Introduction to ROS Group Project: Robodog

Group 28
Xin Yutong, Lin Run, and Zhang Yihan

1. Perception

First, we use package: depth_image_proc to convert the depth image into a point cloud. Then use the point cloud information to generate an Occupancy Grid in Octomap.

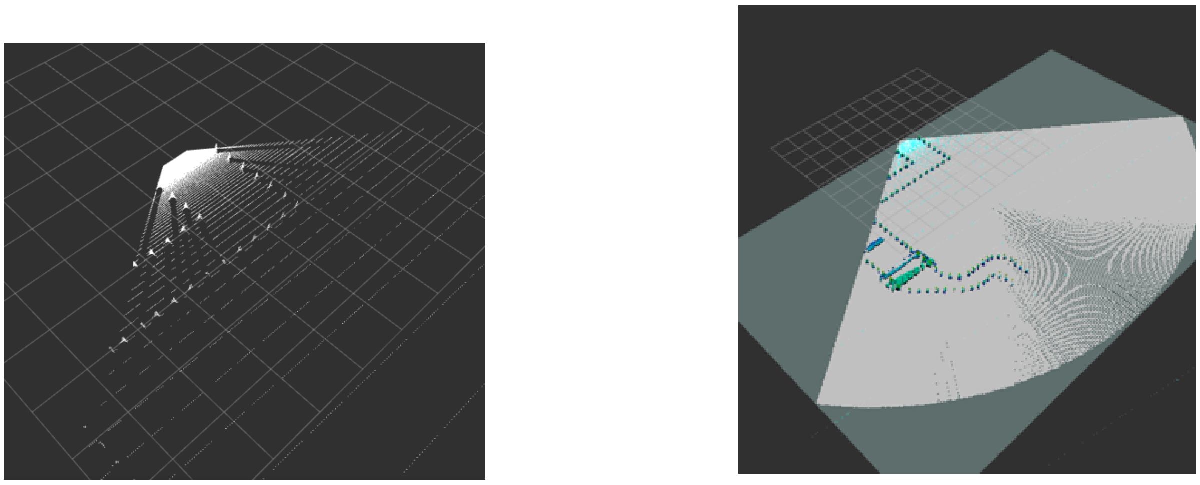


Fig. 1. PointCloud(left) and OccupancyGirdMap(right)

1.1. Get Octomap

we use octomap_server to generate the occupancy grid/map for the RoboDog. In order to prevent the problem of continuous stacking of obstacles caused by the repeated refreshing of the point cloud, the filtered_point_cloud.cpp file is used for filtering.

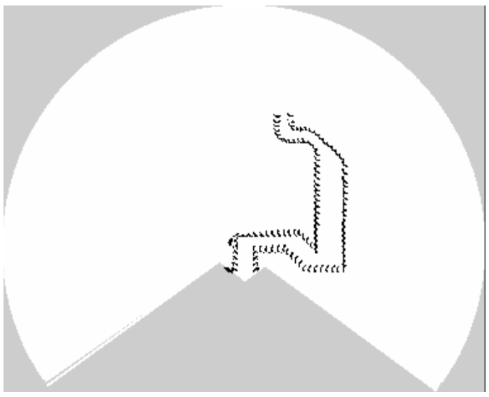


Fig. 2. Offlinemap

1.2. An Alternative Solution

After the map is generated, in order to help the follow-up path-planning to proceed better, we decided to let the Robodog rotate at the start point first, and obtain the global static map through camera scanning. Use the rosrun map_server map_saver command to save the static map generated after scanning as two filesoffline_map.pgm and offline_map.yaml.

We should establish the conversion between the frame of the depth camera and the frame of the world coordinates so that the point cloud scanned by the camera can be displayed in the 2D plane coordinate system of the Rviz environment. In the file: simulation.launch we wrote the following command: ¡node pkg="tf2_ros" type="static‘transform_publisher" name="depth_camera_frame_to_body" args="0 0 0 0 0 -1.745

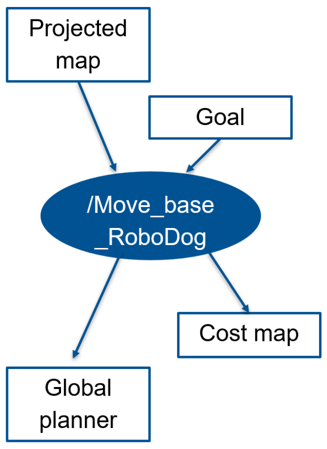
Used external packages: depth_image_proc,octomap_server,map_server,map_saver.

2. Path Planner

In order to get a path with no obstacles in between, we need to filter steps and slopes in the map. Only point clouds higher than steps or slopes will be taken into account, thus $z \in [0.167, 3]$.

The official package "move_base" from ROS provides a way to collect and integrate the information about sensor perception and the odometry from the moving robot, Use the traditional Dijkstra Algorithm and the costmap from the global view to generate the optimal path. We use the occupancy grid in 2D plane from "octomap_server_node" and publish the topic "/goal_topic" about the destination position of the robot dog.

At first, we used dynamic point cloud information as the observation source, hoping that move_base could read the point cloud information to generate an obstacle layer and then generate a costmap, but after trying, it was found that the newly generated obstacles will be superimposed on the old generated obstacles during the moving of the robot. As a result, the costmap cannot represent the true position of the traffic cones and cannot build a valid path. Actually sometimes obstacles will appear dynamically on the road and cause the global path to pass through the traffic cones. Receiving the point cloud topic and forcing a reset does not solve this problem. Therefore, in the end, we first control the robot rotate 360 degrees clockwise at the start point to get the global map, and pass the global map as a static map into the move_base node. The first advantage is that we can get a stable costmap and a stable path. The inflation_radius of the obstacles was chosen as 0.13 after the trial. And the cost_scaling_factor will decide how expensive the inflated area around the obstacles. A bigger value means a smaller cost. We decide to set it to 1.0 to get a reasonable path.



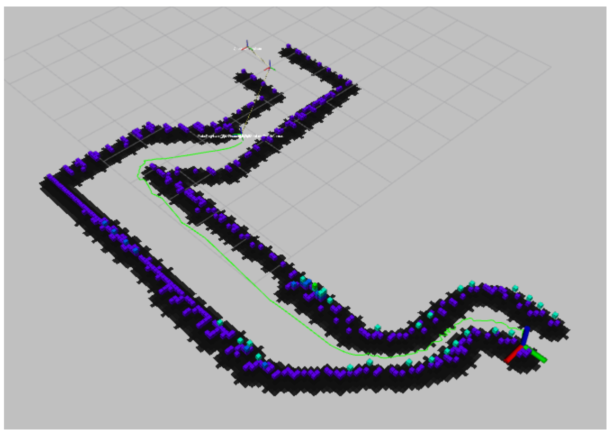


Fig. 3. MoveBase(left) and PathPlanning(right)

3. Trajectory Planner

The purpose of the trajectory planner is to subscribe to the global planner information sent by "move_base“, which represents the planned path for the robot. After filtering, the trajectory planner selects specific waypoints along the path that can serve as local goals.

3.1. Start Point Check

First, we need to verify whether RoboDog has reached the starting point. If not, this node will continuously transmit the starting point coordinates as the local goal. Only once the starting point is reached will the filtering of valid waypoints along the path commence.

3.2. Principle of Waypoints Filtering

Two Principles are used to identify reliable local goals from the path. For each pose in the path, the pose position and the rotation angle of RoboDog will be checked.

We aim for the target point to be positioned within a suitable distance in front of RoboDog. Simultaneously, we want the rotation angle θ to stay in the range $[-\pi, \pi]$ at all times. The filtering based on rotation angle helps avoid selecting paths that are situated behind RoboDog, which is shown in Fig. 4.

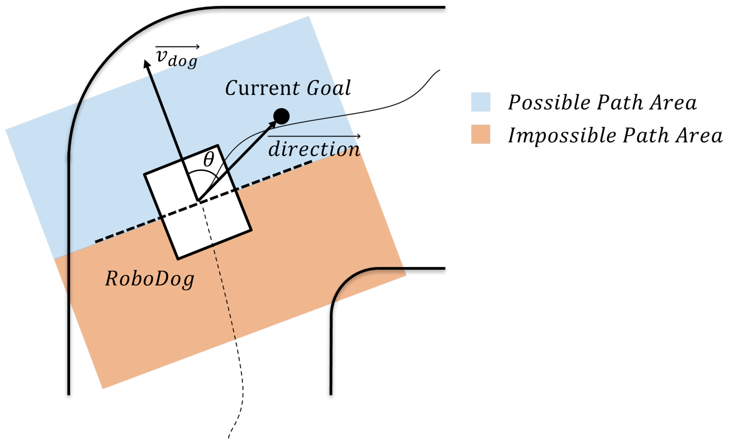


Fig. 4. The potential area for choosing local goal

4. State Machine and Controller

4.1. Calculation of the rotation angle and distance

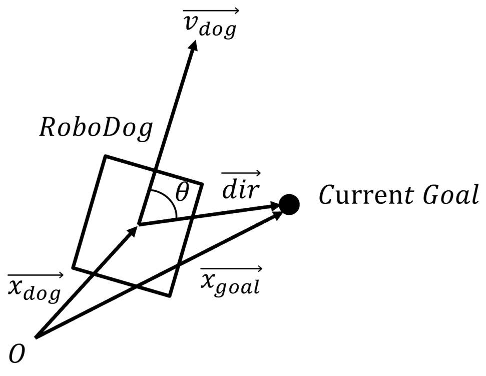

$\overrightarrow{v_{dog}}$
RoboDog
θ
$\overrightarrow{dir}$
Current Goal
$\overrightarrow{x_{dog}}$
$\overrightarrow{x_{goal}}$
O

Fig. 5. Calculation graph

Rotation angle and distance are the two main control variables in our control loop. To calculate the rotation angle and the distance between RoboDog and the current goal, we subscribe to the topic "current_state_est" which represents the true position in the "world" frame, and the topic "/desired_state". In the xy-plane, we name them $\mathbf{x}_\mathrm{d}$ and $\mathbf{x}_\mathrm{goal}$. Easily we can get equation (1), (2), (3)

$$\mathbf{x}_\mathrm{dir} = \mathbf{x}_\mathrm{dog} - \mathbf{x}_\mathrm{goal} \tag{1}$$

$$\theta_\mathrm{goal} = \tan^{-1}(\mathbf{x}_\mathrm{dir}) \tag{2}$$

$$\theta_v = \tan^{-1}(\mathbf{v}_\mathrm{dog}) \tag{3}$$

Combining the above equation(1), (2), (3) we get equation(4), (5)

$$\text{dis} = |\mathbf{x}_{\text{dir}}| \tag{4}$$

$$\theta_{\text{yaw}} = \theta_{\text{v}} - \theta_{\text{goal}} \tag{5}$$

Where *dis* stands for the distance scalar and θ_{yaw} stands for the rotation angle.

4.2. State Machine

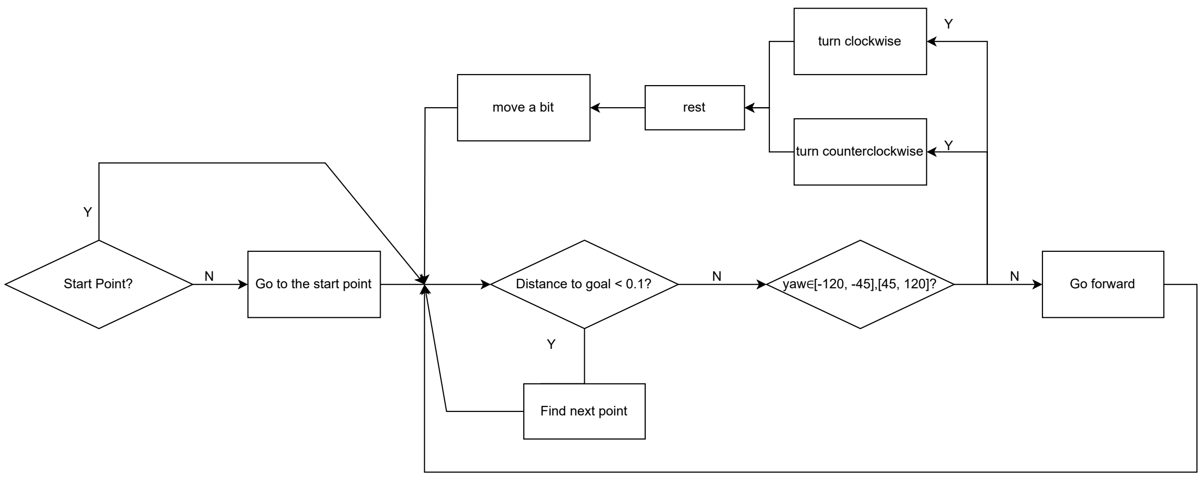


Fig. 6. State machine

- **Receive local goal from "pathplanner_node"** We create a variable named "receive_goal" that represents whether or not we are listening to the local goal. If RoboDog is $< 0.1m$ away from the goal, we consider that Robodog has reached the goal and is ready to listen to the next goal, so we set "receive_goal" to "true". Set "receive_goal" to "false" immediately after receiving a local goal. Set RoboDog state to "move a bit" when there is no suitable target point incoming, thus. move forward slowly and wait for a new target point to be issued.
- **Rotation state** Determine the interval of the rotation angle θ_{yaw} calculated from section 4.1, if the rotation angle θ_{yaw} is in range of $[45°, 120°]$ or $[-120°, -45°]$, entering the rotation state. After the rotation state, RoboDog will rest and move a bit to stabilize itself, since RoboDog's movement is affected by random disturbance.
- **Go forward state** After the rotation angle is settled. RoboDog can easily go forward toward the local goal.

4.3. Control Mode

We have designed six modes of operation including stop, go forward, slow walk, clockwise rotation, counterclockwise rotation, and jump.

Although we haven't designed the detection of steps and slopes in the state machine yet, we have tested that RoboDog can climb the steps successfully in jump mode.

5. Result

We have completed all components of perception, path planning, trajectory planning, controller, and part of the state machine. In addition to detecting steps and slopes, RoboDog can walk along the planned paths. Here's a video that shows how RoboDog works. And here is the link to the rqt_graph which demonstrates communication between different nodes.

We also provide several local goals in the code as an aid to help the mechanical dog move forward when the path planning is not performing well.

6. Remain Task

What we still need to do is the detection of steps and slopes. We haven't started working on this part yet due to time constraints.

We can get the height information of the obstacles in the map via nav_msgs/OccupancyGrid. In the Perception section we filtered steps and slopes by setting the height range. Now we can create a new map without filtering the height information, and determine whether there are steps/slopes in front of us by determining the z-axis height during the RoboDog's movement.

7. Contribution

Run Lin: Perception and Mapping
Yihan Zhang: Perception and Mapping
Yutong Xin: path planning, trajectory planning, state machine, controller

8. References

1.http://wiki.ros.org/depth_image_proc

2.http://wiki.ros.org/octomap_server

3.http://wiki.ros.org/map_server

4.http://wiki.ros.org/map_server

5.https://pointclouds.org/documentation/tutorials/passthrough.html

6.http://wiki.ros.org/move_base