Introduction to ROS Group Project: Autonomous Quadruped

Group 23: Rui Xiao, Yiwen Liu, Bo He, Zhixian Huang, Ali Rabeh

1 Introduction

In this section, an introduction to our project will be given. To solve the autonomous quadruped problem, we can divide this task into perception, path and trajectory planning, and controller parts. Meanwhile, the state machine will guide the quadruped to follow different task strategies. The following Figure 1 shows the general pipeline of our project.

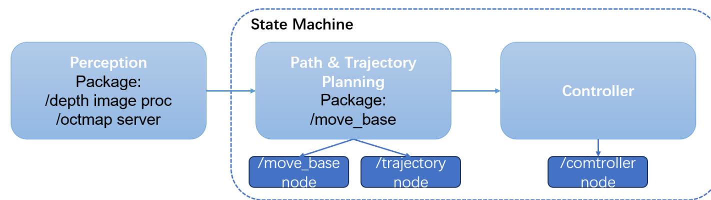

Figure 1: **Pipline of our project**:The package means which kind of outside package we used, the last line shows you the different ROS nodes.

We aim to solve the problem **without any previous information for the quadruped**, so we decided to use the perception to give the real-time map information to the path & trajectory planning node, then the path & trajectory planning node will according to map information to formulate the future path. Based **only on the real-time path planning information**, we switch the different state of state machines to publish the next desired point that the RoboDog should arrive at and the corresponding control state. With the next desired point and control state, we can specify different control parameters to achieve the desired motion.

However, due to the global map being complex, it is very hard to judge the state according to the trajectory information, we finally only achieved to let the robot dog move to the front of the parkour. We will analyze this problem in the section 9.

2 Perception

2.1 General Perception Pipeline

We design our perception pipeline as dsiplayed in Figure 2. First of all, the "depth_image_proc package" , which is subsribed to the "camera_info" and "im-

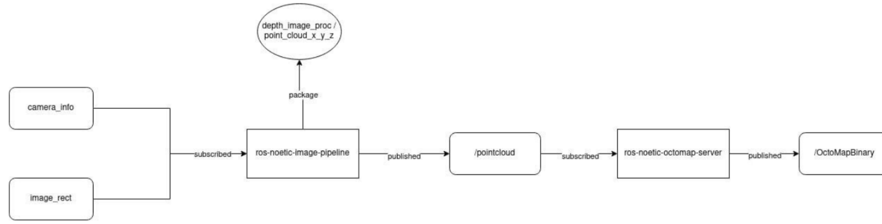

Figure 2: The proposed perception pipeline aims to generate an occupancy map from a depth image, which will be further used in the planning part.

age_rect" topic, is used to convert the camera data into point clouds, and publish under the /point_cloud topic. Secondly, we use "ros_noetic_point_cloud_server" to convert the point cloud into octomap. Since we only need 2D Octomap for the planning, "ros_noetic_point_cloud_server" can also intrinsically convert it to 2D occupancy map. Now the projected 2D occupancy map is published under the "/projedcted_map" topic.

2.2 TF Transform & Filtering

In order to get the correct pose of the point clouds, we need to add transformation of the point cloud frame with respect to the base frame. Here we configure it as "0 0 0 0 0 -1.7453", this means that the point clouds will rotate to the horizontal plane so that we can better visualize in Rviz.
Besides, since the initial Octomap is very noisy, containing irrelevant ground points, we set the "point_cloud_min_x" to be 0.15, "point_cloud_max_x" to be 0.4 to filter out the ground points as well as decrease irrelevant points to generate a cleaner Octomap.

3 Path Planing

3.1 Implementation of move_base package

For path planning, we used the outside package move_base to accomplish this task, and the configuration files are in the "/src/simulation/param" folder. We don't have the global map at the initial time, so we only use the global planner of the move_base package. The move_base subscribe "/projected_map" from perception part to get current map information. It also subscribes the set goal information from "/move_base_simple/goal" topic to get the final goal of the path, note current state information is from "/tf" topic. Based on configuration files and internal algorithms, it will publish a global path from the current state to the set goal point under "move_base/NavfnROS/plan" topic. The above information is contained in Figure 3.

3.2 Important parameters configuration

There are some significant parameters to be set in the .yaml files for move_base. Following we only introduce to you some important parameters to be noted in

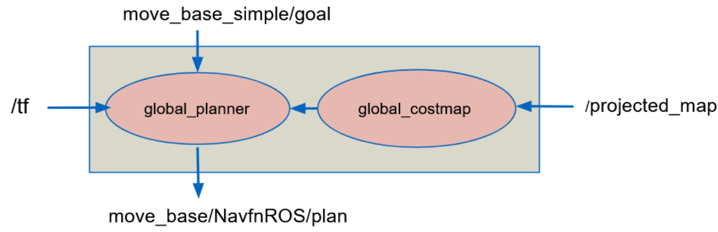

Figure 3: **move_base topic subscriber and publisher**

some files.
In the "**costmap_common_params.yaml**" file, we need to specify:

1)**Footprint**: this embodies the size of a RoboDog, after some experiments, we set this value to "[[-0.2, -0.2], [-0.2, 0.1], [0.2, 0.1], [0.2, -0.2]]".

2)**Inflation_radius**: The value expands the cost area outside the collision area, allowing the robot to plan paths to avoid obstacles. Some of the passages in the map are very narrow, so we let the value smaller such that the path will not be blocked up. But this value should not be too small, otherwise, the robot will come through the obstacles. We finally choose this value to 0.26.

3)**Cost_scaling_facto**: The smaller the value is, the cost of coming through the obstacle is larger. So we choose this value to 0.

4)**Map_topic, point_cloud_sensor**: some topic names should be consistent with other files.

In the "**global_planner_params.yaml**", we need to specify the global planner using Dijkstra algorithm to plan the path.
In the "**global_costmap_params.yaml**", we need to specify the global_frame to the world to set Odom coordinate. And the "robot_base_frame" should be true_body.

4 State Machine

The state machine contains INITIAL, STRAIGHT, TURN, and END four states. The main function of different states is to identify different motion states of RoboDog and publish different next desired points according to the state. We designed some functions using the included angle between the current velocity and distance vector to the desired point. With flag variables to distinguish different states. The core to distinguish STRAIGHT and TURN mode is to check the planning path from move_base. If the included angle is smaller than 10 degrees, we will define this situation as STRAIGHT. If the included angle is between 10 degrees and 150 degrees on both sides, it will be in TURN mode. Otherwise, the path will be invalid and will regenerate again. Figure 4 shows the state machine and corresponding control_state for the controller node to use.

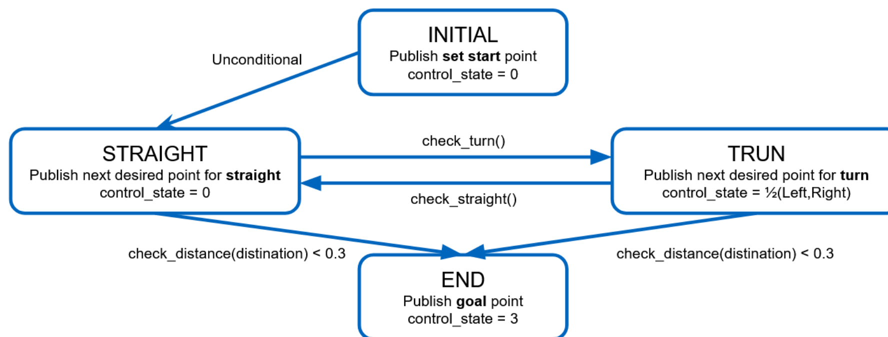

Figure 4: **State Machine**: Each state will publish different next desired point and corresponding controller state.

5 Trajectory Planning

The trajectory planning is based on the state machine. These two functions are merged in the trajectory node. The trajectory node publishes the next desired point as our next trajectory point based on path planning. From INITIAL mode, it will first let the RoboDog move to a specific start point. Then in the STRAIGHT mode, it will publish the next point 1.0-meter far away point from the current. In the TURN mode, it will publish a 0.4-meter far away point from the current. When it is very near to the goal point, it will publish the goal point as our desired point.

6 Controller

6.1 Function

The controller can realize that the RoboDog reaches a target point set in a short distance from the current position, and then repeats this step to reach the final destination. It mainly includes three states: going straight, turning left and turning right. We can compute theta value using cosine law like following Figure 5, and judge the direction using cross product of two vectors.

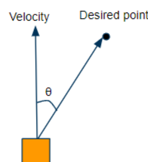

Figure 5: Velocity direction and desired direction

6.2 Input

x: Current position of the RoboDog in 2D.
xd: Desired position in 2D.
v: Current speed of the RoboDog with a direction.

6.3 Speed of the RoboDog

We keep the phase between the front and back legs at 0 degree and then the RoboDog will keep a constant straight speed.

6.4 Rotation

We mainly use the following formula to obtain the rotation angle:

$$steering_angle = rotation * (K - cos\theta)$$

We can get the velocity orientation and distance orientation(defined as $xd - x$) from the map, and then can we get the angle between them by calculation with vector formulas. If the angle is greater than 40 degrees, and the vector cross produkt is less than 0, the robotdog will turn right, if the vector cross produkt is more than zero, the robotdog will turn left. If the angle is less than 40 degrees, the RoboDog will keep going straight. Rotation is a parameter that changes with distance, K is an adjustable constant value if it is far away from the target point, the robot dog will keep going straight. and the steering angle is close to 90 degrees. If it is very close to the target point and theta is greater than 40 degrees, the robot should turn left or right according to the conditions, and the steering angle should be close to 45 degrees or minus 45 degrees.

7 Keyboard Control and Gait Walking Parameters

To test our the gait walking parameters, we implemented an alternative control method using the input from the keyboard. For this we used teleop_twist_keyboard package that publishes different geometry twist messages into the topic "/cmd_vel". The controller subscribes to this topic and depending on the twist input, decides on how to control the robots using the gait walking parameters.
We determined the following numerical parameters depending on the input.

Table 1: Gait Walking Parameters

Angular Velocity	**0**	**1**	**2**	**3**	**4**
Jump Type 1	0	0	8	35	10
Jump Type 2	0	0	8	35	7
Forward	0	90	5	0	10
Right Turn	0	45	0	0	7
Left Turn	0	-45	0	0	7
Stop	0	0	0	0	0

We used the angular velocity 1 which corresponds to the phase between front left and back right legs and front right and left back legs, to control if the robot will move forward or turn. // For the obstacles we had to jump promoters, the first one worked for obstacle 1 and the second one worked for obstacle 3. For Obstacle 2, we simply used the forward case parameters since we had enough frequency of legs to overcome the obstacle.

8 Results

8.1 Real-time Result

In the real-time perception&planning method that we used, the RoboDog is able to perceive the surroundings plan a real-time trajectory, and let the dog move along the line, although very slowly. The RoboDog can reach the first turn before the obstacle, as shown in this video.

8.2 Keyboard Control Result

In the alternative method that we used, we mainly used keyboard control to guide the dog to get over the obstacles. Here our RoboDog can go much further and successfully move over the obstacle. The resulting video is included in this drive folder.

9 Problems and Improvements

- As mentioned in Section 1, we want to **only use real-time planning information** to control motion. The problem is the self-motion estimation is not perfect and it will lead to some mistaken judgment. As a result, we designed kind of complicated algorithms to judge which state should remain and under which requirement the state will change. However, we can successfully change the mode from STRAIGHT to TURN, and the corresponding control parameters will change from "0 90 0 0 7" to "0 45 0 0 7". But when it switches back from TURN to STRAIGHT, due to the window fulfilling the STRAIGHT requirement is very small($\pm$10 degrees), although the control parameter has changed to "0 90 0 0 7", the RoboDog still continues to turn until missing the STRAIGHT window. We analyze this problem due to the RoboDog is not so sensitive to control parameter change in a small time. But we have not found a solution to solve this.

- Becasue of the problem mentioned before, the current version of the controller only uses the information about x,y coordinates of next desired point to control the RoboDog. The control parameter is defined by some variables and changes in real-time. From some point, it is a kind of **common method to control both straight and turn modes**. But the problem is this method is very difficult to adjust and the RoboDog moves very slow.

10 Task Distribution

- **Perception**: Rui Xiao, Zhixian Huang
- **Path Planning**: Yiwen Liu, Bo He
- **Trajectory Planning & State Machine**: Yiwen Liu
- **Controller**: Bo He, Yiwen Liu, Rui Xiao, Ali Rabeh
- **Report Written**: Yiwen Liu, Rui Xiao, Bo He, Ali Rabeh

Note:

- One of the initial group member, who is mainly responsible for the control part, de-registered this course last friday and gave up to continue. Ali Rabeh joined our group this week, due to all of his group members gave it up. He successfully implemented the keybord control method.
- Rui Xiao and Yiwen Liu shared the same laptop for this project, so they used the same account for pushing the code to Gitlab :)