Autonomous Quadruped

– Group 22

Yankun Wei 03768261
Hao Guo 03768270
Hanyu Li 03766800
Shaoxiang Tan 03749103
Yujie Guo 03768329

1. Description of ROS node and package

ROS Package	Functionality
depth_image_proc*	Generate point cloud from depth image.
point_cloud_filter	Filter out the ground points of the point cloud.
octomap*	Convert filtered point cloud to occupancy grid.
move_base*	Generate global and local costmap, configure the local and global path planner.
planning	Get path plan from move_base, choose suitable path point, detect steps and publish path point.
controller_pkg	Change state according to current pose and pathpoint, publish corespond commands to unity in different states, publish trajectory and path points.

Package with * mark is reused code.

2. ROS graph

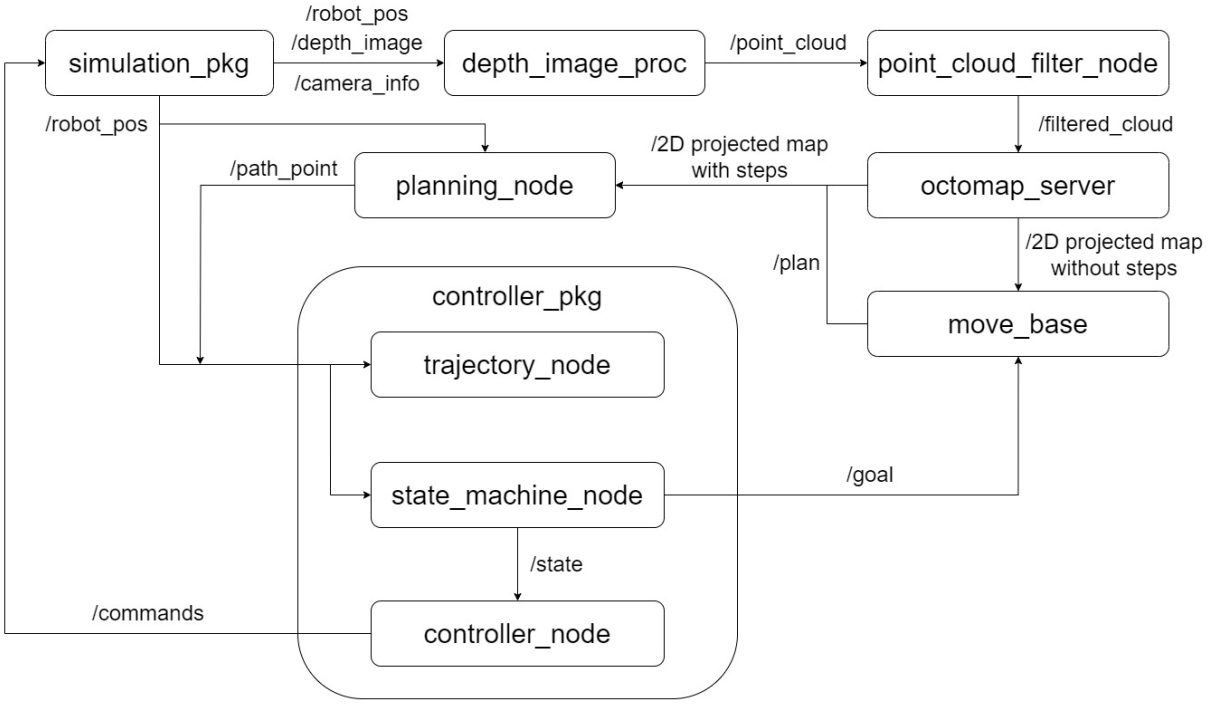

Fig.1 General pipeline

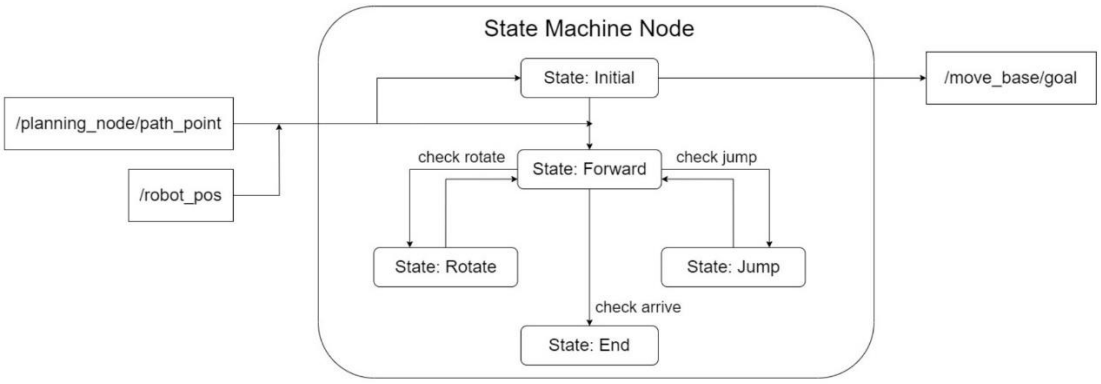

Fig.2 Structure of state_machine_node

3. Implementation Details

3.1 Perception pipeline

Here we convert the depth image, first to a point cloud and second to an occupancy grid. The launch file of perception pipeline is located at /src/simulation/launch/ simulation_rviz. launch.

3.1.1 Point cloud from depth image

We use depth_image_proc package as recommended in group project documentation to convert the depth information into point cloud.[1]

- Use nodelet to run this node
- Remap the topic "camera_info" to topic "/realsense/depth/camera_info"/ to subscribe the depth information of the image.
- Remap topic "points" to topic "/depthcamera/points" which will publish the point cloud information.

3.1.2 Filtered point cloud

We use SACSegmentation algorithm of pcl(Point Cloud Library) in node point_cloud_filter_node to filter out the ground points in point cloud from detph image, which will have bad effects on navigation due to camera shake during motion. The filtered point cloud is published by topic "/filtered_cloud".

3.1.3 Occupancy grid

We use Octomap and the node octomap_server as suggested in in http://wiki.ros.org/octomap to convert the point cloud into two occupancy grids.[2] One without steps is used for 2D navigation, another one with steps is used for steps detecting.

- Subscribe the point cloud from the topic "/filtered_cloud".
- Set the resolution of the voxel to 0.12/0.08.

- Adjust the range of the z axis to filter out the steps voxel which is useless in our 2d navigation.

- publish topics including information about occupancy grid and the projected map in "/octomap_binary", "/octomap_full" and "projected_map". The result can be visualized with the help of the octomap rviz plugins in rviz.

3.2 Path planning

Here we will use move_base package to generate a local costmap and global costmap from which the optimal trajectory will be selected. After knowing the best trajectory, we can choose the control that will allow us to travel the designated path as quickly as possible. The configurations of parameters are located at /src/simulation/param.

3.2.1 Costmap configuration files for move_base [3]

- costmap_common_params. yaml

It contains parameters for both global and local maps. In this file we mainly config footprint of the robot which defines the outline of the robot base (robot size) and the properties of used sensor (frame, data type and topic).

- global_costmap_params.yaml

We mainly define the update_frequency and publish_frequency of the global map.

- local_costmap_params.yaml

We mainly define the update_frequency and publish_frequency of the local map.

- global_planner_params.yaml

- base_local_planner_params.yaml

Once the goal is set manually, node move_base performs path planning and publishes a series of path points to the topic /plan, which is subscribed by planning_node.

3.2.2 planning_node

The planning_node subscribe to three topics with correspond callback functions:

- current_state_est

Published by state_estimate_corruptor_node, contains information of current pose of the quadruped. In callback function planning_node update current pose.

- /move_base_Quadruped/NavfnROS/plan

Published by node move_base, contains a series of planned path points. The planning_node iterates over these path points in the callback function and selects one path point, which is a pre_defined distance away from the current position of the quadruped and also in front of it, as the target path point.

- projected_map_step

2D occupancy grid with steps published by node octomap_server. In callback function planning_node check if the target path point is on a step by checking the proportion of occupied grids around the target path point. If so, the z coordinate of the target path point will be set as 2, otherwise 0.

The planning_node updates current pose and target path points in real time with the callback function above. There is also a timer publishing the target path point at a fixed frequency to topic "/path_point".

3.3 controller_pkg

The controller_pkg consist of three nodes:

- state_machine_node

State machine controls the workflow of the quadruped, subscribing to "/current_state_est" and "/path_point".

Starting from state INITIAL, state_machine_node publish coordinate of the goal to move_base in topic "/move_base_simple/goal".

After receiving first path point, state_machine_node change to one of the following states: FORWARD, Clockwise_rotate and counterclockwise_rotate using function check_rotation, which determines whether rotation is needed by checking the angle between the quadruped's current orientation and the target path point. The state_machine_node continues to use function check_rotation in three states above, which constitutes a simple trajectory planning, i.e. rotating towards the target path point before moving forward.

In three states above, state_machine_node also check the z coordinate of the path_point, if above 0 (set by planning_node), change to state JUMP to pass the steps.

State_machine_node change to state END when current position from "/current_state_est" is close to goal. In each state state_machine_node published correspond message to controller_node.

- controller_node

Subscribing to "/state" from state_machine_node, controller_node adjusts the actuators message according to different states and publish commands to simulation environment.

- trajectory_node

Subscribing to "/current_state_est" and "/path_point", trajectory_node visualizes the trajectory of the quadruped and the target path points over time on rviz.

3.4 Implementation of message type state_indicator_msgs

In fact, in our code, there is no specific need for a newly defined message type. All the required tasks can be accomplished using existing data types. However, in order to fulfill the requested task, we have defined a message type called state_indicator_msgs under the /mav_comm package. This message type consists of a basic int32 data type named state_msg. If there is a need for additional complex types in the future, they can be added to this

message. Currently, we only require this one data type to represent the state.

- publish state topic

Use state_pub = nh.advertise<state_indicator_msgs::state_indicator>("state",1) in the state_machine_node.cpp to publish the state message.

- subscribe state topic

Use state_sub = nh.subscribe("state", 1, &controllerNode::onState, this); in controller_node to subscribe the topic to get the message we defined before.

3.5 Experiment with different commands

Before using path planning, we need five different combinations of input data for testing to enable the quadruped to perform different actions like walking, climbing, rotating, accelerating and jumping. To achieve this, we have implemented a for loop to test the quadruped 's state under various inputs.

The respective experiment result is show in below:

Turn clockwise: [0, 45, 0, 0, 7]

Turn counterclockwise: [0, -45, 0, 0, 7]

Forward: [0, 90, 0, 0, 8]

Jump: [90, 0, 0, 20, 0]

Quick forward: [45, 0, 30, 20, 7]

Climb over obstacles and climb down: [45, 0, 40, 20, 7]

The test video is available on Github.

4. Result visualization

Following are current results can be obtained from the simulation:

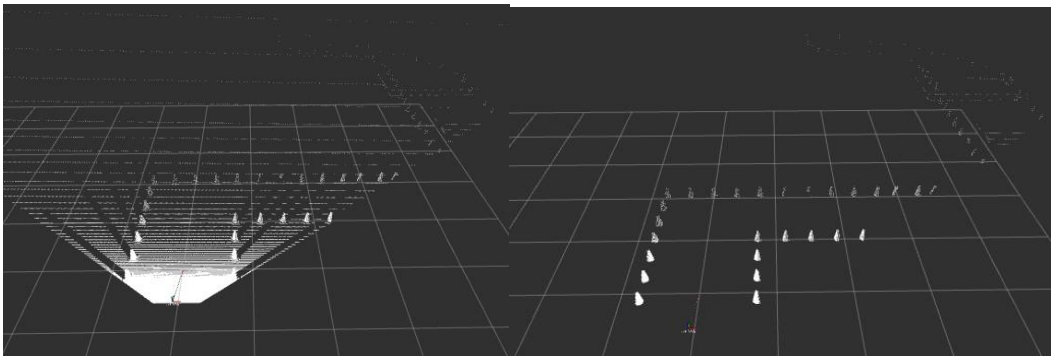

Fig.3 point cloud from depth image(left) and filtered point cloud(right)

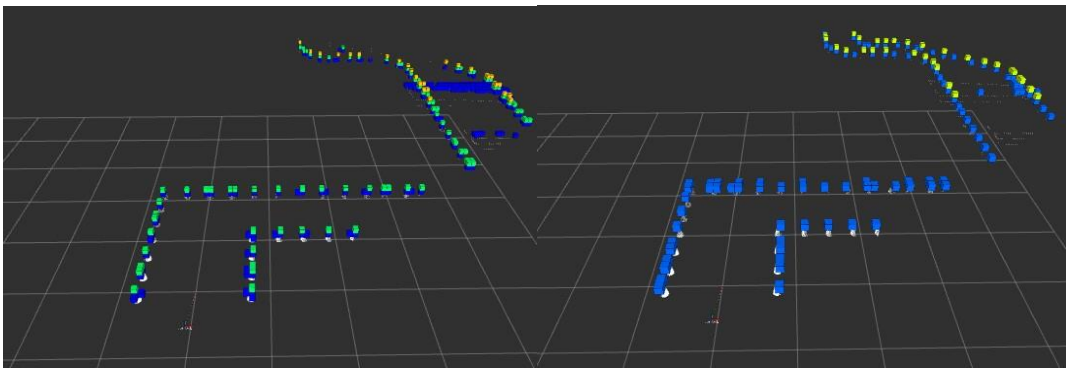

Fig.4 3D occupancy grid with(left) and without(right) steps

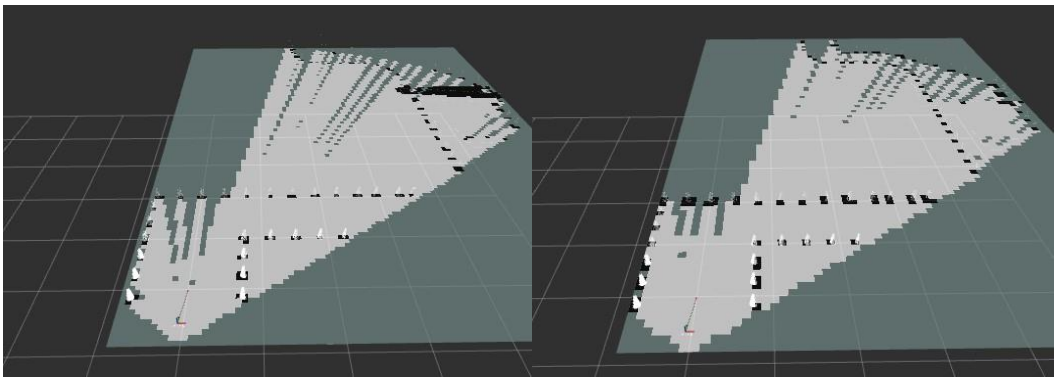

Fig. 5 2D occupancy grid with(left) and without(right) steps

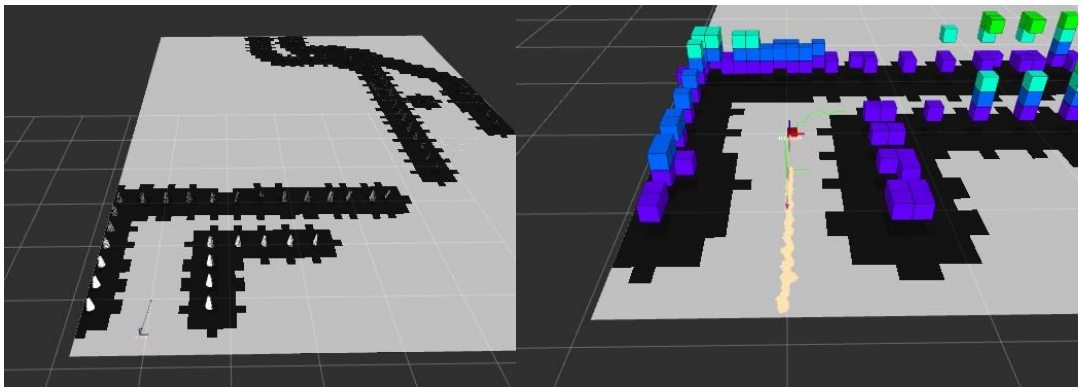

Fig.6 move_base costmap without steps(left).

Fig.7 path planning by move_base green: path plan, red: current target path point, yellow: trajectory(right)

5. Remaining problems

Due to time limitations, there are still some problems that have not been solved by us through adjustments, which has resulted in us not being able to successfully pass the parkour yet.

5.1 Inappropriate motion parameters

Except for the parameters given by default for moving forward, appropriate parameters in the rest of the states are not found. This results in some unintended movements of the quadruped, such as stepping in place or turning very slowly.

5.2 Inappropriate direction judgment

We define the angle between the current orientation of the quadruped and the vector between its current position and the target point as the measure for determining whether the quadruped should rotate or move forward. It will only move forward when the cosine value of this angle is greater than 0.87; otherwise, based on the relative relationship between the two vectors' directions, the quadruped will decide whether to rotate clockwise or counterclockwise. Some unrecognized bugs make the quadruped's judgment of direction very unstable, which makes it often get stuck on the first corner.

5.3 Inappropriate configuration

In simulation, the quadruped may ignore obstacles and cross them sometimes. It can also be observed that the latter part of the path will be blocked even if we are using occupancy grid without steps for path planning. Possible reason is inappropriate move_base configuration such as inflation_radius and octomap_server configuration such as resolution.

5.4 program only be run in steps

Merging two launch files causes a planning bug that we cannot fix due to time limits. To be able to run planning properly, the program can only be run in two steps:

Step1: Start the simulation environment, perception pipeline and the visualization tool rviz.

Step2: Run planning node and controller_pkg.

Bibliography

[1] convert depth information to occupancy grid http://wiki.ros.org/depth_image_proc/
[2] convert point cloud to occupancy grid http://wiki.ros.org/octomap/
[3] configuration of the move base: https://husarion.com/tutorials/ros-tutorials/8-path-planning/