ROS-Autonomous Quadruped-Team 12

Yue Zhang, Mengfei Fan, Xinlong Wang, Guangyan Wu

July 2024

Contents

1 Introduction

The objective of this project is to enable the quadruped robot to navigate along a predefined route, surmounting obstacles, and ultimately reaching the target destination.

Chapter 2 offers an in-depth explanation of the ROS packages employed in this project and their corresponding functionalities. In Chapter 3, we discuss the implementation logic and content of both offline and online methods, followed by a comparison of their advantages and disadvantages. This project is divided into three key segments: perception, path planning, and control, with comprehensive algorithms. Chapter 4 presents our conclusions and future prospects, summarizing our findings and discussing potential improvements. Lastly, Chapter 5 provides a list of the references utilized throughout this project.

2 ROS Information

2.1 ROS Packages

ROS Node	ROS Package	Functionality
Shared Nodes and Packages		
nodelet_manager	nodelet	efficient data sharing and processing
point_cloud_xyz_nodelet	depth_image_proc	converts depth images into point clouds
octomap_server	octomap_server	generates an octree-based map from the input point cloud
map_saver	map_server	save a 2D occupancy grid map
map_server	map_server	load a pre-defined 2D occupancy grid map from a file
Offline Planning and Control		
planning_offline_node	planning	generate an offline path
trajectory_planner	trajectory_planning	publish the trajectory for offline control
state_machine_offline_node	controller_pkg	a state machine in offline control
controller_node_PID	controller_pkg	a PID controller for the speed of robot
Online Planning and Control		
move_base	move_base	local and global path planning in online planning
planning_online_node	planning	add waypoints to increase the accuracy in online planning
trajectory_publisher_node	planning	generate an online path
state_machine_online_node	controller_pkg	a state machine in online control
controller_node	controller_pkg	a normal controller for robot control

Table 1: ROS Nodes and their functionalities

2.2 ROS Graph

We have implemented both offline path planning and online path planning, along with their corresponding control modules. Below is the relationship between each node.

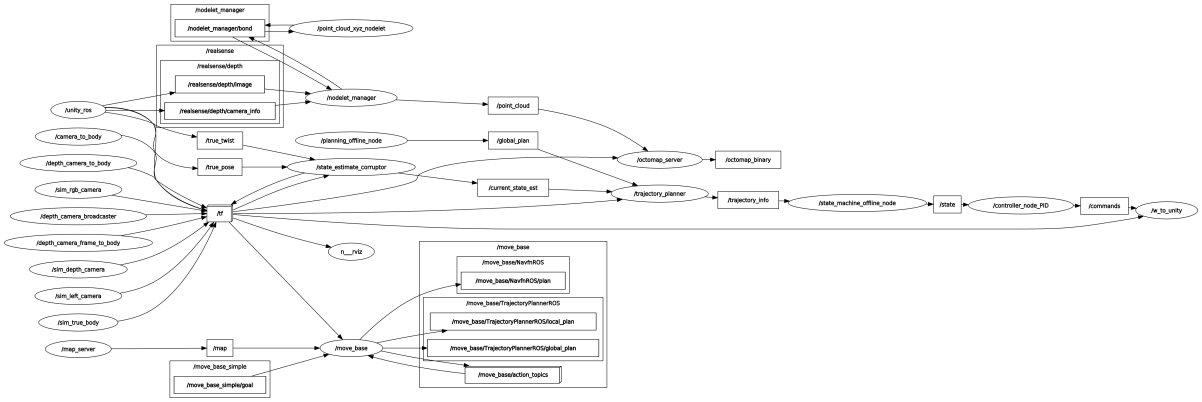

Figure 1: offline_rosgraph

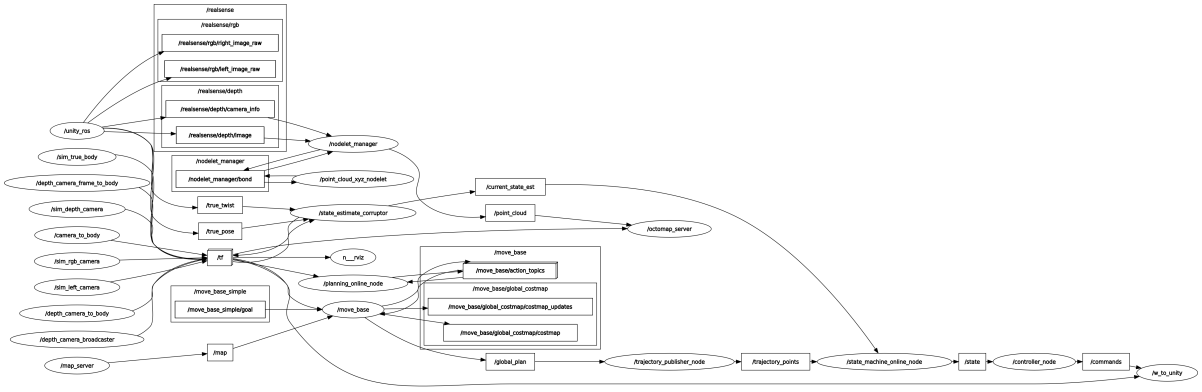

Figure 2: online_rosgraph

The offline and online modes use the same perception module and similar control modules, so their structures are similar.

3 Algorithms

3.1 Perception

3.1.1 Perception Pipeline

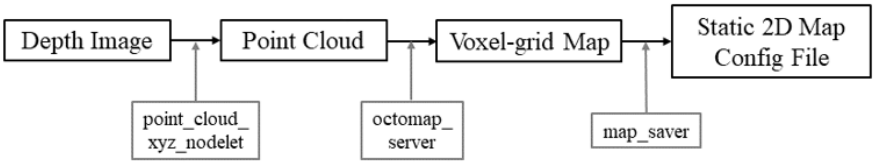

Figure 3: Perception Pipeline

From the simulator, by subscribing to the topic `/realsense/depth/image`, we can obtain the required Depth Image. Then, by using the package `depth_image_proc` and remapping the topic names, we can obtain the Point Cloud information and the topic is `/point_cloud`. Following this, we utilize the `octomap_server` package to generate a 2D occupancy map, remapping the topic name to `/local_map` with the type `nav_msgs/OccupancyGrid`, allowing us to obtain the local map at each timestep. To acquire a global static map, we use the `map_server`, subscribing to the topic `map (nav_msgs/OccupancyGrid)`. The global static map is stored in the form of `filename.pgm` and `filename.yaml` for later use.

3.1.2 Point Cloud and Voxel-grid Map

As mentioned in Chapter 3.1.1, we call `depth_image_proc` package to obtain the topic `/point_cloud`.

`depth_image_proc` is a package in ROS that processes depth images to produce useful data formats for further analysis and processing. Specifically, it takes depth images from cameras like the Intel RealSense and converts them into point cloud data, which can be used for 3D mapping, object recognition, and navigation tasks. This conversion is achieved by transforming the depth data into 3D coordinates, resulting in the topic `sensor_msgs/PointCloud2`.

```
1 <node pkg="nodelet" type="nodelet" name="point_cloud_xyz_nodelet" args="load depth_image_proc/
      point_cloud_xyz nodelet_manager" output="screen">
2     <remap from="image_rect" to="/realsense/depth/image"/>
3     <remap from="camera_info" to="/realsense/depth/camera_info"/>
4     <remap from="points" to="/point_cloud"/>
5 </node>
```

The effect of the point cloud is shown in the Figure 4.

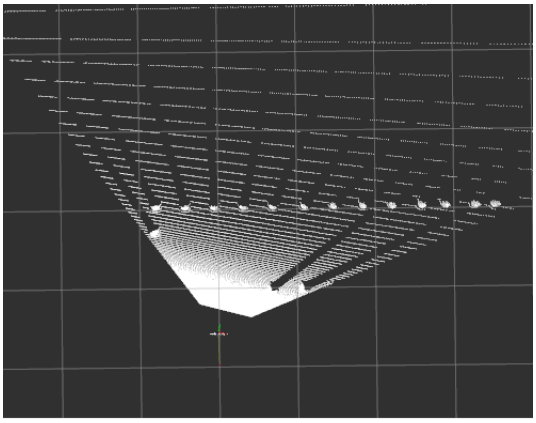

Figure 4: Point Cloud

3.1.3 Global Static Map

To obtain the global static map, we need to use the package `map_server` and run the following command:

```
rosrun map_server map_saver -f <filename>
```

In order to get a more accurate global static map, we first wrote a keyboard control node for getting the global map.

The resulting global static map is shown in the Figure 5 .

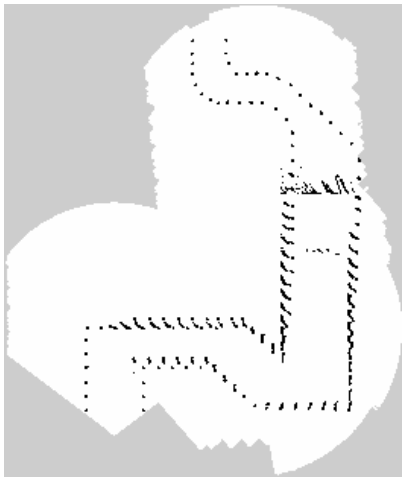

Figure 5: Global Static Map

3.2 Planning

3.2.1 Receiving Global Map

To successfully perform path planning, we first need to load the global map. Here, we use the `map server` package to load the global static map obtained in the perception module. The following code is used:

```
<node pkg="map_server" type="map_server" name="map_server" args="$(find perception)/global_map/
    modified_global_map.yaml">
    <param name="frame_id" value="world"/>
</node>
```

3.2.2 Offline Planning

globe path generation For the planning in this project, we developed a custom path planning algorithm. Our approach utilizes predefined waypoints based on the map generated from the depth camera's point cloud data. These reliable waypoints form a continuous path, which is then published to a designated ROS topic: **/global_plan**. This method ensures that the vehicle navigates along a specified route by continuously following the published path. Below is a detailed description of our planning algorithm and its implementation. **Published Topic**: /global_plan

- **Type**: nav_msgs
- **Function**: guides the robot through a predefined route by continuously publishing waypoints.

Waypoints Definition: A series of waypoints are predefined in a vector, where each waypoint includes positional coordinates (x, y, z) and orientation in quaternion form (qx, qy, qz, qw).

Waypoint	X	Y	Z	Roll	Pitch	Yaw
Waypoint1	0	0	0	0	0	1
Waypoint2	0	1.75	0	0	0	1
Waypoint3	1	1.65	0	0	0	1
Waypoint4	3	1.65	0	0	0	1
Waypoint5	4.4	0.3	0	0	0	1
Waypoint6	5.5	1.3	0	0	0	1
Waypoint7	5.5	2.65	0	0	0	1
Waypoint8	5.5	4.65	0.3	0	0	1
Waypoint9	5.5	5.0	0	0	0	1
Waypoint10	5.5	6.5	0.32	0	0	1
Waypoint11	5.5	7.2	0	0	0	1
Waypoint12	4.70	8.14	0	0	0	1
Waypoint13	4.75	8.06	0	0	0	1
Waypoint14	4.29	8.5	0	0	0	1
Waypoint15	3.55	8.67	0	0	0	1
Waypoint16	2.68	8.33	0	0	0	1
Waypoint17	2.93	8.64	0	0	0	1
Waypoint18	2.5	10	0	0	0	1

Table 2: Waypoints Data in Offline Planning

Path Publishing: The algorithm initializes a ROS node and sets up a publisher for the /global_plan topic.Path message is constructed using the predefined waypoints. The frame and timestamp are set appropriately to ensure consistency.Each waypoint is converted into a geometry_msgs message, which is then added to the path message.o The constructed path is published to the /global_plan topic at regular intervals. **Continuous Path Publishing**: The main loop of the algorithm continuously publishes the global path at a fixed rate (e.g., 1 Hz). This ensures that the robot always has an up-to-date path to follow, facilitating smooth and continuous navigation. The generated global path is shown in the Figure 6.

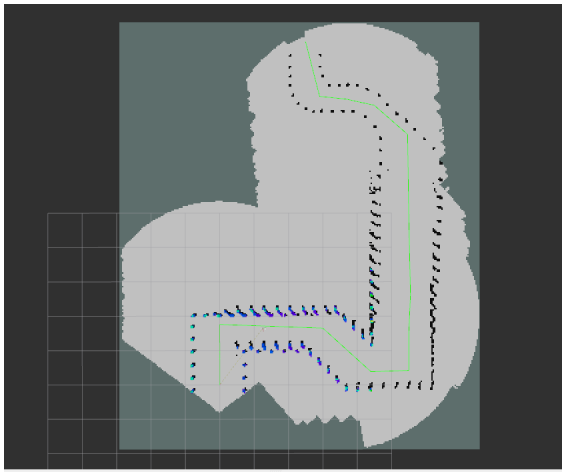

Figure 6: Offline Path (green path)

trajectory point generation *Offline Control Based on Trajectory Point Publish*

Offline control using trajectory point publishing is a method aimed at validating the entire control process of the quadruped robot in a controlled environment. This approach focuses on generating trajectory points offline and publishing them, providing a simplified and computationally efficient way to verify the robot's control mechanism.

1. **Motivation for Offline Control and Choice of Trajectory Point Publishing**
   - *Global Path Planning*: We obtained the global path through planning and identified key points (waypoints) along this path to serve as the basis for offline trajectory point generation.
   - *Static Environment*: In our project map, there are no random obstacles. Hence, the lack of real-time dynamic path planning does not affect the outcome. Offline trajectory point generation, with its reduced computational load and simpler logic, can quickly validate the entire control process of the quadruped robot.
   - *Benchmark for Comparison*: This approach provides benchmark data to compare with online trajectory point generation and online control, facilitating the analysis of the pros and cons of both methods.
2. **Principle of Offline Trajectory Point Generation**

   In the x, y plane, based on the current point coordinates, the robot moves a fixed step length (0.1) towards the target waypoint. When the distance between the current point and the waypoint is less than a threshold value (0.25), the waypoint updates to the next target.

   - *Receiving Points (Subscribe)*:

```
path_sub_ = nh_.subscribe("/global_plan", 1, &TrajectoryPlanner::pathCallback, this);
current_pose_sub_ = nh_.subscribe("/current_state_est", 1, &TrajectoryPlanner::
    currentPoseCallback, this);
```

     These subscriptions receive the current timestep's corresponding waypoint and current point.

   - *Publishing Target Points (Advertise)*:

```
traj_pub_ = nh_.advertise<trajectory_msgs::MultiDOFJointTrajectoryPoint>("/trajectory_point",
    1);
```

     Methods include:
     - **moveTowards**: Move a step towards the waypoint.
     - **convertToPose**: Update the waypoint.
     - **calculateDistance**: Determine when to update the waypoint.
   - *Publishing New Message*:

     To facilitate offline control, we defined a new message format in the trajectory part and published the corresponding message, ensuring synchronized timestamps.

```
// Publishing TrajectoryInfo message
trajectory_planning::TrajectoryInfo traj_info;
traj_info.current_pose = current_pose_;
traj_info.generated_trajectory_point = new_target_pose;
traj_info.waypoint_pose = target_pose;
traj_info_pub_.publish(traj_info);
```

TrajectoryInfo.msg:

```
geometry_msgs/Pose current_pose
geometry_msgs/Pose generated_trajectory_point
geometry_msgs/Pose waypoint_pose
```

3.2.3 Online Planning

globe path generation This node is designed to control a robot by sending it a series of waypoints to follow. The node communicates with the move_base action server to send goal positions and orientations. It uses a TransformListener in TF to verify if the robot has reached each waypoint within a specified tolerance. The main functionalities include defining waypoints, sending goals to the **move_base** action server, and monitoring the robot's progress.

Although we successfully completed the entire track using offline planning, the large number of waypoints required presents a high demand on manual input and incurs significant labor costs during the planning process. To address this, we use a global planner for path planning. In practice, to ensure smooth paths and facilitate obstacle recognition, we still select certain waypoints as the targets the robot needs to reach. However, the number of waypoints is significantly reduced compared to the offline approach. The waypoints can be found in the table 3.

Waypoint	X	Y	Z	Roll	Pitch	Yaw
Waypoint1	0	0.60	0	0	0	1
Waypoint2	5.62	0.40	0	0	0	1
Waypoint3	5.62	3.60	0	0	0	1
Waypoint4	5.6	7.42	0	0	0	1
Waypoint5	2.8	9.00	0	0	0	1
Waypoint6	2.5	10.00	0	0	0	1

Table 3: Waypoints Data in Online Planning

During the planning process, we use the **move_base** package for global path planning. Since **move_base** does not support the addition of via points and can only plan paths between two points, we implement global navigation by sequentially publishing waypoints. When the robot reaches a specific waypoint, the next waypoint is published, enabling real-time path planning.

Detailed Steps

1. **Initialization and Setup**
   - *Node*: **planning_online_node**
   - *move_base Configuration*: Utilizes point cloud data from a camera, subscribing to the **/point_cloud** and **/map** topics.global_frame is set to **world**, and robot_base_frame is set to **true_body**.Uses NavfnROS for global path planning and uses dwa_local_planner for local path planning, and the costmap for online planning is Figure 7.
   - *Action Client*:An action client is created to communicate with the move_base action server.
     Uses NavfnROS for global path planning and uses dwa_local_planner for local path planning.

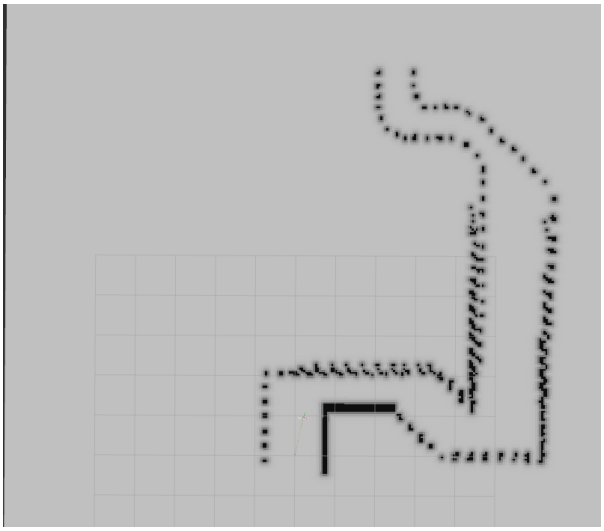

Figure 7: Global Costmap for online planning

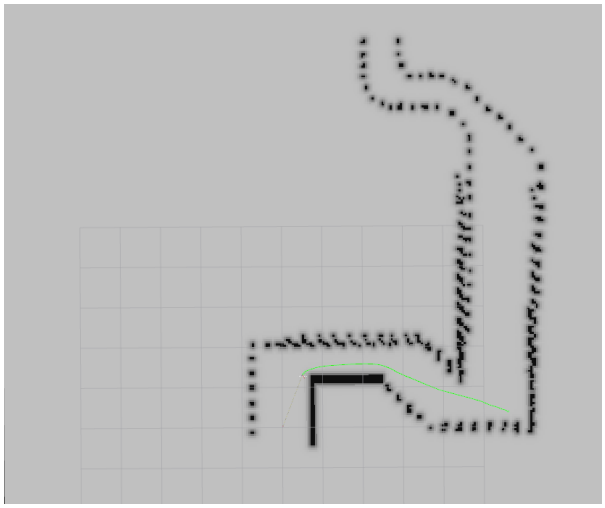

Figure 8: Online Path (green line)

2. **Defining and Sending Waypoints**: A list of waypoints is defined, each containing position and orientation data. The node sends these waypoints as goals to the move_base action server one by one. For each waypoint, a **move_base_msgs::MoveBaseGoal** message is populated and sent to the move_base action server. The goal's reference frame is set, and the current timestamp is included.

3. **Monitoring and Verifying Goal Achievement**: The current position of the robot is obtained through the Transform Listener, comparing it to the target waypoint.After the Position Check, the distance between the robot's current position and the target waypoint is calculated.If the robot is within a specified tolerance (e.g., 0.18 meters) from the waypoint, it is considered to have reached the goal.

4. **Handling Goal Transitions**: Once the robot reaches a waypoint, the node proceeds to the next waypoint in the list. This process repeats until all waypoints are processed.

trajectory point generation *Online Control Based on Trajectory Point Publish*

In the online trajectory point generation process, we iterate through the path generated in the **planning_node**, using points 0.45 meters ahead on this path as trajectory points. These points are then used in the online control section.

1. **Principle of Online Trajectory Point Generation**

   - *Receiving Points (Subscribe)*:

```
// Subscribe to the global path ros topic
path_sub_ = nh.subscribe("/global_plan", 10, &TrajectoryPublisher::pathCallback, this);
```

   - *Iterate through the entire path and publish the trajectory points*:

```
void pathCallback(const nav_msgs::Path::ConstPtr& path_msg)
{
    // Check if the path is empty
    if (path_msg->poses.empty()) return;

    // Initialize distance traveled
    double distance_traveled = 0.0;
    geometry_msgs::Point last_point = path_msg->poses[0].pose.position;

    // Iterate through the path
    for (size_t i = 1; i < path_msg->poses.size(); ++i)
    {
        geometry_msgs::Point current_point = path_msg->poses[i].pose.position;
        double distance = distanceBetweenPointsXY(last_point, current_point);
        distance_traveled += distance;
```

```
16
17          if (distance_traveled >= lookahead_distance_)
18          {
19              // When the desired distance is reached, publish this point.
20              traj_pub_.publish(path_msg->poses[i]);
21              return;
22          }
23
24          last_point = current_point;
25      }
26
27      // If the lookahead distance is not reached, publish the last point in the path
28      traj_pub_.publish(path_msg->poses.back());
29  }
```

These subscriptions receive the current timestep's corresponding waypoint and current point.

- *Publishing Target Points (Advertise)*:

```
1  // Publish the trajectory points
2  traj_pub_ = nh.advertise<geometry_msgs::PoseStamped>("/trajectory_points", 10);
```

3.3 Control

3.3.1 Offline Control

state machine *Offline Control Based on six states define*

we defined six states: STOP (0), STRAIGHT (1), LEFT_TURN (2), RIGHT_TURN (3), UPHILL (4), and UPSTAIR (5). The state transitions are determined based on the angle between the target direction and the current direction, the z-coordinate of the waypoint, and the distance to the endpoint. These transitions help the robot navigate to the target position effectively.

1. **State Machine Principle: Conditions for State Transitions**

   - *Angle Calculation (calculateSinTheta)*: The angle between the target direction and the current direction is calculated. The target direction is the vector from the current point to the trajectory point. The current direction is derived from the quaternion of the coordinates. The calculation is as follows:

```
1      sin = (-current_direction(0) * goal_direction(1) + current_direction(1) * goal_direction
           (0)) /
2            (current_direction.norm() * goal_direction.norm());
```

   - *Distance Calculation (calculateDeltaS)*: The distance from the current point to the endpoint is computed.
   - **Waypoint z-coordinate (waypoint_pose.position.z)**:
     The z-coordinate of the waypoint is used as a condition for state transitions.

2. **State Transitions (Refer to the provided state machine diagram)**: The robot transitions between states such as STRAIGHT_FAST, LEFT_TURN, RIGHT_TURN, UPHILL, and UPSTAIR based on the calculated sine value, distance to the endpoint, and the z-coordinate of the waypoint.

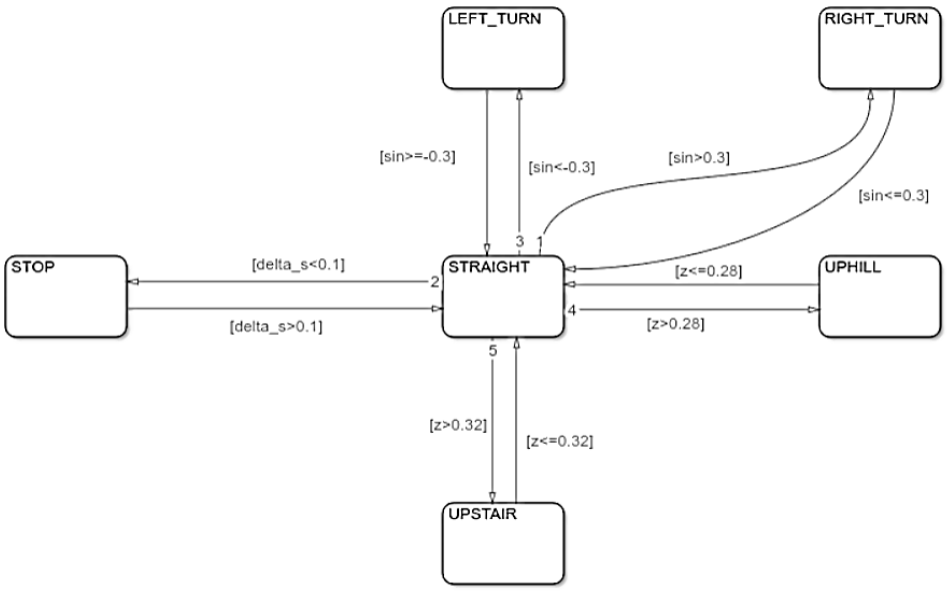

Figure 9: State machine diagram

controller The goal of the controller module is to develop a robot controller node that can send appropriate control commands based on different states. The controller node subscribes to messages published by the state machine to obtain the next target state and publishes control commands based on these states. **Node: Controller_node Function**: Control the robot to perform corresponding actions based on different states. **Published Topic**: commands

- **Type**: mav_msgs/Actuators
- **Function**: Publish commands to control the robot's actions.

```
commands_pub_ = nh_.advertise<mav_msgs::Actuators>("commands", 1);

mav_msgs::Actuators msg;
msg.angular_velocities.resize(control_inputs.size());
for (size_t i = 0; i < control_inputs.size(); ++i) {
    msg.angular_velocities[i] = control_inputs[i];
    }
commands_pub_.publish(msg);
```

Subscribed Topic: state

- **Type**: state_indicator_msgs/state_indicator
- **Function**: Receive the current state indication message of the robot.

```
state_sub_ = nh_.subscribe("state", 1, &ControllerNode::stateCallback, this);
```

Internal Timer: Calls controlLoop method at a frequency of hz_ times per second.

```
timer_ = nh_.createTimer(ros::Rate(hz_), &ControllerNode::controlLoop, this);
```

Control Input Description From the provided simulation, there are five available control inputs:

1. Phase between front and back legs
2. Phase between front left and back right legs and front right and left back legs
3. Amplitude change of all legs

4. Amplitude change of back legs
5. Frequency of legs

Different Gaits and Corresponding Values

State	Action	Control Input Values
0 (STOP)	Stop	{0, 0, 0, 0, 0}
1 (STRAIGHT)	Straight	{0, 90, 0, 0, 10}
2 (LEFT_TURN)	Left Turn	{0, -45, 0, 0, 7}
3 (RIGHT_TURN)	Right Turn	{0, 45, 0, 0, 7}
4 (UPHILL)	Uphill	{0, 0, 4.5, 32, 9.5}
5 (UPSTAIR)	Upstairs	{0, 0, 4, 35, 8}

Table 4: State and Action Control Input Values

3.3.2 Online Control

state machine *Online Control Based on six states define*

We defined six states. The state transitions are determined based on the angle between the target direction and the current direction, jump range condition and the distance to the endpoint. These transitions help the robot navigate to the target position effectively.

1. **State Machine Principle: Conditions for State Transitions**
   - *Angle Calculation (calculateSinTheta)*: same as offline control state machine.
   - *Distance Calculation (calculateDeltaS)*: same as offline control state machine.
   - **Jump Range Condition**: Based on the global path, obstacles are identified. When the robot's current position is within the obstacle range and the angle between the current direction and the target direction is less than a threshold value, the jump state is triggered.
2. The rest of the state transition logic follows the principles outlined in the offline control state machine section.

controller Same as online controller

3.3.3 PID Controller

Using PID controllers to achieve control of the robot's turning and speed, adjusting the system's errors to achieve more precise control. The code defines two PID controllers:

```
// PID controller class definition
class PID {
public:
    PID(double kp, double ki, double kd)
        : kp_(kp), ki_(ki), kd_(kd), prev_error_(0), integral_(0) {}

    double compute(double setpoint, double pv) {
        double error = setpoint - pv;
        integral_ += error;
        double derivative = error - prev_error_;
        prev_error_ = error;
        return kp_ * error + ki_ * integral_ + kd_ * derivative;
    }

private:
    double kp_, ki_, kd_;
    double prev_error_;
    double integral_;
};
```

- pid_turn_ for controlling turning speed, with parameters kp=1.0, ki=0.0, kd=0.1.

- pid_speed_ for controlling driving speed, with parameters kp=0.3, ki=0.1, kd=0.0.

The compute function adjusts the robot's turning and driving speed by calculating the current error. Specific control rules are as follows:

- **Turn Control**: By calculating the angular error (sin_theta) between the current direction and the target direction, the larger the directional error, the greater the turning speed.

```
straight_speed = std::max(0.0, 10.0 + pid_speed_.compute(0.0, delta_s));
```

- **Speed Control**: By calculating the distance (delta_s) between the current position and the target position, the closer to the target, the slower the speed.

```
turn_speed = std::max(0.0, pid_turn_.compute(0.0, sin\_theta));
```

These control rules ensure that the robot can smoothly and accurately follow the predetermined path.

4 Conclusion and Outlook

In summary, we successfully converted camera images into depth images, point cloud data, and voxel-grids in the perception module. Based on the perception module, we implemented two planning and control schemes: offline planning and control and online planning and control, and compared their simulation results. The results of the two were compared (mainly in terms of completion time, stability, and obstacle avoidance performance).

Offline Control: This approach demonstrated precise and collision-free control but is limited to static environments with minimal obstacles. The test run was completed in 7 minutes and 5 seconds with zero collisions.

Online Control: This method allowed for more autonomous control by planning real-time paths at each timestep based on the current position, making it suitable for dynamic or more complex scenarios. The test run was completed in 7 minutes and 18 seconds with two minor collisions.

While the advantages of online control are evident, such as its adaptability to changing environments, our implementation encountered challenges. The real-time path planning in online control is influenced by several factors, including the accuracy of the costmap and obstacle detection. Consequently, the performance of our online control code was not as precise as the offline control. Moving forward, refining these aspects will be crucial to enhancing the accuracy and reliability of online control.

By addressing these challenges, we aim to improve the robustness and efficiency of online control systems, ultimately enabling more versatile and reliable autonomous navigation for quadruped robots in dynamic and complex environments.

5 Team Members

Components	**Members**
Perception & Map	Yue Zhang
Globe Path Generation	Guangyan Wu
Offline Trajectory Generation & State Machine	Mengfei Fan
Online Trajectory Generation	Yue Zhang
Controller	Xinlong Wang

Table 5: Team Members and Their Components