Introduction to ROS Group Project: Autonomous Quadruped

Yinglei Song, Dian Yuan, Zhelin Yang, Hang Li, Zhaoqi Zhou

Group 14 AwesomeDog

1 Perception & Mapping

Used external packages: depth_image_proc, octomap_server.

1.1 Implementation of Packages

First, we implement depth_image_proc to convert depth image to point cloud. Then we use octomap_server to generate map for the quadruped. In order to perform path planning and step & slope detection at the same time. We generate two maps.

In the first map, only point cloud between $z = 0.15m$ and $z = 5.0m$ is taken into account. Step and slope are no longer regarded as obstacles in this map, but traffic cones are still visible. Then we can apply path planning algorithm from current position to the finishing line without interruption.

In the second map, we filter out the ground, while keeping all the point cloud from $z = 0.0m$ to $z = 5.0m$, so that we can get occupancy information of step and slope from this map.

Besides, these two maps are all projected into 2D maps (message type: nav_msgs/OccupancyGrid) for further applications. This part of code is written in simulation.launch.

1.2 Step & Slope Detection

After getting a path (message type: nav_msgs/path) from move_base, we must know where step and slope are, in order to change amplitude and perform specific actions. For one point in the world frame, we take its coordinates x and y. We can also get map width, map height, origin of map in world frame, map resolution from nav_msgs/OccupancyGrid. As occupancy values are stored in a 1D array, a transformation should be applied. Here is the procedure.

$$\begin{aligned} Columns \text{ (in 2D array)} &= (x - origin_x)/resolution \\ Rows \text{ (in 2D array)} &= (y - origin_y)/resolution \\ Index \text{ (in 1D array)} &= (Rows \times map_{width}) + Columns \end{aligned}$$

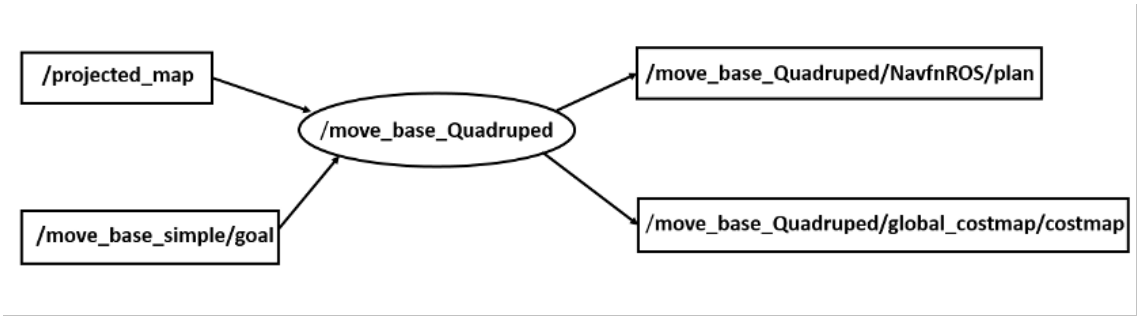
Then we can get occupancy values from messages. In Octomap there are only three different values. 100 means occupied, 0 means free and -1 means unknown. For the purpose of robust detection, we take first 10 points in the path, if more than 5 of them are occupied, then we determine that there are step or slope in front of robot. This part of code is integrated in the planning_node.

2 Path Planning

Used external package: move_base.

2.1 Implementation of Package

We use move_base to generate a global path for the robot using Dijkstra's algorithm. Then trajectory planner can extract waypoints from this path. We do not use local planner in the move_base. The configuration files are located in src/simulation/param. The map source of move_base is the projected 2D map without step and slope. Based on that, a cost map will be generated. The global_frame is "world", and the robot_base_frame is "true_body". The following is pseudo rqt_graph of this part. Move_base subscribes topic "/projected_map", which is occupancy map, to generate and publish costmap. It also subscribes topic "/move_base_simple/goal" for planning. Then a global path is published.



2.2 Important Parameters

- inflation_radius: 0.352.
  Inflation is very important for path planning, because track is defined by traffic cones. One the one hand, we should inflate these traffic cones into "wall", so that the path cannot go through them. On the other hand, if we inflate traffic cones too much, there will be no empty space for path planning. Besides, inflation helps keep enough distance between robot and obstacles. After long-time parameter tuning, we found 0.352 is the suitable value.

- cost_scaling_factor: 0.0.
  Generally, cost_scaling_factor influences the cost in the inflated area. Path is calculated based on this cost. A bigger value means smaller cost. After trying a lot of settings, we decide to set it to zero. Under this condition, the path will hardly go through the inflated area.

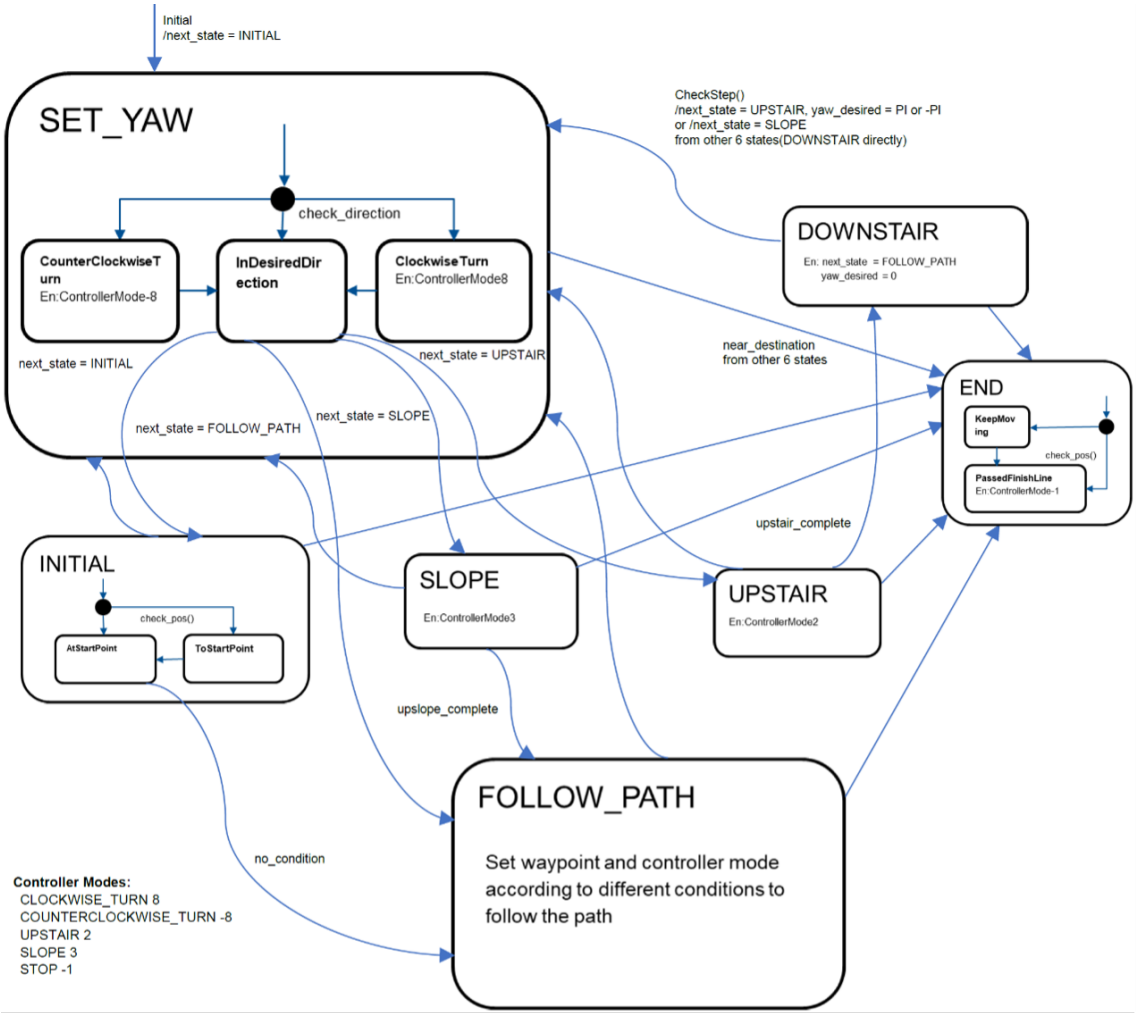
- footprint: [[-0.2, -0.2], [-0.2, 0.1], [0.2, 0.1], [0.2, -0.2]].
  Footprint defines the corners of our robot, which also reflects size. Based on estimation we set the size as 0.4×0.3, and origin is the center of robot.

3 State Machine

The states experienced by the robot are described by a hierarchical state machine with 7 main states: SET_YAW, END, INITIAL, FLLOW_PATH, UPSTAIR[1], DOWN_STAIR and SLOPE. With the variable next_state and some functions as condition, the state can be changed as desired. When entry some of the states, the controller mode will be changed accordingly. The structure of the waypoint we use has 3 variables, for the position information we only need to use the x, y components, and we use the z coordinate to communicate the state information to the controller. The robot will have different velocity, rotation speed and amplitude under different controller modes. In the controller node, totally 6 modes are defined, which are default, CLOCKWISE_TURN, COUNTERCLOCKWISE_TURN, UPSTAIR, SLOPE and STOP.

The state machine and controller modes with corresponding numbers are shown in the figure blow.

[1]We'd like to thank Ulf Kasolowsky from Group 11 very much for sharing his experience of getting the robot to walk up the step backwards, making this state possible.



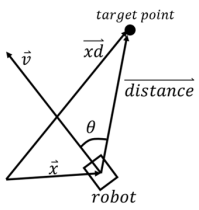
4 Trajectory & Visualization

This node subscribes the waypoint from planner and publishes trajectory to the controller. In our case the waypoint is usually quite close to current position so we can approximate the waypoint as a trajectory point. Besides, it also publishes some markers representing the real trajectory of the robot and the current target point to realize visualization in rviz.

5 Controller

5.1 Function

Given a target point, the controller adjusts the forward orientation of the robot towards the point and drive the robot to reach the point. The variables used in the controller are shown in the figure below.



5.2 Input

- x: current position of the robot, consisting of a 2D array representing the direction x and y;
- v: current speed of the robot, consisting of a 2D array representing the direction x and y;
- xd: target position of the robot, consisting of a 2D array representing the direction x and y;

5.3 Speed Gain

The speed gain is the control variable to control the magnitude of the speed. It depends on the absolute distance between the current position and the target position. The farther the robot is from the target position, the larger the speed will be. There is a maximum(1.0) as well as a minimum(0.2) limit for the speed gain.

5.4 Orientation and Rotation

5.4.1 Orientation

The moving orientation of the robot is described by the speed vector. The goal of adjusting the orientation is to guide the speed vector into the same direction as the distance vector. The distance vector is defined as $xd - x$.

5.4.2 Rotational Direction

The rotational direction is decided by the cross product of the speed vector and the distance vector. If the cross product is larger than zero, the robot should turn left and if on the contrary, turn right.

5.4.3 Rotational Speed

The rotational speed depends not only on the angle θ between the distance vector and the speed vector but also on the absolute distance between the robot and the target position. For θ, the rotational speed will be linear to the absolute value of $\cos\theta$. The influence from the distance is described with a scaler:

$$scaler_{distance2rot} = \min(4 \cdot distance^2, 3) + \frac{0.05}{distance^2 + 0.01}$$

The front part of $scaler_{distance2rot}$ takes effect when the robot is far from the target point while the lateral part takes effect when the robot is pretty near to target point.

6 Results

The robot was able to follow the dynamically planned path and identify the step/slope in the path to reach the end point in a total time of about 5min 30s. Here's an example video that captures the entire process of the robot reaching the finish line.

7 Task Division

- Perception & Mapping: Yinglei Song
- Path Planning: Yinglei Song, Hang Li
- State Machine: Dian Yuan, Zhaoqi Zhou
- Trajectory and Visualization: Dian Yuan
- Controller: Zhelin Yang