School of Engineering and Design
Professur für Autonomous Aerial Systems

# Dokumentation Group 15
## Autonomous Driving

**Rongxin Wang, Chen Yu, Zhenjiang Li, Jincheng Pan, Zhenyu Jin**

Lecture *Introduction to ROS* SS 2024

**Supervisor:** Prof. Dr.-Ing. Ryll, Markus

**Submission:** 19. Juni 2024

# Autonomous Driving

Rongxin Wang
Technische Universität München
Email: ge53bik@mytum.de

Chen Yu
Technische Universität München
Email: go96wig@mytum.de

Zhenjiang Li
Technische Universität München
Email: zhenjiang1.li@tum.de

Jincheng Pan
Technische Universität München
Email: ge53huy@mytum.de

Zhenyu Jin
Technische Universität München
Email: zhenyu.jin@tum.de

***Abstract*—The objective of this project is to achieve automatic driving of a vehicle on a specified route. The vehicle must complete the entire route in the shortest possible time while avoiding obstacles and complying with traffic rules. We have divided the project into three main components: perception, planning, and control. These components are interconnected through a state machine.**

**In the perception component, we employ two methods: a traditional computer vision (CV) approach and a learning-based approach. For the planning component, we utilize the way_point_global planner for global planning and the Teb Planner for local planning. Finally, in the control component, we use a PID controller for vehicle control.**

***Index Terms*—Computer vision, YOLOv5, Teb planner, PID controller**

## I. Perception

The perception system is designed to process sensory information into useful representations for navigation and scenario understanding. In this project, the primary tasks are developing a perception pipeline and understanding traffic scenarios.Zhenjiang Li and Zhenyu Jin worked on this part collaboratively.

Since the Unity environment provides precise pose information, localization is not a concern. The simulator uses a depth camera to capture depth images, which are then converted into a 3D point cloud using the depth_image_proc package. The processed point cloud is mapped using the octomap package, enabling effective navigation and route planning within the simulated urban environment.

In understanding traffic scenarios, two distinct methods are utilized: a traditional Computer Vision (CV) approach and a Deep Learning-based approach with YOLO.

*a)* ***Pointcloud****:* Zhenjiang integrated the depth_image_proc package into the perception pipeline. This package processes depth camera data into a meaningful point cloud, representing the environment in 3D coordinates. It subscribes to camera information from /DepthCamera/camera_info and converts depth data from /DepthCamera/image_raw into point cloud data, publishing it to the /DepthCamera/pointcloud topic. This point cloud is crucial for tasks like object detection, localization, and obstacle avoidance.

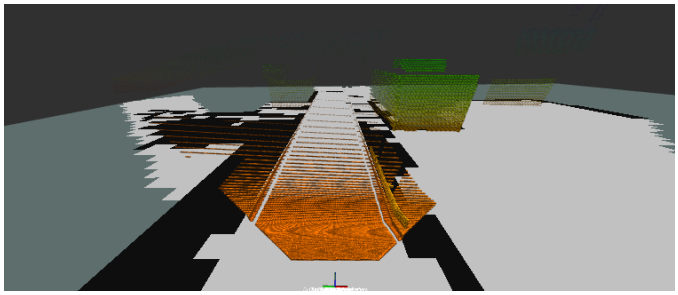


Figure 1: Point Cloud and occupency grid

*b)* ***Mapping****:* Zhenjiang also contributed to mapping using the OctoMap package. The RGB-D camera provides depth image data, which OctoMap uses to create a 3D map via occupancy grids. The point cloud data from depth_image_proc is processed to construct a 3D map, then projected into a 2D occupancy grid. Key parameters were adjusted to balance model accuracy and calculation speed, including setting the resolution to 1 and expanding the sensor model range to 25m. The final map, including all boundaries and obstacles, is shown in figures 2. Mapping is vital for navigation, allowing the robot to plan routes, avoid obstacles, and identify safe areas.

*c)* ***Detection and classification****:* The main task in traffic scenario understanding is detecting traffic lights. Based on sensor data from the simulation and experiment requirements, two solutions were proposed: a traditional CV-based method and a deep learning-based method using YOLO.

**1. Sementic-Camera-based solution:** In this project, two key source files, `detector.cc` and `trafficlights_detect_node.cpp`, are used to implement the traffic light detection functionality. The following are the implementation details:

**1) Traffic Light Localization:** This part contributed by Zhenyu Jin and Zhenjiang Li, utilizes semantic image data from /SemanticCamera/camera_info identifies regions of interest (ROI) where traffic lights are likely. Specific visual patterns and colors (RGB: [255, 234, 4] for traffic lights) are analyzed to extract relevant pixels, focusing on the middle screen to avoid multiple lights interference.

**2) Color Recognition:** After identifying the ROI, Zhenyu

Figure 2: Sementic camera-based detector

and Zhenjiang has used a color-feature scan of the RGB image data from /RGBCameraRight/image_raw determines the traffic light's color. This method reduces the number of iterations needed for processing. Through debugging, they analyzed the state of the traffic lights within the ROI by adjusting the RGB thresholds for red and green lights. The optimal thresholds identified were: Results are published as

| Color | Red Threshold | Green Threshold | Blue Threshold | Traffic State |
|---|---|---|---|---|
| Red | > 180 | < 80 | < 80 | True |
| Green | < 150 | > 180 | < 150 | False |

Table I: Traffic Light Color Recognition Thresholds

Trafficstate.msg to /perception/traffic_state and as bounding boxes to /perception/boundingbox_image. This ensures precise navigation and adherence to traffic regulations.

**3) Box Drawing:** The bounding box drawing functionality in the Perception Detector Node, designed by Zhenyu Jin, visually marks detected traffic lights on RGB images. It starts by calculating the bounding box coordinates for the identified traffic lights, determining the minimum and maximum x and y values. The core idea of this part is to update the bounding box coordinates. If the new point's x or y value is smaller or larger than the current minimum or maximum values, respectively, the values are updated accordingly. These coordinates are then used to draw the bounding box on the RGB image. The color of the box indicates the traffic light state: green for "go" and red for "stop." This visual marking enhances traffic signal recognition and localization, aiding safe navigation. The modified images with bounding boxes are published for further processing in the autonomous driving system.

**4) Image publishing:** The modified images with bounding boxes are then published to the ROS topic boundingbox/image for visualization purposes. This step ensures that the processed images, now containing visually marked areas of interest, can be easily viewed and analyzed. By publishing these images, developers and operators can visually verify the detection and localization of traffic lights, aiding in debugging and system validation. This visualization facilitates better understanding and monitoring of the autonomous driving system's perception capabilities.

**2. Learning-based solution:**

In this project, the key source file, detector.py , are used to implement the traffic light detection functionality. Zhenjiang Li is responsible for this part. This method detects traffic lights directly from camera images without using semantic data. Deep learning models, particularly one-stage detectors YOLO, are suitable for real-time object detection in autonomous driving.

**1)Why YOLO?** Naive ideawhy don't we using Transfer Learning approach involving an Encoder-Decoder network, such as one based on AlexNet(figure 3)? In practical, YOLO over traditional NN architectures offers several significant advantages:

- **Real-time Performance**: YOLO is designed for high efficiency, performing detection in a single forward pass through the network, making it highly suitable for real-time applications. This is critical for autonomous driving, where timely responses are essential.
- **Resource Efficiency**: The single-network architecture of YOLO demands less computational power compared to the more resource-intensive Encoder-Decoder networks. This makes YOLO ideal for deployment on resource-constrained platforms, such as small autonomous vehicles. And the model size generated from yolo for detection is 35 mb smaller than one from a Encoder-Decoder networks.
- **Ease of Integration**: YOLO is widely adopted, with robust frameworks and extensive documentation available, facilitating easy integration, rapid prototyping, and iterative development.
- **Accuracy Improvement**: Initial tests using AlexNet for Transfer Learning resulted in lower accuracy for traffic light detection(see figure 4). The complex structure and lack of real-time processing capability made it less suitable for our autonomous driving application. Despite fine-tuning, the accuracy remained below expectations, highlighting the need for a more robust and efficient detection method.
- **Visualization and Detection Box Generation**: YOLO's architecture directly predicts bounding boxes and class probabilities, making it straightforward to visualize and interpret the state of traffic lights.
- **Pretrained Models**: There are numerous pretrained models based on YOLO that can be readily used. These models, trained on large datasets, provide a solid starting point and can be fine-tuned to specific tasks, significantly speeding up the development process.

In conclusion, YOLO (You Only Look Once) is a mature and widely used technique in practical applications, particularly suited for real-time object detection tasks.

**2) Dataset Collection and Labeling:** Images with traffic lights are collected and annotated to create a comprehensive dataset. The collection process involves capturing photos of traffic lights from various angles using a right-side camera. This approach ensures a diverse set of images, enhancing the model's ability to generalize across different scenarios.

The collected images are then annotated to identify the regions of interest, specifically the traffic lights. The annotation process is facilitated by using the LabelImg tool, which allows for precise and accurate labeling. Each image is annotated to

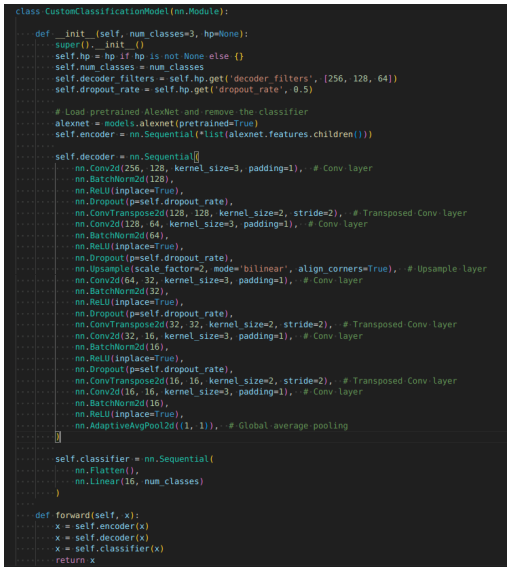



Figure 3: AlexNet Transfer Learning

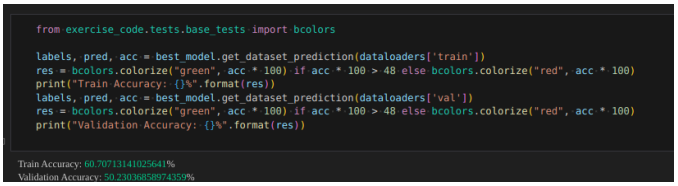



Figure 4: Acurracy of encoder and decoder structure

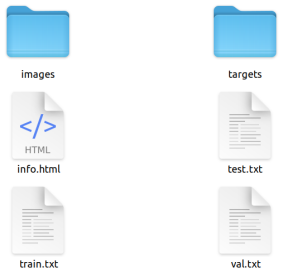



Figure 5: Dateset collection and splitting

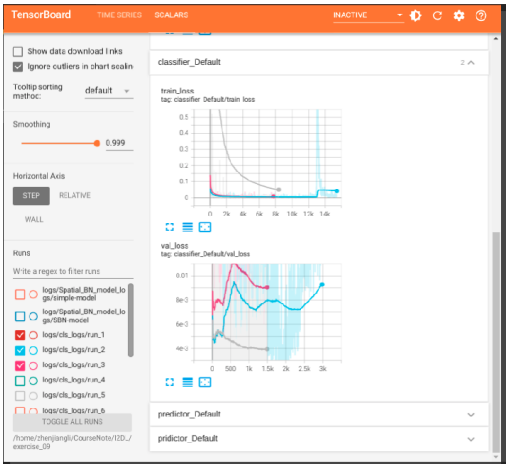



Figure 6: Tensor board

mark the bounding boxes around the traffic lights, indicating their positions and states (red, green, or other).

The annotation details for each image are saved in a corresponding '.txt' file, following the YOLO format. Each line in the file represents one bounding box and contains the following information:

- `class_id` - The identifier for the traffic light's state (e.g., 0 for red light, 1 for green light).
- `center_x` - The x-coordinate of the bounding box center, normalized by the image width.
- `center_y` - The y-coordinate of the bounding box center, normalized by the image height.
- `width` - The width of the bounding box, normalized by the image width.
- `height` - The height of the bounding box, normalized by the image height.

An example of the annotation format for a red light and a green light in the same image might look like this:

```
0 0.5 0.5 0.1 0.1
1 0.1 0.1 0.05 0.05
```

The annotated dataset is then augmented to further improve model generalization. Augmentation techniques such as rotation, scaling, and flipping are applied to create a more varied dataset. This helps in training a robust model capable of detecting traffic lights under different conditions.

**2) Model Training and Evaluation:** YOLO offers pre-trained models of various sizes. YOLOv5s was chosen for its balance between speed and accuracy. Training was conducted using the annotated dataset. The training process was carried out using the official YOLOv5 Jupyter notebook provided by Ultralytics [1].

During training, TensorBoard was utilized to assist in monitoring the training process, allowing for visualization of metrics and model performance over time.(Figure 2)

**Detector Node:** The YOLOv5 ROS Node for Traffic Light Detection leverages a pre-trained YOLOv5 model to identify traffic lights in images, publishing the bounding boxes and traffic light states. The node subscribes to an input image topic and publishes the results to specific output topics, enhancing autonomous driving capabilities through real-time traffic light recognition. Key functionalities include:

- **Initialization:** The node initializes by loading parameters from the ROS parameter server, setting up the YOLO model with specified weights, and configuring inference settings (e.g., confidence threshold, IoU threshold, image size).
- **Image Processing:** The node subscribes to the input image topic, converting ROS image messages to OpenCV format for processing. Images are preprocessed (resized, normalized) and converted to tensors for model inference.
- **Inference and Detection:** The YOLO model performs


Figure 7: Classification Result

inference on the input images, detecting traffic lights. Non-max suppression is applied to filter out redundant bounding boxes. Detected bounding boxes are then scaled to match the original image dimensions.

- **Color Recognition:** Detected traffic lights are classified as red, green, or yellow based on the predefined color thresholds. The classification results are published to the `/perception/traffic_state` topic, and bounding boxes are published to the `/perception/boundingbox_image` topic.
- **Visualization:** For visualization, the node publishes images with annotated bounding boxes to an output topic. An optional parameter allows the display of these images during runtime. The different light states are tagged like in figure 5.

## II. Planning

In this part, we chose `move_base` as the primary package. Jincheng Pan and Ronxin Wang worked on this part collaboratively. Jincheng was responsible for the `move_base` configuration and defining a separate `service client:Waypoint.srv` to send waypoints, while Ronxin handled the later adjustments of `move_base` configuration parameters.

We have inputs from the perception system generating the occupancy grid map, traffic light states, and predefined global paths. The goal is to generate global paths, local trajectories, and corresponding velocity commands. Therefore, there are three main challenges: 1. How to generate appropriate waypoints and send them to the vehicle for trajectory planning and optimization; 2. How to perform global planning; 3. How to perform local planning.

### A. Waypoints Generation

First, we need to obtain a series of waypoints for guidance. These waypoints should be on the predefined path, but since there is no global map for guidance, we need to fit the target path as shown in the figure below. Jincheng's strategy is to initially provide rough waypoints, then set a distance threshold of 20m (this value should be the same as the threshold for global planning later). When the distance between two adjacent points is greater than the threshold, linear interpolation is applied. If the distance is less than the threshold, the point is classified as a turning point. Finally, these groups of turning points are uniformly fitted using spline interpolation, and the results are visualized.

So, Jincheng wrote a `waypoint_generator.py` script, generating results as shown in the figures below. By comparing the two figures, a good fitting effect can be seen. Then, we save all the obtained points and select a subset of them as the actual waypoints for subsequent use. Additionally, this part facilitates later optimization of waypoints based on the performance of the vehicle and the curvature size.

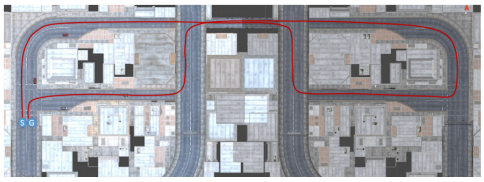


Figure 8: The Goal Path

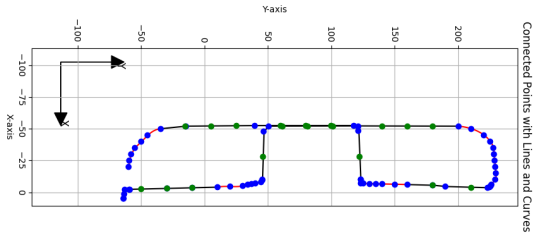



Figure 9: The Fitted Curve

Next, Jincheng used a service-client pair and defined a `Waypoint` Message (containing pose information) to send waypoints to the `move_base`'s global planner.

During the initial debugging process, we found that sending all waypoints at once would cause the global planner to fail because these points were unordered and the start and end points were very close, causing the planner to tend to connect them directly. Therefore, in the node named `global_path_planner`, Jincheng grouped the waypoints, with each traffic light as the endpoint of the current stage. The next group of waypoints is sent to `move_base` for planning only when the traffic light is not red.

To better simulate the braking behavior before the traffic light, we employed a trick by inserting an additional point in front of the endpoint of each stage (located ahead of the endpoint), as shown in the figure below. This way, the vehicle simulates a braking effect and can stop at a red light. On the other hand, if the light is green, the next stage planning is immediately performed, so it is not affected.

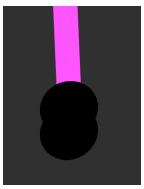


Figure 10: The additional point inserted to simulate braking behavior.

### B. Global Planning

Although `move_base` provides several global planners (including A*, Dijkstra, etc.), due to the precondition of our task that we do not have complete map information, we cannot directly generate the corresponding global cost map. Therefore, we can only use the local cost map generated by the vehicle during operation as the global cost map. However, this presents a problem: the scope of our global planning will be limited to the range of the local cost map. In other words, the default global trajectory planning provided by `move_base` will no longer be applicable.

To solve this problem, Jincheng found a more suitable global path planner on GitHub, called `waypoint_global_planner`. The implementation idea of this global path planner is to linearly connect all the waypoints sent to it and then perform linear interpolation according to the preset interval value (20m). The generated global path is then sent to our local trajectory planner in the form of a `nav_msgs/Path` message. The specific implementation effect is shown in the figure below.

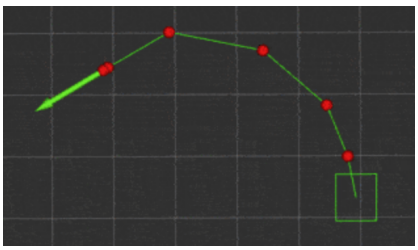


Figure 11: Implementation effect of the waypoint_global_planner

The implementation of this global path planner is very similar to our initial waypoints generation approach, making it highly compatible. On the other hand, this plugin allows the generation of global paths without being limited by the current global cost map range, significantly speeding up the global path planning process as it only needs to linearly connect and interpolate all the received waypoints. However, there is also a downside: it is highly dependent on the quality of the initial waypoints, making the generation of waypoints particularly important.

### C. Local Planning

The main task of local planning is to accept the `nav_msgs/Path` generated by the global planner and generate a locally optimal trajectory in conjunction with the local costmap. It also provides the corresponding speed commands to effectively avoid dynamic obstacles and track the global path.

The commonly used local planners in `move_base` are the DWA Planner and the Teb Planner [2]. Here, we choose the Teb Planner because our mission goal is to complete the task in the shortest possible time, and the Teb Planner is more suitable for such racing scenarios. It uses the Time Elastic Band algorithm, which treats the given path as an elastic rubber band influenced by internal and external forces, causing it to deform. These forces balance each other, allowing the path to shrink while maintaining a certain distance from obstacles. The internal and external forces represent all constraints on the robot's movement. Specifically, the algorithm inserts N control points (robot poses) that control the shape of the rubber band along the given path. It defines a motion time between these points, achieving multi-objective optimization, which includes minimizing trajectory execution time, maintaining a safe distance from obstacles, and adhering to kinematic constraints.

Although `move_base` provides a configuration method for the Teb Planner, it has a large number of parameters that need to be adjusted (such as `min_obstacle_dist` and `inflation_dist`). Additionally, most constraints in the Time Elastic Band algorithm are soft constraints. Rongxin found during debugging that if the parameters and weights are not set properly or the environment is too harsh, the Time Elastic Band algorithm may fail, resulting in very strange trajectories. Therefore, it is also necessary to consider the actual contour of the vehicle. We approximate it using a rectangle as the `footprint_model`.

Below is a demonstration of our implementation. The pink line represents the global path, the green line is the current local tracking trajectory generated by the Teb Planner, and the rectangle represents the assumed vehicle dimensions. The surrounding area shows the generated costmap and the corresponding inflation layer. As can be seen, when there is a deviation in the path or a significant obstacle appears, Teb effectively corrects the trajectory.

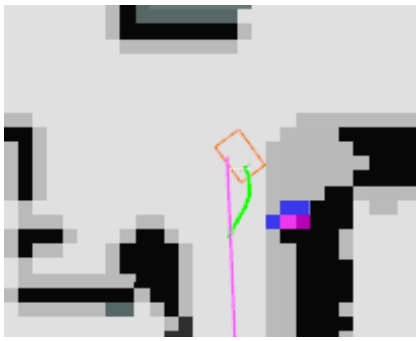
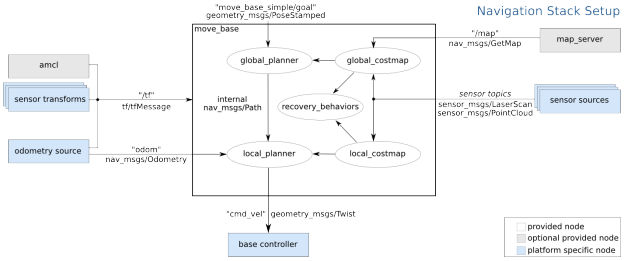


Figure 12: Implementation effect of the Teb_local_planner

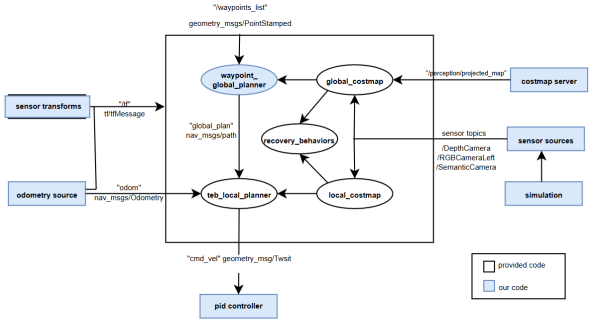
Overall, for the Planning section, we based our configuration on the `move_base` package, incorporating an open-source global planner plugin and using the built-in Teb local planner. We performed segmented global path planning, successfully completing this part of the task. The diagrams below show the official `move_base` configuration and our final implemented configuration.

## III. Control

In this part, we use the PID Controller as our control algorithm. Jincheng Pan is responsible for the implementation




(a) Official move_base Configuration Diagram




(b) Our move_base Configuration Diagram

Figure 13: Comparison of move_base Configurations

of the PID Controller, while Rongxin Wang is responsible for the subsequent parameter tuning.

The goal of this task is to generate corresponding control speed based on the target speed generated by `move_base`, and input the simulated speed into the simulation environment to simulate the vehicle's speed. There are three main parts to complete: 1. Conversion of speed commands. 2. PID controller. 3. Tuning of PID parameters. 4. State machine.

### A. Conversion of speed commands

Initially, we used the output of `move_base` as the reference speed for our PID controller, but after testing, we found that this approach was not effective. The input from `move_base` is the 'cmd_vel' message of type 'geometry_msgs/Twist', which mainly includes the linear velocity 'linear.x' and angular velocity 'angular.z'. However, our desired final output speed for the simulation should include acceleration, turning angle, and braking, which are not directly provided by the 'cmd_vel' message. Therefore, we need to convert the speed commands.

Jincheng adopted the following strategies: 1. Calculate the acceleration by computing the change in speed between the current and the previous time steps, and include this information in the simulation. 2. Convert the angular velocity to the turning angle using the following formula:

$$\tan(\delta) = \frac{l \cdot \omega}{v} \tag{1}$$

where $l$ is the wheelbase, $\omega$ is the angular velocity, and $v$ is the linear velocity.

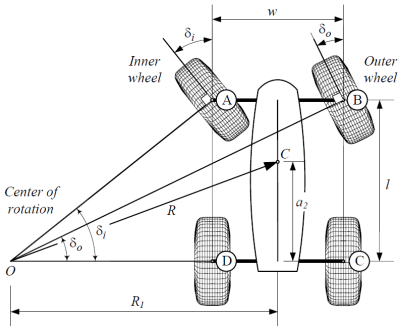



Figure 14: Ackermann steering geometry

This formula is actually a simplified version of the traditional Ackermann steering formula. The traditional Ackermann steering formula is:

$$\tan(\delta) = \frac{l}{R + \frac{w}{2}} \tag{2}$$

where $\delta$ is the front wheel steering angle, $l$ is the wheelbase, $R$ is the turning radius, and $w$ is the track width.

1) **Turning Radius**: As we all known, the turning radius $R$ can be expressed as:

$$R = \frac{v}{\omega} \tag{3}$$

2) **Substituting the Turning Radius**: By substituting this turning radius $R$ into the traditional Ackermann steering formula, we get:

$$\tan(\delta) = \frac{l}{\frac{v}{\omega} + \frac{w}{2}} \tag{4}$$

3) **Simplification**: To simplify this formula, assuming that the distance between the front wheels is negligible during turning, i.e., $w \approx 0$:

$$\tan(\delta) \approx \frac{l \cdot \omega}{v} \tag{5}$$

3. Conversion of braking speed is not needed because for our task, the main braking scenario is at intersections with traffic lights. Since we have already added waypoints to simulate braking in the Planning section, braking can be quickly achieved.

Through the above conversion relationship, we can achieve the conversion of speed commands. To facilitate expression, we use ROS's `ackermann_msgs/AckermannDriveStamped` to represent the converted speed commands.

### B. PID controller

Since our task does not provide the vehicle's dynamic model, we cannot use more advanced control algorithms (such as MPC, ILQC, etc.) and have to choose the simplest yet effective PID Controller.

The PID controller calculates the error between the desired and actual acceleration and steering angle of the vehicle and

applies corrective actions to minimize this error. The control law is given by:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{6}$$

where $e(t)$ is the path error at time $t$, and $K_p, K_i$, and $K_d$ are the proportional, integral, and derivative gains, respectively.

Therefore, we only need to follow a certain frequency, first converting the speed commands to obtain the target speed as the setpoint. Then, we compare it with the actual speed read from the "odom" (the output) to get the error. By performing the following calculation, we can obtain the speed that needs to be output to the simulation environment.

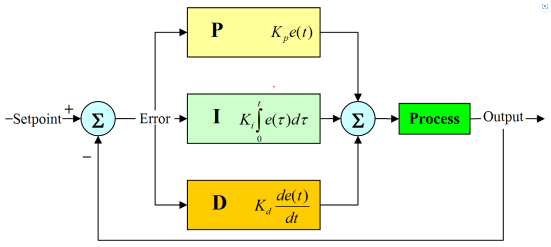



Figure 15: PID Controller

### C. Tuning of PID parameters

The PID controller calculates the error between the desired and actual positions of the vehicle and applies corrective actions to minimize this error. The control law is given by:

Tuning the PID parameters is indeed one of the difficulties in this part. To facilitate tuning, we record the PID error calculated at each moment during the operation process and visualize it using a Python script to assist in parameter tuning, as shown in the figure below.

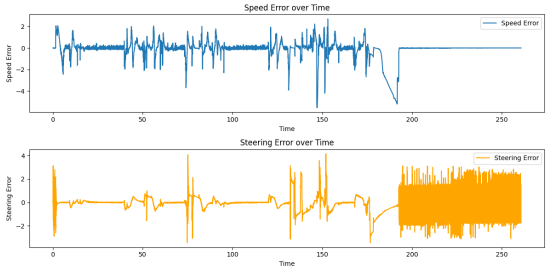



Figure 16: PID Error

Rongxin configured the move_base settings and tuned the PID parameters for the autonomous driving system. The objective was to accurately compute and plan the vehicle's position in real-time using data from various sensors, including an IMU. Ensuring the vehicle consistently followed the desired path was a significant challenge, necessitating meticulous optimization of both move_base parameters and the PID values.

During the tuning process, an intriguing phenomenon was observed: increasing the values of the Integral (I) or Derivative (D) terms led to a loss of control over the vehicle. Additionally, the same program with identical parameters performed differently across various hardware configurations. For example, a high-performance desktop computer running Ubuntu managed the program without collisions or loss of control, while a laptop with a virtual machine frequently experienced control issues and collisions.

To investigate these discrepancies, extensive trials were conducted and the underlying issue was pinpointed. The motion signal publishing frequency was set to 10 Hz, requiring the computer to perform multiple tasks—such as data reception, transmission, and processing—within 0.1 seconds. Higher values of I and D introduced complex integral and differential calculations that the computer could not complete within the allocated time, resulting in control loss. Additionally, the laptop running a virtual machine had slower processing capabilities, which affected the timeliness of control signals and overall vehicle management.

Ultimately, it was determined that setting P to 1.70, I to 0.01, and D to 0.01 provided effective results. Minimizing the I and D values reduces computational complexity, allowing the system to maintain accuracy and stability while minimizing oscillations. With these PID settings, the program achieves relatively precise control across different hardware configurations, though performance still has an impact on overall effectiveness.

### D. State machine

We use the state machine to complete the transition between the driving state and the stopped state. Chen completed this part of the work and integrated the various documents together.During the driving phase, the target speed and angular velocity are published, and both are set to 0 during the stopped phase. In addition, the start time is read when entering the driving state for the first time, and the end time is read when reaching to the end point to obtain the overall time. The basis for judging whether to reach the end point is the distance from the end point.

## IV. Summary

In this project, we first used traditional computer vision-based methods to complete the localization and color recognition of traffic lights, and then used a learning-based method to use YOLOv5 to complete perception without relying on semantic cameras. In the planning part, we generated multiple waypoints and performed global planning, then used Teb planner for local planning, and used a PID controller to control the movement of the car. Finally, we tuned the waypoint and PID parameters, and connected the various parts with a state machine to complete the transition between the driving state and the stopped state.

## References

[1] Ultralytics, "Yolov5 jupyter notebook integration." https://docs.ultralytics.com/integrations/jupyterlab/#step-4-start-experimenting, 2023. Accessed: 2024-07-29.

[2] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, and T. Bertram, "Trajectory modification considering dynamic constraints of autonomous robots," in *Proc. 7th German Conference on Robotics*, pp. 74–79, Citeseer, 2012.