Autonomous Quadruped

Project summary of Team 8

0. Team members

Zhiping LI: complete controller part; adjust path planner parameters for better performance.

Ying SU: complete perception pipeline to get the cost map; develop path planner.

Xinyu JIA: complete perception pipeline to get the cost map; develop path planner.

Chenglin LI: adjust path planner parameters; complete final documentation.

Johannes Heidersberger (deregistered): helped build the first version of controller.

1. Project overview

The initial goal of the project is to pass a parkour, including obstacles in the environment and slopes on the road near the target point, with the robot in minimal time.

In this project, we first tried to control the robot to achieve Simultaneous Localization and Mapping (SLAM) without any additional information (such as a static global map, etc.), and reach the target point in the process of mapping. But in the process of implementation, we finally gave up on this solution due to time constraints and some configuration issues with the initial version of controller.

We finally decided to use the keyboard to control the robot to explore the surrounding environment to obtain a static global map, which was used to provide information for subsequent path planning. Next, use the stdr_move_base package to generate local and global paths, and write the controller to enable the robot to follow the path and reach the specified target point.

2. Implementation

The detailed steps of the project implementation can be found under the readme file of the git repository.

2.1. Project structure

Figure 1 shows how we built this project. In addition to the given packages, we also use two additional packages for perception and path planning respectively.

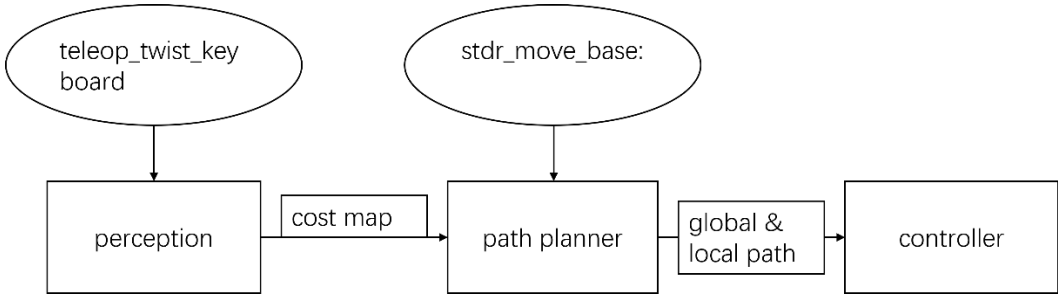

Figure 1. Project flow

- teleop_twist_keyboard [1]: this package is used to control the robot with keyboard to explore the environment and get the cost map.
- stdr_move_base: this package is used to generate local and global (optimal) path for the robot to reach the target point.

2.2. ROS nodes and functionality

The following lists some of the ROS nodes we use in the project and their main functions:

- "/stdr_move_base": all the information generated by the package "move_base"
- "/stdr_move_base/TrajectoryPlannerROS/global_plan": desired points provided by the package "move_base"
- "/current_state_est": current position and orientation of the robot in quaternion form
- "/rotor_speed_cmds": control signals to control the robot in quaternion form [vel_cmd, turn_cmd, amp_cmd, 0]

2.3. Perception and map generation

In this part, the depth camera of the robot is leveraged to perceive the robot's surrounding environment. At the same time, the keyboard is used to control the movement of the robot to walk through the entire path, and the depth images captured by the depth camera are combined with the robot's self-localization to realize map construction. In the process of map construction, the depth image is first converted into a point cloud using Point_cloud [2], which publishes pointcloud2 messages, and then the point cloud is converted into a cost map using octomap [3] package and saved [4].

The specific process of obtaining the map is as follows:

1.Convert the depth map to a point cloud map. Add PointCloud2 in rviz.

2.Convert the point cloud map to a octomap. Add OccupancyMap in rviz.

3.Add Map in rviz.

4.Save the map. (rosrun map_server map_saver -f map1 map:=/projected_map)

5.Convert map to costmap for path planning and trajectory planning.

Figure 2 shows the static global map we got for the following path planning tasks.

Figure 2. Global cost map

2.4. Path and trajectory planner

The path planning and generation is mainly implemented using the ROS official package stdr_move_base [5].

Here, we use the depth camera as the sensor and the cost map obtained in 2.3 as the global cost map, use the global_planner and the local_planner to generate the global and local paths, respectively, and issue linear and angular velocity commands to the controller.

In this part of the work, we mainly adjusted a lot of parameters, and wrote the parameter configuration files, including the move_base_params.yaml, base_local_planner_params.yaml, global_costmap_params.yaml and etc., to make the package better adapt to this project.

After manually assigning a 2DNavGoal in rviz, the messages published by move_base can be checked using: “rostopic echo /stdr_move_base/TrajectoryPlannerROS/global_plan” and “rostopic echo /stdr_move_base/TrajectoryPlannerROS/local_plan”. The global and local target position and orientation are published respectively.

In Figure 3, the green line shows the global path given a target point.

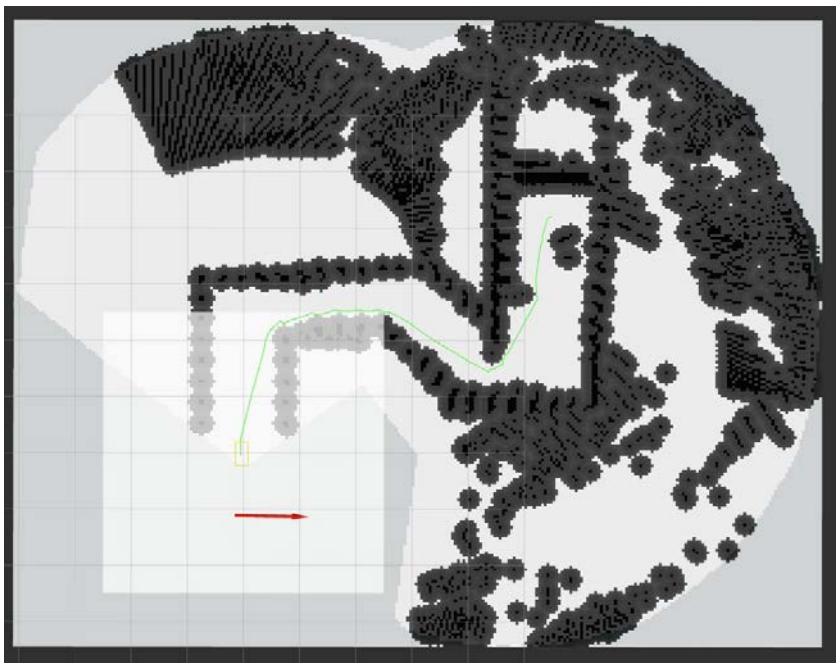

Figure 3. Generated global path

2.5. Controller

In the control part, we proposed and tested a variety of different schemes, such as subscribing to the position and orientation information and motion commands published by the local path planner, using position and speed control for the robot, or directly subscribing to the end position information published by the global path planner, and implementing the position control of the robot, etc.

Our final solution is to calculate the desired orientation based on the current position and the desired position and use a position controller to control the robot to achieve path following and finally hope to reach the specified target point.

The detailed implementation steps are as follows:

- All control codes are integrated in the file "src/controller_pkg/concontroller_node.cpp".
- Subscribe to the current position and orientation of the robot from "/current-state-est".
- Subscribe to the desired position of the robot from "/global-plan".
- Calculate the desired orientation based on the current position and the desired position.
- The forward control speed of the robot is set to the absolute value of the difference between the current position and the desired position.

In addition, we also designed different modes, so that the robot can automatically adjust the forward speed and steering according to the error of the current orientation and the target orientation under different conditions.

- Mode 1: When the difference between the current orientation and the desired orientation is greater than 32 degrees, the parameter "state_M" changes from 0 to 1, and

the steering mode is turned on. In this mode, the forward speed is 0.1 and the steering speed is 4.8.

- Adaptive Mode: When the difference between the current orientation and the desired orientation is less than 32 degrees, different steering speeds will be generated according to the size of the error angle.
- Mode 2: When the robot has no position change for a long time, it means that the robot reaches the first obstacle, the parameter "state_M2" changes from 0 to 1, and starts to "cross the obstacle" mode.

3. Result and summary

In this project, although we failed to complete the task on time and achieve the initial goal, we learned and practiced a lot of practical knowledge of ROS, and also initially mastered the ability to use git for distributed development.

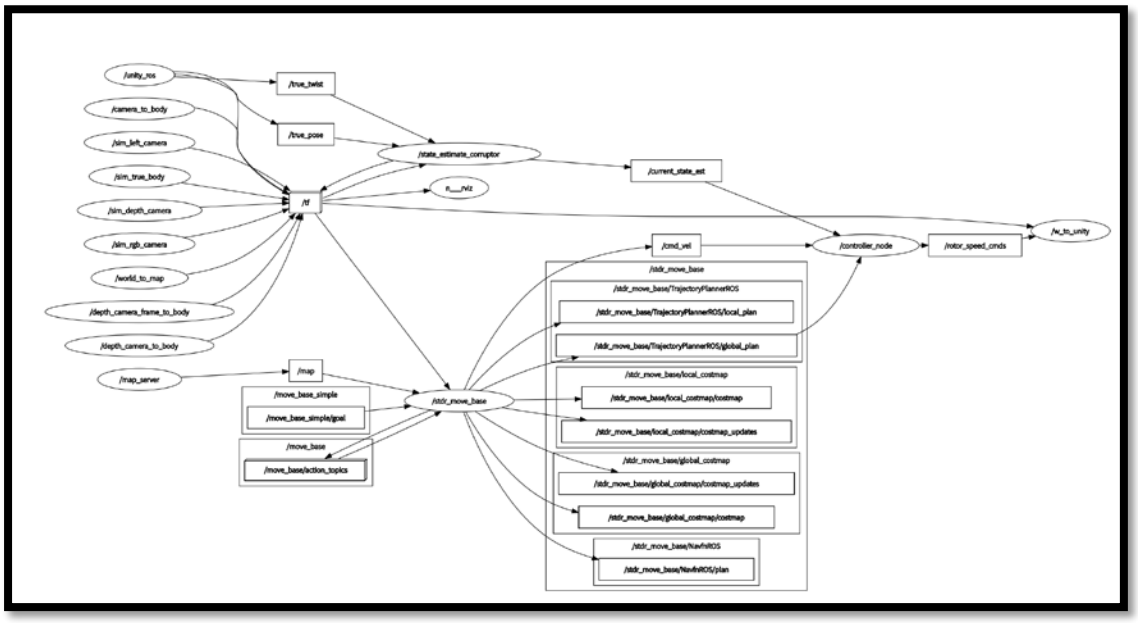

Figure 4. rqt_graph

3.1. Achievements

The path provided by the "move_base" package is the optimal path. The path at the corner is too close to the corner. If the robot moves completely along the path, it will cause the robot to collide with the road post at the corner. Our code will only fully switch to steering mode when the error angle is greater than 32 degrees, and adopt adaptive mode when it is less than 32 degrees, which allows the robot to autonomously choose a non-optimal but safer route.

3.2. Problems and analysis

However, our current version of the code still has certain problems, which cause the robot to not successfully reach the target point. We tried many different sets of control parameter combinations, all without success over the first obstacle. In the current situation, our robot's front legs can climb the obstacle, but the hind legs can't, so it can't reach the goal.

The possible reasons could be that, the map construction of the final obstacle (the slopes) during the perception part is incomplete, or the control part has a problem with the instruction of the robot to raise its legs (the implementation of “amplitude”), or the path planner provided by move_base package only considers a 2D path and cannot generate path points on the third dimension.

4. Literature and references

[1]. teleop_twist_keyboard: http://wiki.ros.org/teleop_twist_keyboard

[2]. depth_image_proc: http://wiki.ros.org/depth_image_proc

[3]. octomap: http://wiki.ros.org/octomap

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

[5]. move_base: http://wiki.ros.org/move_base