation, there were no missing values in the target or in the rays. However, we did have to remove ~300 rows at the beginning of the dataset to account for the lag between when we started recording the data and when the car began moving. These early rows

provided no useful learning signal, so we removed them to prevent the model from learning data from when the car was stationary.

3.2 Outliers

Sometimes, LiDAR readings output very large distance measurements when no obstacle is detected, which manifest in very high distance readings as opposed to realistic distance measurements. However, in our dataset this effect did not occur. The simulator we used normalizes all LiDAR returns to physically plausible ranges, meaning that even when no obstacle is present, the reported distances remain bounded within the sensor's configured maximum distance (reported as `range_max`). Because of this preprocessing performed by the simulation environment, our LiDAR columns did not contain outliers or inflated maximum-range values, and therefore capping was not required.

3.3 Feature Elimination and Extraction

After importing the data, non-essential or redundant columns (such as timestamps, position, and configuration metadata) are dropped to isolate the LiDAR beams and the steering/velocity target. Took out irrelevant features in the dataset like:
'vehicle_linear_y',
'pos_y',
'pos_x',
'vehicle_linear_x',
'timestamp',
'scan_number',
'angle_min',
'angle_max',
'angle_increment',
'range_min',
'range_max',
'num_ranges',
'heading_angle'

Engineered Features – Implemented between model training phases I and II

We implemented feature engineering to provide structured geometric information:

- **min_left:** minimum distance among rays 0–19
- **min_center:** minimum distance among rays 20–39
- **min_right:** minimum distance among rays 40–end

These values summarize where the closest obstacles are, giving the model directional awareness instead of forcing it to learn that structure just from raw rays.

We also added temporal context, after noticing that the model lacked any real way to make a time/previous value dependent prediction.

- **steering_ang_prev** = previous timestep's steering angle

- **speed_prev** = previous timestep's speed

Vehicles exhibit inertia, and the previous control command is often a strong predictor of the next. This significantly improved performance, especially for speed prediction. Finally, we removed the first row after shifting to eliminate NaN values.

More information about feature engineering as model improvement in section 4.2.

4. Model Training and Evaluation

4.1 Model Training: Phase I

We first trained two separate **RandomForestRegressor** models (one for steering, one for speed), as a simple first step in determining data learnability and the overall behavior of a model trained on this data. We chose to train two separate models to prevent interference between the two targets and to allow each model to specialize.

Below is the outcome of RandomForestRegressor:

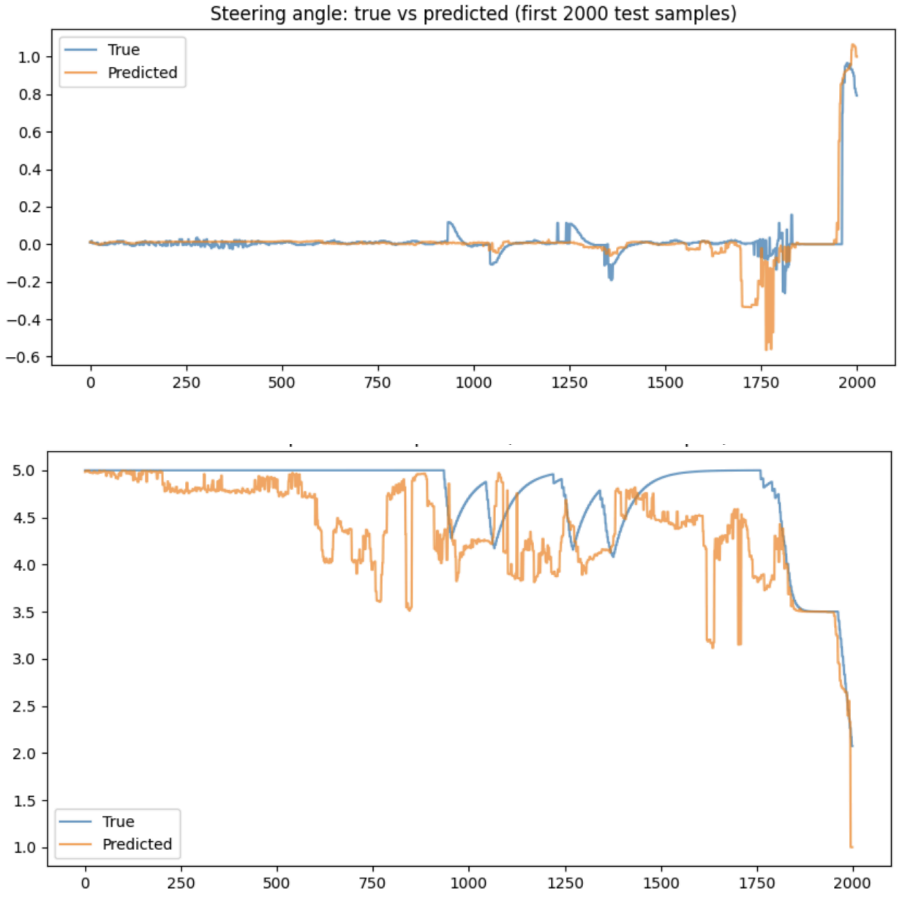


fig. 1

Metrics:

```
Random Forest - Angle MSE: 0.004760129157140778
Random Forest - Angle R^2: 0.6955674457470977
Random Forest - Speed MSE: 0.1903261493096143
Random Forest - Speed R^2: 0.3970301473056864
```

Based on the visual outcomes and metrics of the Random Forests model, we determined that it was able to reveal clear learnable structure, with the following caveats:

- Steering angle predictions captured the general shape but failed during sharp turns.
- Speed predictions were noisier and underfit during transitions.

These baseline models confirmed that LiDAR data contains meaningful predictive information, but also highlighted weaknesses:

- an imbalance of near-zero steering values (straight driving)
- sensitivity to rare turning events
- insufficient temporal context for predicting speed

From this information, we could perform feature engineering in an attempt to decrease the effect of these error-inducing data characteristics. We could also think about implementing a new type of model that would be better suited for our dataset and our project goals.

4.2 Model Training: Phase II, Improvement

After using the Random Forest models to establish a baseline of the model's ability to capture general patterns in the driving behavior, we focused on strengthening the model's ability to handle sharp turns, speed transitions, and other difficult cases. This phase of model training introduced several key improvements designed to give the model better geometric understanding, temporal context (time dependencies), and a more balanced representation of driving behaviors. This next section provides more details regarding the feature engineering touched upon in section 3.3, the implementation of which inevitably contributes the most to the improvement in the outcome of our machine learning model.

First, we added new LiDAR-derived geometric features that summarize obstacle proximity in broad directional regions. Rather than relying just on all 54 raw LiDAR rays, we computed `min_left`, `min_center`, and `min_right`, which give us the minimum distance values within the left, center, and right subsets of beams. These features give the model a compact description of the closest obstacle in each region, making it easier for it to understand when to steer left, right, or straight in response to nearby obstacles.

Next, we introduced previous-step memory to account for temporal continuity in the robot's motion. Vehicle speed and steering angle both exhibit inertia in a real system: the value at time t is often strongly influenced by the value at $t - 1$. To add this dependency effect to our dataset, we added `steering_ang_prev` and `speed_prev`, which store the previous timestep's control values (targets). This significantly improved speed prediction in particular,

since the continuous behavior of speed changes cannot be inferred reliably from a single LiDAR snapshot alone.

Finally, we addressed the heavy imbalance in the steering angle distribution. Because the robot spends most of its time driving straight, near-zero steering angles were vastly overrepresented in the dataset. This caused models to overpredict near-zero values and underperform during turns. To correct this, we downsampled rows where `|steering_angle| < 0.02` in the *training set only*, reducing them to 20% of their original count. This preserved the real test distribution while giving the model a more informative, balanced set of steering examples to learn from.

4.3 Model Training: Phase III, Gradient Boosting

Random Forests average many uncorrelated trees, which can cause underreaction during sharp turns. To address this, we moved to HistGradientBoostingRegressor, which is a fast, histogram-based boosting algorithm well-suited for large numerical datasets.
Gradient boosting:

- builds trees sequentially, correcting errors from prior steps
- captures nonlinear interactions between rays
- and focuses model capacity on the “hard cases” (turns, braking events)

We trained:

- gb_angle on the balanced training set (better turning performance)
- gb_speed on the original training set (preserving original speed distributions)

Outcome of HistGradient Boosting:

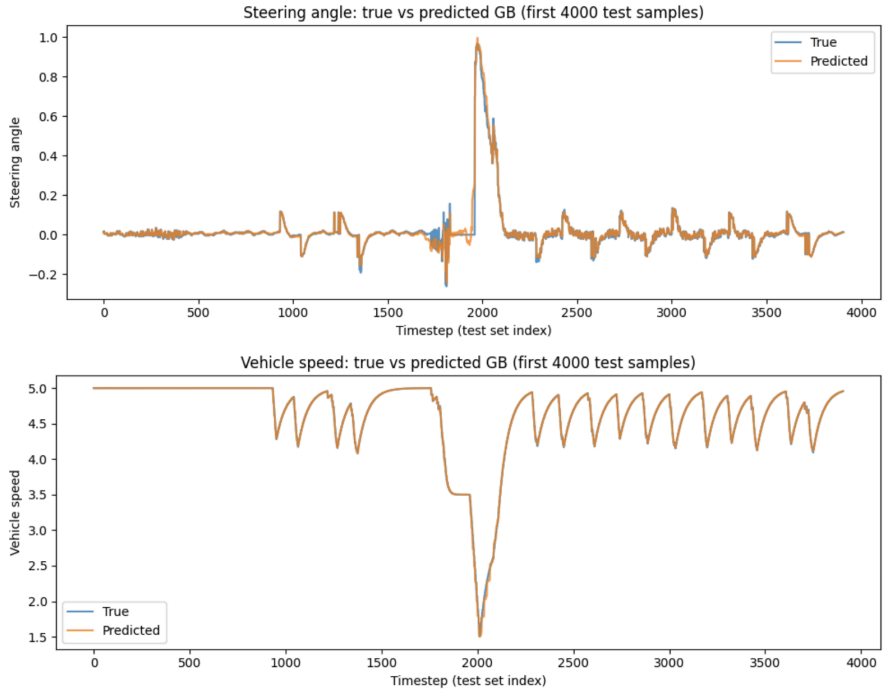


fig. 2

Metrics:

```
HistGradient Boosting performance, angle:
  MSE_GB  : 0.00064
  RMSE_GB : 0.02522
  MAE_GB  : 0.01014
  R^2_GB  : 0.95934

HistGradient Boosting performance, speed:
  MSE_GB  : 0.00057
  RMSE_GB : 0.02383
  MAE_GB  : 0.01123
  R^2_GB  : 0.99820
```

4.4 Model Evaluation

We evaluated performance using:

- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)
- Mean Absolute Error (MAE)
- R^2 score

We used these metrics because they quantify both average prediction error (MAE, MSE, RMSE) and the model's ability to explain variance in the target (R^2).

MAE measures the average absolute deviation between predictions and true values, while MSE and RMSE penalize larger errors more heavily, making them useful for identifying moments when the model fails during sharp turns or sudden speed changes. The R^2 score complements these by measuring how much of the total variance in the target the model can explain; this was especially important for steering prediction because the distribution is dominated by near-zero values. A model could achieve deceptively low error simply by predicting zero most of the time, so R^2 helped us determine whether the model was actually learning meaningful structure rather than exploiting imbalance in the dataset.

5. Results

The final HistGradientBoostingRegressor models showed strong performance in predicting both steering angle and vehicle speed from LiDAR data, with substantial improvements over the baseline Random Forests. After incorporating feature engineering, previous-step memory, and downsampling of near-zero steering values, the steering model achieved highly accurate tracking of the true driving behavior.

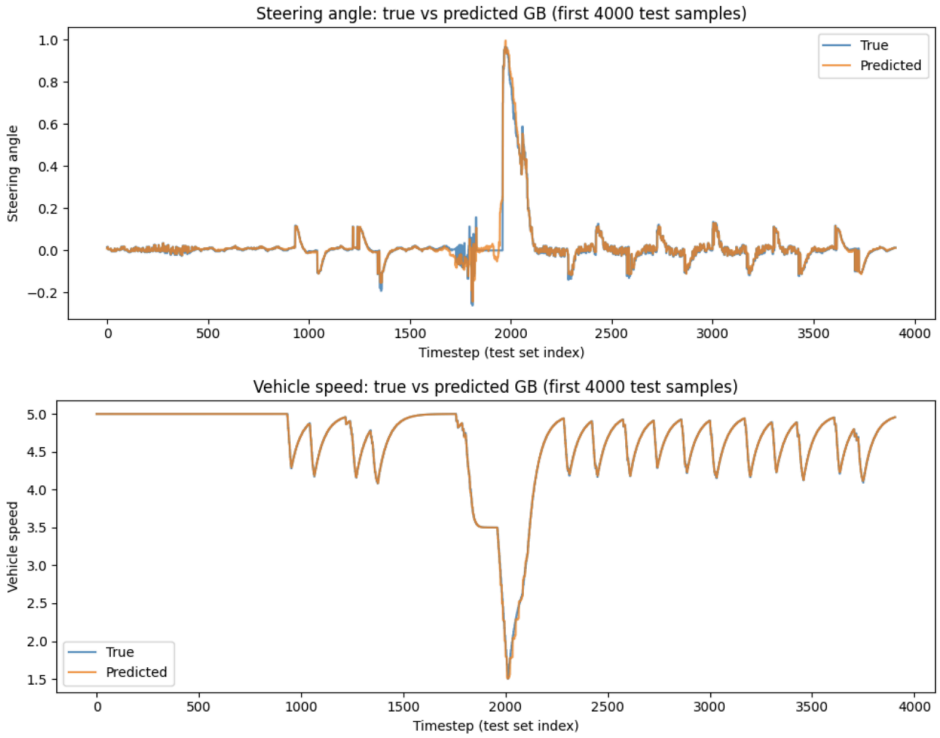


fig. 2

As shown in the top plot, the predicted steering angle closely follows the timing and magnitude of the turns (including the large spike around timestep 2000) while remaining stable during straight segments. The final metrics reflect this precision, with an MSE of **0.00064**, RMSE of **0.02522**, MAE of **0.01014**, and an R^2 score of **0.95934**, indicating that the model captured over 95% of the variance in the steering signal.

The vehicle speed model demonstrated similarly strong improvements. Speed tends to vary smoothly except during sharp turning sequences, which create periodic dips in velocity. The gradient boosting model successfully reproduced the oscillatory structure of the simulation data, including the very deep braking event near timestep 2000 and the repeated slow-down cycles that follow. The predictions match the shape and timing of the true speed curve with only a small amount of instability, a notable improvement over our earlier model. The speed metrics confirm this performance, yielding an MSE of **0.00057**, RMSE of **0.02383**, MAE of **0.01123**, and an exceptionally high R^2 score of **0.99820**, showing that nearly all of the variance in the speed signal was explained by the model.

Overall, the results demonstrate that LiDAR-derived features (especially directional minimum distances and previous-step memory) combine effectively with gradient boosting to reconstruct the robot's control decisions. Steering behavior, which is strongly tied to instantaneous geometry, proved highly learnable, while the addition of temporal context was key to accurately predicting speed. These outcomes show that a significant portion of the driving policy used by the reactive controller can be recovered from perception data alone, highlighting the potential of machine learning to model and replicate autonomous vehicle behavior.

6. Conclusion and Future Work

Our model provided an accurate prediction of our two target variables, vehicle speed and steering angle. The next phase of this project involves integrating what we learned to the F1-TENTH club. Before we can do that, our first priority is to upgrade the model to be more reliable and accurate for brand new race tracks. To do this, we plan to implement a Reinforcement Learning framework to establish robust behavior, which will later be refined and optimized using a neural network. This approach would allow us to gain the ability for real-time processing and decision-making.

The objective is to integrate our model with our autonomous control system to race in competition. This would be particularly effective in the competition's speed challenge where we get to map the track before starting. Our model could provide an advantage by processing the necessary environmental data beforehand. With Roboracer competitions held globally throughout the year (including an upcoming event in Vienna, Austria, in June 2026) we have approximately six months to prepare. Integrating our model will significantly streamline the development process and enhance our vehicle's readiness for competition.

7. References

F1Tenth. (2020). Roboracer.ai. https://roboracer.ai/

f1tenth. (2020). *GitHub - f1tenth/f1tenth_gym_ros: Containerized ROS communication bridge for F1TENTH gym environment.* GitHub. https://github.com/f1tenth/f1tenth_gym_ros

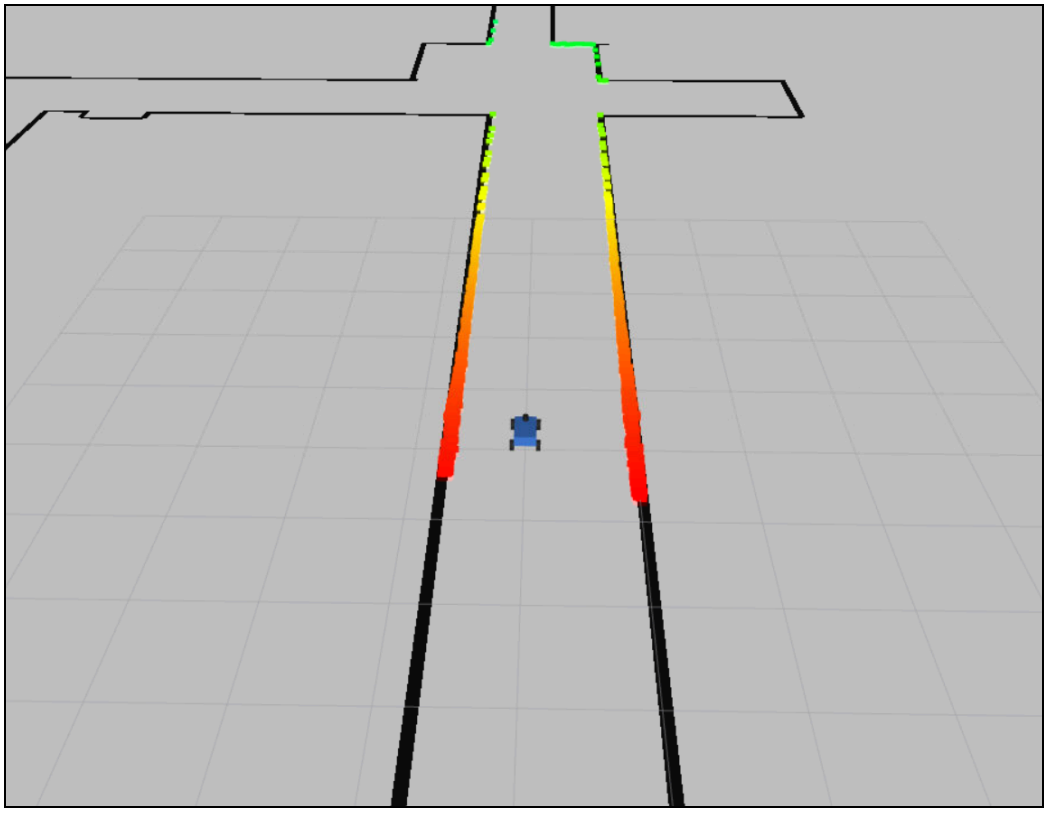
sklearn.ensemble.HistGradientBoostingClassifier. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html

Team, G. (2025). *Gradio docs*. Gradio.app. https://www.gradio.app/docs/gradio/blocks

8. Appendix A – ROS2

ROS2 stands for Robotic Operating System 2, the successor of ROS. We use ROS2 as a hub that connects all of our scripts and simulations for seamless integration and communication. ROS2 allows us to run multiple different python programs using a single run command and this can receive from and send to the simulation.

This is a sample of what the ROS2 robot-track simulation environment looks like.

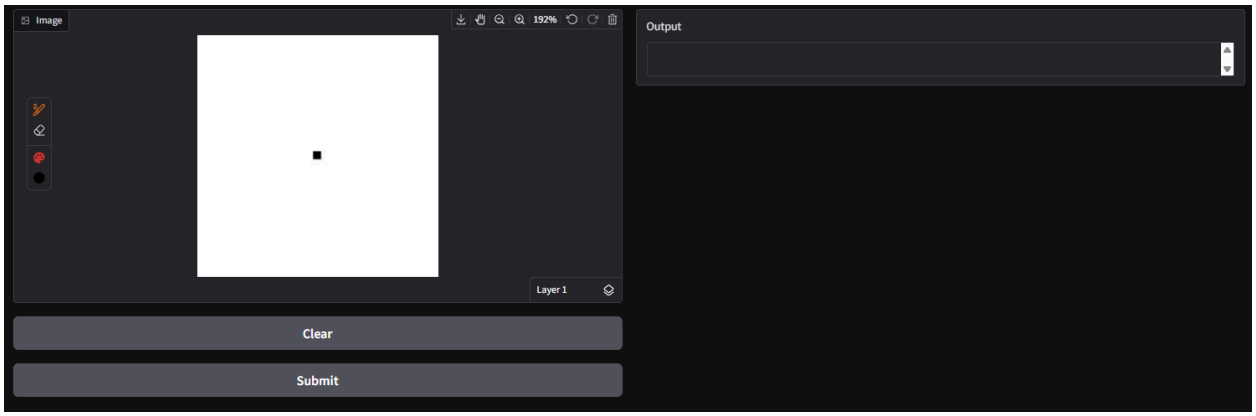


The LiDAR rays are represented in this sim by the colors lining the obstacles (walls), where red corresponds to closer obstacles and green corresponds to farther obstacles.

During simulation, the robot's control node sends velocity or actuator commands just as it would in the real world. The simulator receives those commands, moves the robot model accordingly, and publishes sensor data back to ROS 2. This creates a real-time feedback loop where code reads simulated sensor messages, makes decisions, and sends new commands to the virtual robot. The neat thing about ROS2 is that it provides a controlled, reproducible environment that mimics real robot behavior closely enough that you can test navigation, mapping, perception, and control before ever powering up a physical robot.

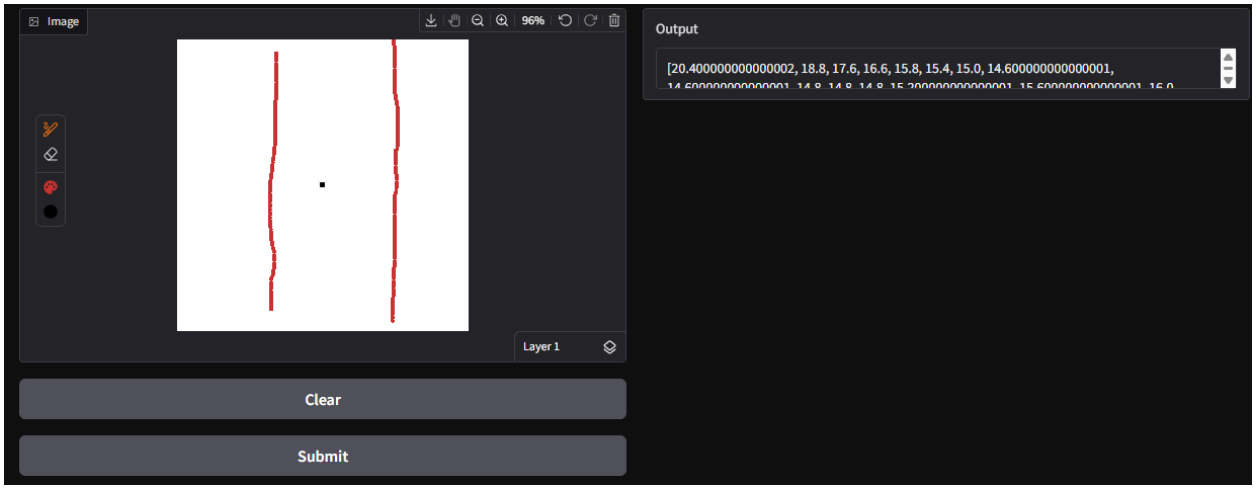
9. Appendix B – Additional Code & Demo Code

Sketchpad Demo Interface:



Gradio sketchpad interface

Using the sketchpad capability of the Gradio library, it was possible to create an interactive UI where the user is able to draw a map. Of which, the user is able to draw a map which the RC car can navigate autonomously. To resemble the vehicle, a black square was added to the middle of the canvas. In addition, the gradio.Blocks() function was used, which is the low level API to allow for a more custom demo. The main reason for using blocks was to customize the clear button. Which when clicked would reset the sketchpad to having the square in the middle instead of being a blank space.

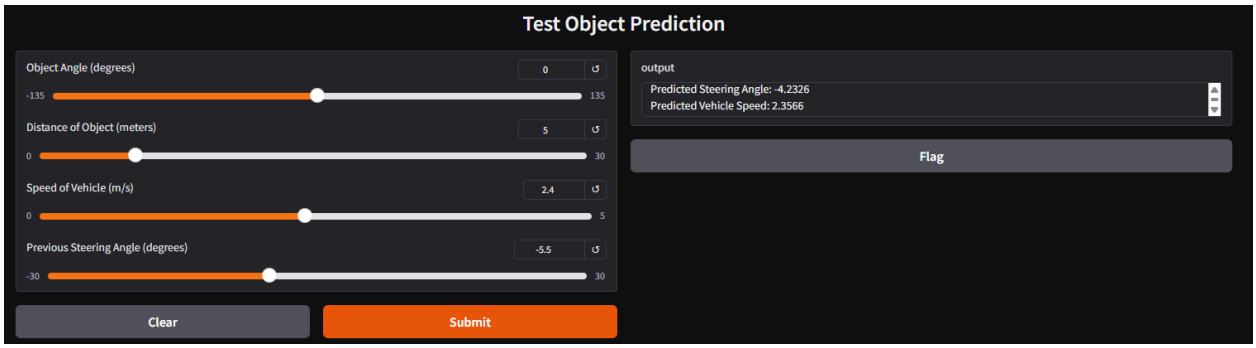


Gradio for testing the 54 ray outputs

By drawing on the sketch and then submitting the sketch, an image is then submitted, which need to be processed. First, the image will be converted from a RGB formatted array, to a binary array. Where a 0 would indicate white (#ffffff hexi) and 1 would be any other color. After this is done, the program then sets the center on the sketch to 0 so the square in the middle won't interrupt the processing algorithm. To convert the array into usable ray data, the program needs to use a

processing algorithm, which is inside the distanceAtAngle() function. This function takes the mask (or the canvas / sketch) as well as which theta needs to be calculated. Then the change in x and y are calculated by cos(theta) and sin(theta) respectively. Then going from the center the x and y are calculated, by going through a for loop that calculates the x = cx + i * dx. Which then will return the distance if mask[x, y] == 1.

Object-Based Prediction Demo Code:



Slider demo

The slider demo takes the input of Object Angle, Distance of Object, Speed of Vehicle, and Previous Steering Angle. The speed of vehicle and previous steering angle incorporate the needed information of how fast and what angle the vehicle turning angle, since the model heavily relies on the previous speed and angle to make a prediction. The object angle and distance need to use an algorithm to convert the inputs. This is because the features inside of the program use rays and radians as the header, which is not easy to visualize or interpret by most people. So in the demo, angle in degrees was chosen to be used instead.

To do this the findRay() function was created. Where it calculates the angle interval, which subtracts the minimum from the maximum angles and then divides by the ray total ((135-(-135))/1080 = 0.25 degrees / ray). Then computes the ray of the inputted angle by subtracting the minimum angle from the inputted angle and dividing it by the angle interval. Finally it is rounded to the nearest 20 and inputted into the model.