I2ROS G13: Autonomous Driving

Chenyu Meng chenyu.meng@tum.de
Zefeng Wang zefeng.wang@tum.de
Qiyang Zong qiyang.zong@tum.de
Yulong Xiao yulong.xiao@tum.de
Yichao Gao yichao.gao@tum.de

***Abstract*—The objective of this project is to develop an autonomous driving system capable of navigating an urban track as fast as possible while adhering to traffic rules and safety regulations. The system should avoid collisions with other cars, stay within the designated road boundaries, and correctly respond to traffic lights. In this work, we design two routes to achieve the goals. In the Route 1, the goal is accomplished with a relatively simple method. The method is optimized based on the specific task, and it performs well with minimum resources. It is simple but efficient. In the Route 2, a more general approach is explored, which can be used in scenarios that go beyond the current application scenario, e.g., with dynamic traffic and greater noise. In the end, we show the results of the 2 routes and the insights from the exploration.**

I. Perception

Perception processes sensory information into meaningful representations. In this project, the main task of perception is building a perception pipeline and traffic scenario understanding.

Since the Unity environment provides the ideal pose information for the car, there is no cause for concern regarding localization. In our project, the simulator provides a depth camera that captures depth image data. We utilize the `depth_image_proc` package to convert this data into a 3D point cloud representation. Subsequently, we create the map using the `octomap` package, leveraging the processed point cloud. This mapping process allows the autonomous vehicle to navigate and plan routes effectively in the simulated urban environment.

In the traffic scenario understanding part, two distinct methods are employed to achieve our goal. The first method is a **traditional Computer Vision (CV)** solution, while the second is a **Deep Learning-based** approach using **YOLOv5**.

A. Pointcloud

Yichao has integrated the `depth_image_proc` package [1] into the perception pipeline.

This package is responsible for processing depth camera data and transforming it into a meaningful point cloud representation. The package generates a point cloud representation of the scene. Each point in the point cloud corresponds to a three-dimensional coordinate in real-world space, providing an accurate spatial mapping of the environment. Then it subscribes `camera_info` from `~/DepthCamera/camera_info` and from depth image data in `~/DepthCamera/image_raw` converts depth data into point cloud data, and publishes it to the `~/DepthCamera/pointcloud` topic. The resulting point cloud serves as a fundamental input for subsequent perception tasks, such as object detection, localization, and obstacle avoidance. Figure 1 shows the visualized point cloud data in Rviz.

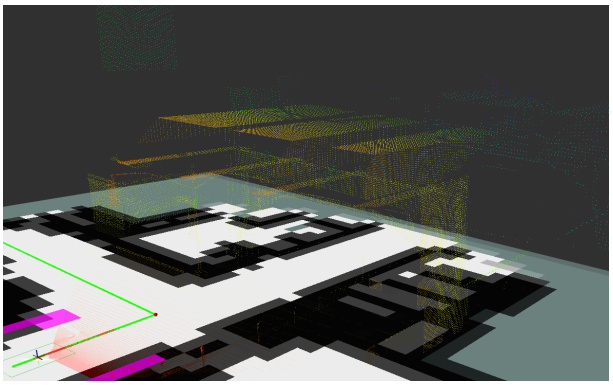

Fig. 1: Pointcloud

B. Mapping

Yichao has made contributions to mapping using the package `OctoMap` [2].

Since the depth image data is provided by the RGB-D camera, OctoMap algorithm could be applied. It is a 3D mapping framework that uses occupancy grids to represent the environment. It generates a map by acquiring the point cloud data, which is generated by package `/depth_image_proc` from the depth image data and the camera information, and constructing a 3D point cloud map. A 2D occupancy grid map will then be generated by the projection of the 3D point cloud map.

Some parameters were adjusted: `resolution` was set to 1 for a balance between model accuracy and calculation speed; `sensor_model/max_range` was expanded to 25m for a faster mapping process; `occupancy_min_z` and

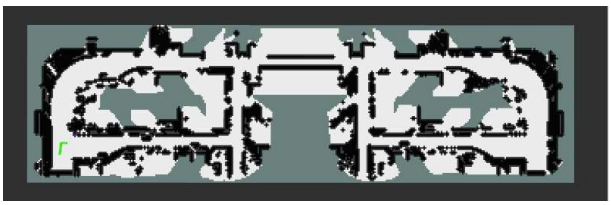

Fig. 2: Map

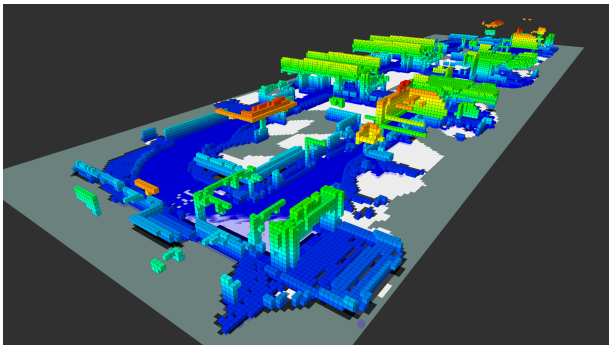

Fig. 3: Occupancy grids

`occupancy_max_z` were set to -1m and 1m in order to generate a map including all boundaries and obstacles.

The final mapping result is shown in figure 2 and 3.

Mapping is an essential component of robotics, as it provides a robot with the necessary information about its environment to perform a task effectively and safely. Especially in navigation, mapping helps robots to navigate through their environment. A map allows a robot to plan a route from its current location to a target location, avoid obstacles, and identify safe areas to move through.

C. Traffic Scenario Understanding

To understand the current traffic scenario and follow traffic rules, the main task is to complete the detection of the traffic light. Based on the sensor information provided by the simulation and the requirement of the experiment, we proposed 2 solutions. The first solution is a traditional CV-based method using camera and semantic camera information. The second solution is a deep-learning-based method using only camera information.

1) Traditional CV-based solution: Yichao's role involves working on the traditional CV-based detection to identify red lights using semantic image data. To achieve this goal, Yichao implemented the `perception_pkg/trafficlights_detect_node`.

The detection process is divided into two parts: traffic light **localization** and **recognition**.

a) Traffic Light Localization: To locate the traffic lights, semantic image data from topic `~/SemanticCamera/camera_info` plays an important role to identify regions to interest(ROI) where traffic lights area likely to be present. By analyzing specific visual patterns and colors(RGB: [255,234,4]for traffic lights) in the semantic data, the pixels which are placed on the traffic lights area are extracted. To avoid interference from multiple traffic lights, we choose to extract only the pixels in the middle part of the screen as the Region of Interest (ROI).

b) Traffic Light's color Recognition: After identifying the ROI of traffic lights, a color-feature scan of the RGB image data from `~/RGBCameraLeft/image_raw` at the ROI was generated to recognize the color of the traffic light. By analyzing the color patterns present in this loop scan, we accurately determine the current state of the traffic light, whether it is red or green. Scanning only the ROI part of RGB image data, the number of iterations required for processing pixels is significantly reduced.

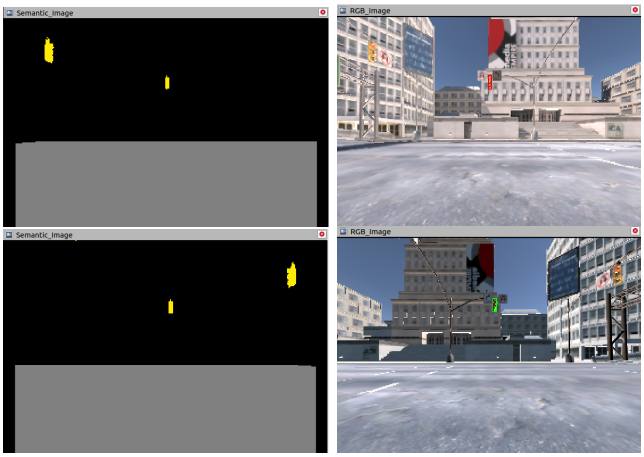

Fig. 4: Traffic light detection based on CV solution

The combination of the semantic image data-based approach and the scan for state determination enables the vehicle to make precise decisions in response to traffic lights. The result will be published as the Boole message type `Trafficstate.msg` to the topic `/perception/traffic_state`. Then a bounding box covering the traffic light is drawn on the RGB image, which is published to the topic `/perception/boundingbox_image`. Figure 4 shows the result.

With high accuracy in detecting and recognizing traffic lights, our perception system ensures the autonomous vehicle can navigate safely and efficiently through intersections while obeying traffic regulations.

2) Deep-learning-based solution: The previous subsection applied a traditional CV-based method to achieve traffic light detection with the help of the provided semantic camera. In this subsection, a deep learning-based method is introduced to successfully detect the traffic light information directly from the image captured by the camera without using the provided semantic camera.

In recent years, deep learning has made significant strides in image processing, becoming a core technology in the field of computer vision. It leverages deep neural network models to extract high-level features from images, enabling various tasks such as image classification, object detection, segmentation, and generation.

Currently, there are many deep-learning models available for object detection, which can be categorized into two main types: one-stage object detection and two-stage object detection. In the field of autonomous driving, real-time object detection must be considered. Therefore, we utilize the famous one-stage detection YOLOv5 (You Only Look Once) as our model [3].

The structure of YOLOv5 is shown in figure 5.

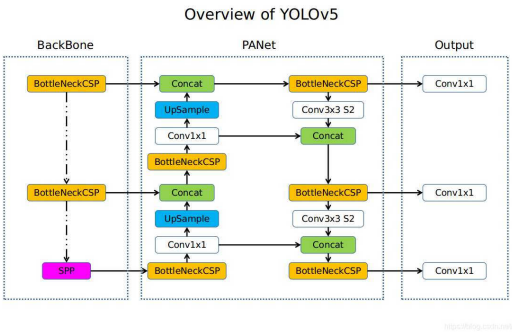

Fig. 5: YOLOv5 Structure

To perform traffic light detection using YOLOv5, the subsequent steps have to be accomplished:

a) Dataset Collection and Annotation: Collect and annotate a dataset of images containing traffic lights from the unity simulation environment and label each image with bounding boxes around the traffic lights. In this step, the topic `~/RGBCameraLeft/image_raw` information was recorded as a video when the car was driving at each intersection with traffic lights. The original dataset consists of 424 source images with an image extraction speed of five frames per second. The dataset is shown in figure 6. The dataset is split into a training set, validation set, and test set according to the ratio of 70%, 20%, and 10%. To improve the model's generalization ability, robustness, and avoid overfitting, data augmentation was applied to increase the size of the training dataset. Three distinct methods are used for data augmentation, horizontal flip, exposure, and noise. Ultimately, the size of the training set is expanded to 891 images. The dataset annotation is achieved with the help of the tool Roboflow. The annotated images and labels were exported into a YOLO format supported by YOLOv5. YOLOv5 uses a YAML format dataset configuration file containing the class information, with each image corresponding to a txt file containing the label information for the traffic lights.

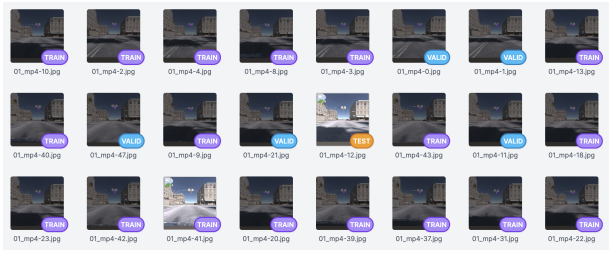

Fig. 6: Traffic light dataset

b) Model Training and evaluation: YOLOv5 offers pretrained models of different sizes, denoted by s, m, l, x, representing small, medium, large, and extra-large models. Smaller models are faster but less accurate, while larger ones are more accurate but slower. According to the experiment requirements, YOLOv5s model is more suitable. After the determination of the model, the training process was conducted using the training dataset. After training, the model was evaluated using the validation set. Mean average precision(mAP), precision, and recall were applied to assess the model's performance. In the end, the model achieved a mAP of 98.5%, precision of 97.7%, and recall of 98.6%.

c) Traffic Light Detection and Result Publication: We employed the trained model to perform traffic light detection. The results are depicted in figure 7. The detection of all traffic lights was successful, with an exceptionally high confidence level. The inference function was implemented in the script `detect.py` and the corresponding ROS node successfully published three output topics. The bounding boxes of the detected traffic lights were disseminated as the message of type`BoundingBoxes.msg` via the topic `/perception/detections`. Additionally, the traffic state was conveyed as the message of type `Trafficstate.msg` through the topic `/perception/traffic_state`. The traffic state is set to True if the traffic light is red, otherwise, it is set to False. Furthermore, another topic `/perception/boundingbox_image`, which contains the annotated image as the message of type `Image.msg` is also published to facilitate visualization.

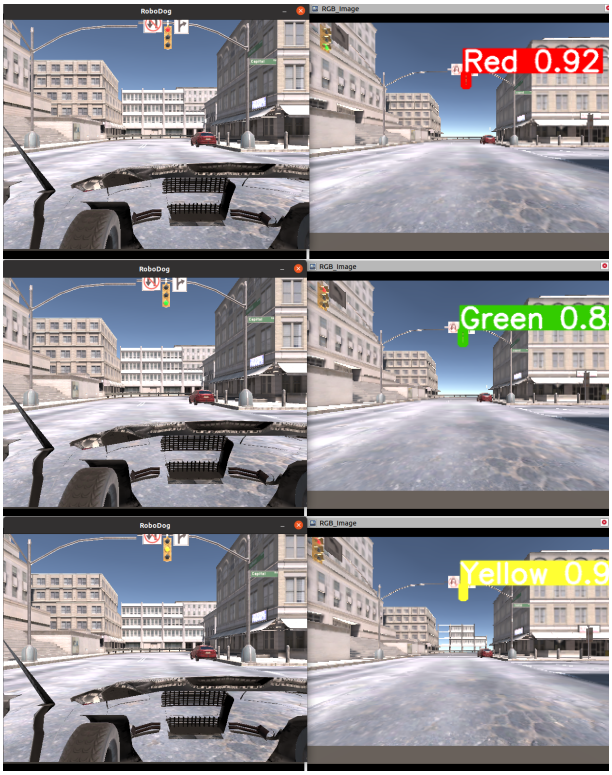

Fig. 7: Traffic light detection using YOLOv5

II. Planning

For the planning in this project, we chose `move_base` as the primary package. Yulong Xiao and Qiyang Zong worked on this part collaboratively. For more details, Yulong straightened out the chaotic coordinate relationship and

achieved the successful publication of the odom topic. He also added and configured the TebLocalPlanner. Meanwhile, Qiyang handled the goal points publication for the global planning by using a service-client pair and configured the parameters in move_base. There are two distinct designs for the global planning and local planning. The first design involves utilizing the built-in global_planner and local_planner, which employs sophisticated searching algorithms to identify a feasible path towards the goal. On the other hand, the second design entails using the waypoint-global-planner, which generates a path by establishing a direct straight line between two waypoints and teb_local_planner, which provides better support for car-like robot. We use waypoint-global-planner and teb_local_planner in **Route 1** and use build-in global_planner and base_local_planner in **Route 2**.

A. Global Planning

The global_planner package in the move_base is responsible for generating a feasible path for a robot to reach a destination in a given environment. As shown in Figure 8, it considers the global_costmap and the goal published in topic move_base_simple/goal to plan the path from the robot's current position to the goal. The generated path is of type nav_msgs/Path and will be utilized by local_planner. The global_planner package provides different global planning algorithms that can be used to generate paths, such as Dijkstra's algorithm and A* algorithm. These algorithms take into account the robot's start position, the goal position, and the global map to explore and find an optimal path.

Here we use a service-client pair to achieve the publication of a series of Goal Points. To obtain these Goal Points, we utilize the 2DNavGoal service to set a single point on the map and then extract the relevant information from the /move_base_simple/goal topic, which includes the pose and quaternion data. Once the vehicle receives a Goal Point, the global planner generates a corresponding global path. It is important to note that the map range is limited, meaning that setting a Goal Point beyond this range would result in an error. In our approach, we set a total of 43 Goal Points and organize them into a 2-dimensional vector. To ensure smooth navigation, we implement a while-loop that continuously assesses the tolerance between the true pose of the car and the goal pose. When this tolerance becomes smaller than a predefined value, the client node proceeds to send the next Goal Point to the server, which, in turn, publishes it to the /move_base/goal topic. This process continues until all the Goal Points have been successfully sent, at which point the while-loop terminates.As is shown in Figure 9.

Inside the server_client pair, a new message type planning::PlanGoal::Request was defined by Qiyang. In the judge node, the conditional judgment and the acquisition of the next target point are realized. The extracted data is stored on the client side of the custom message. In the sender node, the customer-side data stored in the custom message is read and assigned to the server-side data. In the calling function of the sender node, the data of the server is sent to the corresponding topic. This logic is used in **Route 2**.

However, in our simulation environment, which is deterministic and devoid of dynamic obstacles, we have another option to utilize the waypoint-global-planner [5] as a plugin to the global_planner. Unlike the default search algorithms employed in dynamic environments, the waypoint-global-planner adheres to a simpler and more direct approach by drawing straight lines between two predefined waypoints. The deterministic nature of our environment and predefined global path implies that obstacles and paths remain constant. Thus, the waypoint-global-planner can efficiently guide the robot towards its destination by adopting a more direct and predictable approach. There is no need for sophisticated exploration, as the environment's layout is already known and the global path is already predefined. The simplicity of waypoint-global-planner also adds to the overall robustness of the navigation system, reducing the likelihood of path-planning errors or failures. When using waypoint-global-planner, the server-client pair is further modified to publish a series of waypoints to generate a global path. One example is illustrated in Figure 10. In total, we have 5 paths separated by traffic lights. When the distance between the car and the traffic light is less than 2m, the traffic state is obtained by subscribing \perception\traffic_state topic, which is published by trafficlights_detect_node. Only when the traffic light is green, the next path to the next traffic light will be published. Otherwise, the car will stop and wait for the green signal. This waypoint-global-planner and modified server-client pair are used in **Route 1**.

B. Local Planning

For the local planning, we implemented besides build-in planner another planner, which is more suitable for the car-shape robot, named teb_local_planner. The build-in base_lacal_planner is used in **Route 2** and the teb_local_planner is used in **Route 1**.

A local planner will read the global path and odom message, so that the local planner can finally according to the local cost map generate a local trajectory and a geometry_msgs/Twist message containing translational and angular velocity v and $omega$ respectively for commanding the robot.

The odom (short for odometry) used by the local planner is an essential piece of information. The odom topic publishes data in the form of a nav_msgs/Odometry message, containing header, child_frame_id, pose and twist. pose contains information about the robot's position and orientation in the global coordinate frame given by header.frame_id at a particular timestamp. twist expresses the robot's linear and angular velocities specified

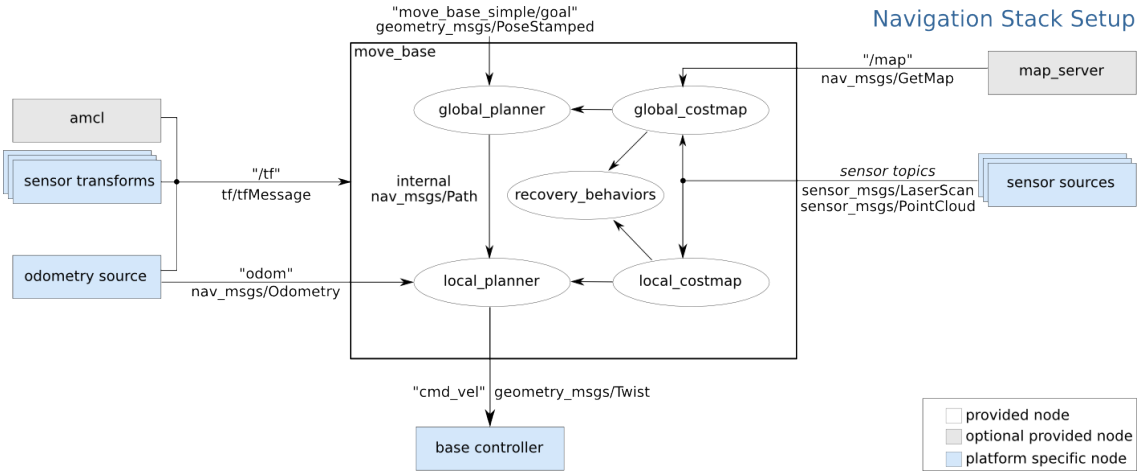

Fig. 8: The nodes and interactions used in the move_base package [7]

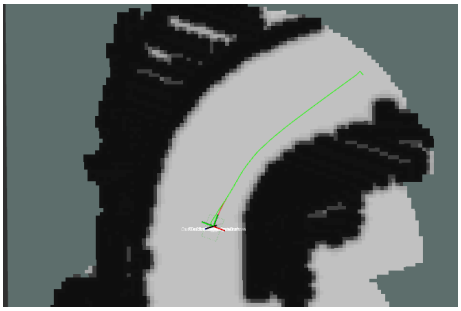

Fig. 9: global path generated by build-in global_planner

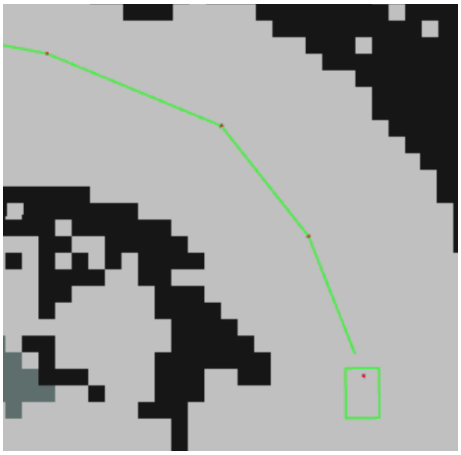

Fig. 10: global path generated by waypoint_global_planner

in the coordinate frame given by the `child_frame_id`. The `local_planner` uses the `odom` information to understand the robot's current location and motion in the environment to generate appropriate control commands to follow the planned path smoothly. To properly publish an accurate `odom` topic, we must first establish a `base_link` frame that adheres to the standard axis orientation convention in relation to a body. The standard convention dictates that the x-axis points forward, the y-axis points to the left, and the z-axis points upwards [6]. The `base_link` frame is associated with the body frame but rotated 90 degrees along the z-axis. This `base_link` frame is then published using the `static_transform_publisher`. In the `state_estimate_corruptor_node`, we establish the odometry between the `base_link` frame and the `world` frame. We set the `header.frame_id` to `world` and designate the `child_frame_id` as `base_link`. Next, the odometry data of body frame undergoes a transformation. Specifically, the `pose` and `twist` of the `base_link` frame are converted from the odometry data expressed in the body frame. Finally, we publish the odometry data of `base_link` frame to `odom` topic.

We found that in order to use this package, we must first determine the basic coordinate system and the robot coordinate system, especially the robot coordinate system, which has a crucial connection with the confirmation of the vehicle's own position and orientation. However, in the original system, the orientation and position of some coordinate systems were not correct, so we sorted out the corresponding coordinates and rotated them so that the vehicle can be positioned correctly, and the `move_base` can be used to initially move in the virtual environment. In addition, we also need to input the corresponding map to the `move_base`. At the beginning, we used the information of the depth camera as a map, and found that the effect was not stable. Due to a series of factors such as the computing power of the personal computer, the vehicle cannot accurately detect the edge of the road in real time when it is moving, which leads to the instability of the global and local planners, making the vehicle vibrate more strongly when moving, and sometimes even hits the road. shoulder. In order to solve this problem, we use `projected_map` as the `static_layer` of local and global maps, which greatly improves the stability of the map and makes the planned path information smoother. Until now, the build-in `base_local_planner` has already worked, it will be mainly used in **Route 2**. Finally, we

configured a series of parameters, which is also an important part of the whole local planning. For the shape of the robot(here the car), we use `foot_print` to generate a rectangle. For the global and local cost map there are also many important parameters, `inflation_radius` determines the risk of collision when the car is close to obstacles, it can be used both in generation of local and global path; `max_obstacle_height` and `min_obstacle_height` is to limit the vertical perceptual range; we can adjust `update_frequency` and `publish_frequency` to effect the update of the map, and likely, `planner_frequency` can effect the update of the global and local path; `obstacle_range` Determines how far away objects are recognized as obstacles, `raytrace_range` is the range of sensor; when we turn `turning_window` to true, the local cost map can move with the car. What's more, Zefeng added `observation_persistence` to the local cost map, so it is possible to keep the map detected for a period of time before, which is essential to the final effect of obstacle avoidance.

An alternative of build-in `base_local_planner` package is `teb_local_planner` [8], which provides better support for car-like robot. Planning and navigation of car-like robots is not intended explicitly by the navigation stack. However, the `teb_local_planner` tries to overcome this limitation by providing local plans that are feasible for ackermann drives [8]. This is accomplished by extending the non-holonomic constraint by a minimum bound on the turning radius resp. by satisfying $v/\omega > r_{min}$ [4]. The minimum turning radius is set by the parameter `min_turning_radius`. This `teb_local_planner` is utilized in **Route 1**.

III. Control

In this section, the control algorithms are introduced. The car takes the linear acceleration and turning angle as the controlled variables. As aforementioned, the planner outputs the desired velocity in the `base_link` frame, which needs a further transformation before being sent to the car. Here two controllers are designed, one is straightforward, and the other uses a PID controller with a state machine. The straightforward controller is used in the **Route 1**, and the PID controller with a state machine is used in **Route 2**.

A. Straightforward transformation

Yulong Xiao implements the straightforward transformation used in the **Route 1**.

As our car-like robot accepts acceleration, steering angle and braking as direct control value but the output `cmd_vel` contains only desired tranlational and angular velocity, a `cmd_vel_to_ackermann_drive` node provided by `teb_local_planner` tutorial [8] is used to convert the original `cmd_vel` of type `geometry_msgs/Twist` to `ackermann_msgs/AckermannDriveStamped` and publish it to topic `ackermann_cmd`. `ackermann_msgs` is time stamped drive command for robots with Ackermann steering, which contains steering angle for car-like robot

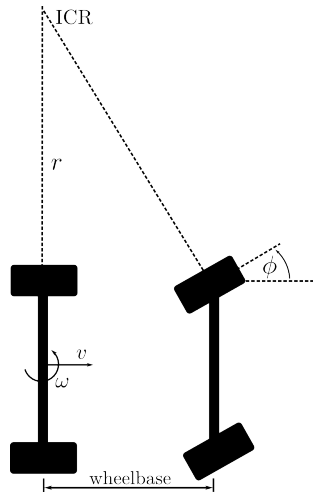

Fig. 11: relevant variables to calculate steering angle ϕ [8]

control and translational velocity. `cmd_vel` as the output of `teb_local_planner` contains desired translational and angular velocity v and ω respectively for commanding the robot. Given translational ang angular velocity, the steering angle ϕ is calculated by $\phi = atan(wheelbase/r)$, where the radius r is calculated by $r = v/\omega$ [8]. The relevant variables including the translational velocity v, angular velocity ω and the steering angle ϕ are illustrated in Figure 11. One critical parameter in this context is the wheelbase, which represents the distance between the front and rear axles. Since we lack exact measurements, we approximate the wheelbase value and set it to 3m.

The `controller_node` subscribe to `ackermann_cmd` and `odom` topic to calculate the final command value. To fine-tune the steering control, we introduce a gain factor that multiplies the steering angle subscribed from `ackermann_cmd` topic. This gain helps to adjust and optimize the steering response of our robot. In order to calculate the acceleration command for the car-like robot, a trivial Proportional (P) controller is utilized. The acceleration command is computed using the following formula: $a = k_p \times (v_d - v)$, where:

- a is the acceleration command,
- k_p is the proportional gain of the controller,
- v_d is the desired forward translational velocity, obtained by subscribing to the `ackermann_cmd` topic,
- v is the current forward velocity, obtained by subscribing to the `odom` topic.

The P controller's purpose is to regulate the acceleration of the robot by comparing the desired velocity (setpoint) with the current velocity and applying an acceleration proportional to the velocity error. The gain k_p determines the sensitivity of the controller and influences how quickly the robot reaches the desired velocity. Having the braking functionality automatically activated when approaching the goal is an advantageous feature of the P controller. This design eliminates the need to explicitly calculate a braking command value. As the local planner generates lower velocities as the car nears the goal,

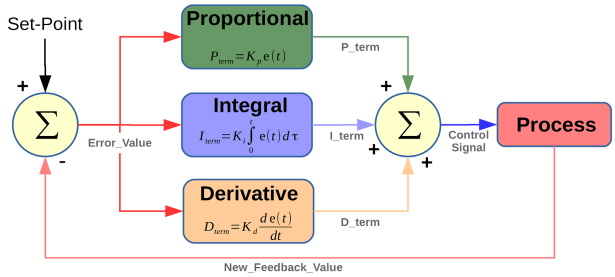

Fig. 12: PID Block

the P controller naturally responds with negative acceleration, facilitating a gradual and controlled deceleration.

B. PID controller pipeline

Zefeng Wang implements the PID controller pipeline, which is used in the **Route 2**.

In this part, Zefeng Wang implements two trivial nodes and a state machine node with the `smach` package as the interface between the `PID` package and the system. Furthermore, Zefeng Wang uses the extended Kalman filter (EKF) from the package `robot_localization` to filter the odometry, which leads to a more stable controller performance. A node to transform the twist information in the odometry from the `world` frame to the `base_link` frame is also implemented by Zefeng Wang.

1) PID controller: The primary thought of using a PID controller is to have a general controller without requesting the model of the car. In this project, there is no dynamic model of the car and the environment. It is impossible to precisely control the car's velocity with a pre-calculated acceleration and turning angle, due to the unknown road-holding capability and the wind resistance. The only solution is to use a closed loop to control the velocities dynamically.

PID is a widely used feedback control algorithm. It continuously calculates the control output based on the error between the desired setpoint and the measured process variable. The Proportional term responds to the current error, the Integral term addresses accumulated past errors, and the Derivative term anticipates future error trends, enabling effective and stable control in various engineering applications. The block diagram is shown in Figure 13.

We observe that yaw rate and linear velocity are highly coupled. Swerve may reduce the linear velocity due to the drift caused by the limited grip, and a consistent turning angle with different linear velocities leads to different yaw rates. Thus in the project, two PID controllers are designed to control them separately, one controls the linear velocity, and the other controls the angular velocity.

Before the controller is used, there are some prerequisites. The first is to separate the desired velocities for the two controllers from the message published by the planner, the second is to transform the odometry from the `world` frame to `base_link` frame, and the third is to extract the desired and current velocities and publishes them at the working rate.

All the topics subscribed and published by the `PID` controller are of the type `std_msgs/Float64`, different from the odometry, the desired velocity, and the control message the car accepts.

The `current_twist_node` subscribes to the `odom` topic, which contains the filtered state of the car. Then it extracts the current linear and angular velocity from the `odom` topic, and publishes the two speeds in the type `std_msgs/Float64`. The two current states are published at the same rate. The 2 PID units subscribe to these two topics and regard them as the plant states. Since the `odom` topic is published in the same frequency as the PID units, the velocity messages can be published directly without adjusting the rate.

The plant states should be in the body fixed frame to leverage the PID controller. While the provided twist information in the `current_state_est` message is in the `world` frame, a body fixed frame `base_link` is built, which has the x axis in the direction of the front of the vehicle and the z axis points in the direction of the roof. The `odometry_node` subscribes to the `current_state_est` topic, transforms the twist information from `world` frame to `base_link` frame, and publishes the new odometry message.

The `controller_node` subscribes to the two effort messages from the 2 PID units and publishes the `car_commands` topic to the unity. To achieve a stable control, the `car_commands` topic is published at a high rate, ten times the PID unit's working frequency. The publisher in the `controller_node` works separately from the subscribers. The two subscribers update the class properties, i.e., the linear acceleration and turning angle information, whenever they receive messages from the PID units. The publisher in this node builds the `car_commands` message from the two properties and publishes the message at the rate of 1000Hz. The `controller_node` uses the effort from the linear velocity PID directly. With the help of Qiyang Zong, we find the following formula. It shows that the angular velocity and the turning angle are proportional when the linear velocity is constant.

$$\left.\frac{\omega_\mathrm{r}}{\delta}\right)_\mathrm{s} = \frac{u/L}{1+\frac{m}{L^2}\left(\frac{a}{k_2}-\frac{b}{k_1}\right)u^2} = \frac{u/L}{1+Ku^2}$$

where u is the linear velocity, ω_r is the yaw rate, and the δ is the turning angle. So the `controller_node` regards the effort from the yaw rate PID as a scaled derivative of the turning angle. The turning angle is accumulated from the effort with upper and lower bounds.

2) EKF filter: The Extended Kalman Filter (EKF) is an extension of the Kalman Filter that allows for state estimation in nonlinear systems. It does this by linearizing the system's nonlinear dynamics using Taylor series expansion. By linearizing the system, the EKF can then use the traditional Kalman Filter equations to estimate the state and its uncertainty.

While there is noise on the twist and pose information, an EKF from the `robot_localization` package is used as the filter, which leads to a more stable PID controller and path planning performance.

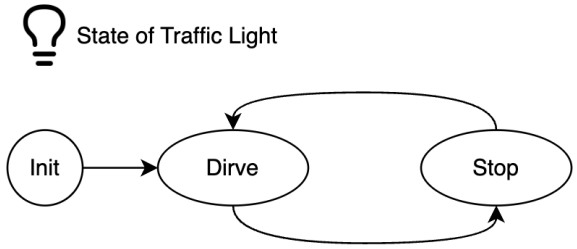

Fig. 13: State Machine: use the state of the traffic light as the switcher.

3) State machine: The `state_machine_node` subscribes to the `cmd_vel` topic, which contains the desired linear and angular velocities. And it separates and publishes the desired linear and angular velocities according to the traffic light state. To use one signal as the switcher in both states, a global flag is used. In the `state_machine_node`, there is a subscriber out of the two states, who update the switcher. The subscriber receives the current state of the traffic light and uses a counter as a low-pass filter. If the current state is the same as the last one, the counter increases, and if the current state is different from the last one, the counter is reset. So that the jitter of the state can be filtered. Since the working frequency of the planning unit and PID unit differ, the `state_machine_node` updates and publishes the desired velocities at different rates with the same method in `controller_node`. When the light is red, the state machine is in the stop state and publishes zeros as the setpoints. When the light is not red, it switches to the drive state and publishes the properties updated with the planning.

IV. Experiments

As aforementioned, there are two separate routes to achieve the goal. In this section, the experiments with 2 routes are introduced separately.

A. Route 1

By experiment, route 1 is particularly time-saving, allowing the car-like robot to finish the predefined path in about 200 seconds. The main goal of achieving time-efficiency is supported by the carefully chosen components in the navigation system, which are highly suitable for the specific environment and scenario.

The selection of the `waypoint-global-planner` is instrumental in achieving time-efficiency. Since the environment is deterministic and devoid of dynamic obstacles, this planner's direct approach is well-suited. It avoids the complexity of searching algorithms employed in dynamic environments and efficiently generates feasible paths towards the goal. The simplicity of this planner eliminates the need for sophisticated exploration, leading to faster path planning and, consequently, quicker completion of the predefined path.

The utilization of the `teb_local_planner` in Route 1 offers multiple advantages that contribute to the time-efficiency and effectiveness of the our car navigation. The `teb_local_planner` provides better support for car-like robots. This compatibility ensures that the local plans generated by the planner are tailored to the car-like control characteristics. As a result, the robot can navigate more smoothly and efficiently. The `teb_local_planner` also offers the capability to convert the original `cmd_vel (geometry_msgs/Twist)` to `ackermann_msgs/AckermannDriveStamped`. By converting to `ackermann_msgs`, the effort required to design and implement the robot's controller is reduced. This simplified format of the output command makes it easier to integrate with the robot's control system, saving time and resources during the development process.

In conclusion, the time-efficiency of Route 1 is a direct result of the carefully chosen components in the navigation system, the direct approach of the `waypoint-global-planner`, the car-like compatibility of the `teb_local_planner`, and the effectiveness of the P controller. Additionally, the deterministic environment and predefined global path eliminate the need for extensive path recalculations and allow the robot to efficiently follow the preplanned route, ultimately achieving the goal of finishing the predefined path in approximately 200 seconds.

B. Route 2

We implemented route 2 by using the traffic light detection, a self made `planning` package and build-in global and local planner and PID controller. Basic idea is to set goal points one by one, and eventually finish the given track. A state machine is used to switch the car state based on the state of the traffic light. Furthermore, we leverage the EKF to perform better planning and control. And with the PID control, we Compared to Route 1, Route 2 is a more general approach. This is because it utilizes automatic planning for generating the global path, reducing the need for manual setup and aligning better with real-life scenarios. However, in this project, Route 1 exhibits a clear advantage in terms of time efficiency.

Findings for Route 2

During the implementation process, we encountered and resolved some interesting issues.

1) Map quality effects plan quality: Initially, the car did not move as we expected, but instead, there were severe oscillations. We tried to fine-tune the PID parameters, but the results were minimal. Finally, Zefeng discovered that poor map quality could cause the planner to generate velocities with intense oscillations that couldn't be eliminated by the PID controller. Therefore, we also adopted a projected map in the local map, significantly improving map stability, which led to a more stable output from the planner and significantly improved the performance of the PID controller.

2) Using publishing frequency to compensate for sensor deficiencies: During the experimentation, we noticed that the sensor signals would jitter when the vehicle turned, thereby affecting global path planning. Sometimes, the planned path would even extend beyond the map boundaries, resulting

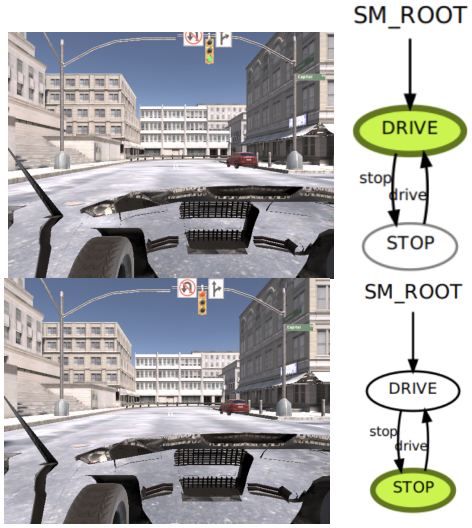

Fig. 14: State machine with different traffic light states

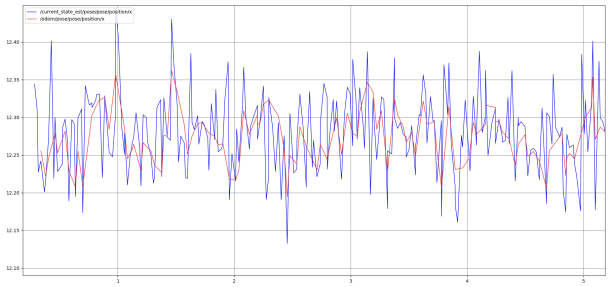

Fig. 15: EKF on pose

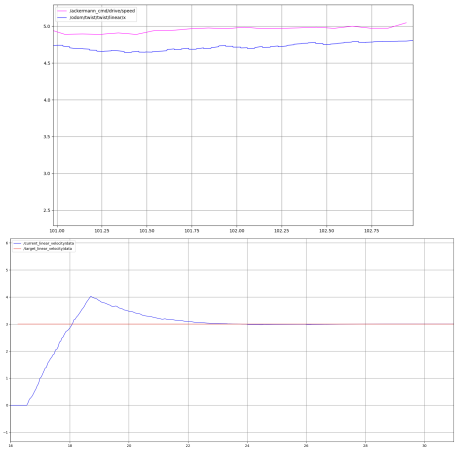

Fig. 16: PID controller eliminates the steady error. The upper image shows that there is a steady error with simple control. And the PID controller eliminates the steady error.

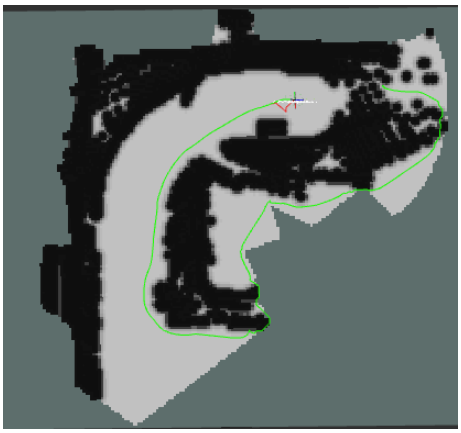

Fig. 17: incorrect global path since sensor jitter

in mission failure, as shown in Figure 17. To address this issue, we adjusted the planning and publishing frequencies of the global planner. This way, even if the sensor signals occasionally degraded, the global planner would update accordingly when the sensor signals were updated the next second, ensuring system robustness.

3) Consistent way points' orientation leads to a smooth local plan: The orientation is also a crucial factor when setting the target points. Initially, the target point information was manually acquired, resulting in overall low quality, particularly when there was a significant difference in orientation between consecutive target points. This difference led to unstable outputs from the planner, placing unnecessary pressure on the controller. Therefore, upon observing the set target points, we standardized the orientations using quaternions, which significantly improved the overall performance.

4) observation_persistence holds the curbing detected before for a while: In Route 2, the built-in planner automatically calculates the shortest path as the global path. However, during vehicle turns, the planned route by the built-in planner often comes very close to the curbing. If the detection of curbing is not accurate, it could lead to collisions. To address this issue, Zefeng introduced the `observation_persistence` parameter, which keeps previously detected curbs in the map for a certain duration. This implementation improved obstacle avoidance by providing better awareness of curbs and avoiding potential collisions.

REFERENCES

[1] ROS wiki depth_image_proc, http://wiki.ros.org/depth_image_proc
[2] OctoMap An Efficient Probabilistic 3D Mapping Framework Based on Octrees, http://octomap.github.io/
[3] Ultralytics YOLOv5 https://github.com/ultralytics/yolov5
[4] C. Rösmann, F. Hoffmann and T. Bertram: Kinodynamic Trajectory Optimization and Control for Car-Like Robots, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, Canada, Sept. 2017.
[5] G. Kouros, A Biswas: waypoint-global-planner, 2020, GitHub repository, https://github.com/gkouros/waypoint-global-planner
[6] T. Foote, M. Purvis: Standard Units of Measure and Coordinate Conventions, 2014, https://www.ros.org/reps/rep-0103.html
[7] ROS wiki move_base: http://wiki.ros.org/move_base
[8] ROS wiki teb_local_planner: http://wiki.ros.org/teb_local_planner

APPENDIX

Module	Contributors	Packages	Nodes	External package
Perception	Yichao Gao	perception_pkg	trafficlights_detect_node	no
		depth_image_proc	/	yes
		octomap	/	yes
	Chenyu Meng	yolov5	detect.py	no
Planning	Yulong Xiao	planning(Route 1)	waypoints_sending_server	no
			global_path_planning_client	
	Qiyang Zong	planning(Route 2)	client_judge	no
			service_sender	
	Qiyang Zong, Yulong Xiao	auto2dnav	/	no
	Qiyang Zong, Yulong Xiao	move_base	/	yes
Control	Yulong Xiao	auto2dnav	cmd_vel_to_ackermann_drive	no
		controller_pkg	controller_node	no
	Zefeng Wang	controller	controller_node	no
			curretn_twist_node	no
			odometry_node	no
		state_machine	state_machine.py	no
		pid	/	yes
		robot_localization	/	yes
		smach	/	yes

TABLE I: Task distribution

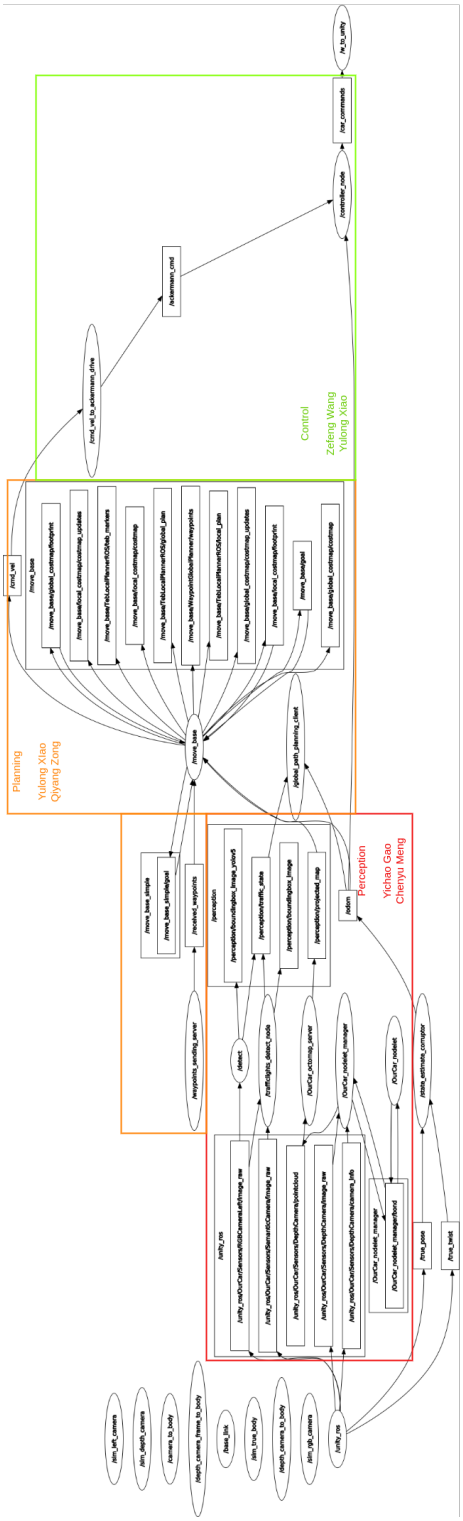

Fig. 18: Rosgraph for Route 1

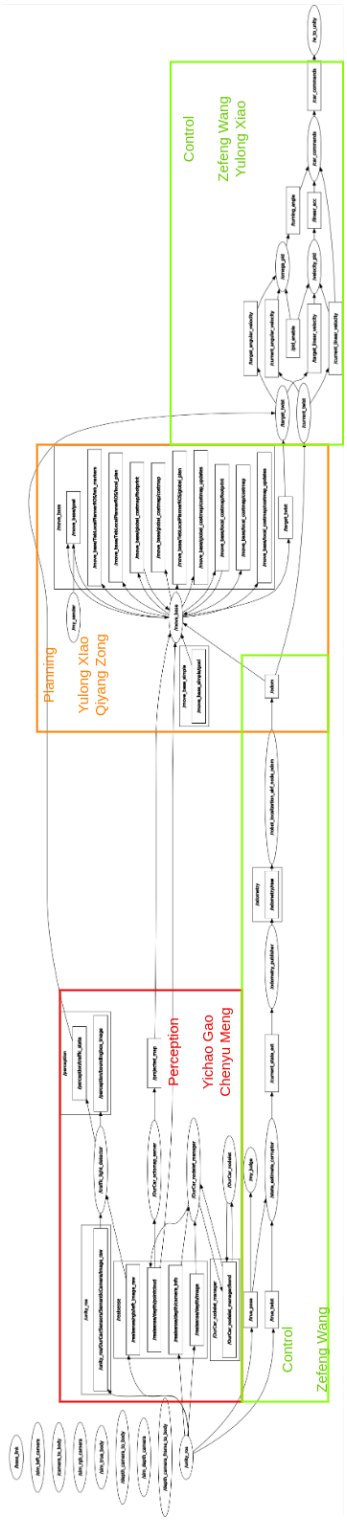

Fig. 19: Rosgraph for Route 2