5. Robot calibration

5.1. Program function description

After the program is running, adjust the parameters here to calibrate the linear speed and angular speed of the robot through the dynamic parameter regulator. The intuitive manifestation of calibrating the linear speed is to let the robot go forward 1 meter to see how far it actually runs and whether it is within the error range; the intuitive manifestation of calibrating the angular speed is to let the robot rotate 360 degrees to see whether the angle of rotation of the robot is within the error range.

5.2, Program code reference path

After SSH connects to the car, the location of the function source code is,

```
#Calibrate linear speed source code
/home/sunrise/yahboomcar_ws/src/yahboomcar_bringup/yahboomcar_bringup/calibrate_linear.py
#Calibrate angular speed source code
/home/sunrise/yahboomcar_ws/src/yahboomcar_bringup/yahboomcar_bringup/calibrate_angular.py
```

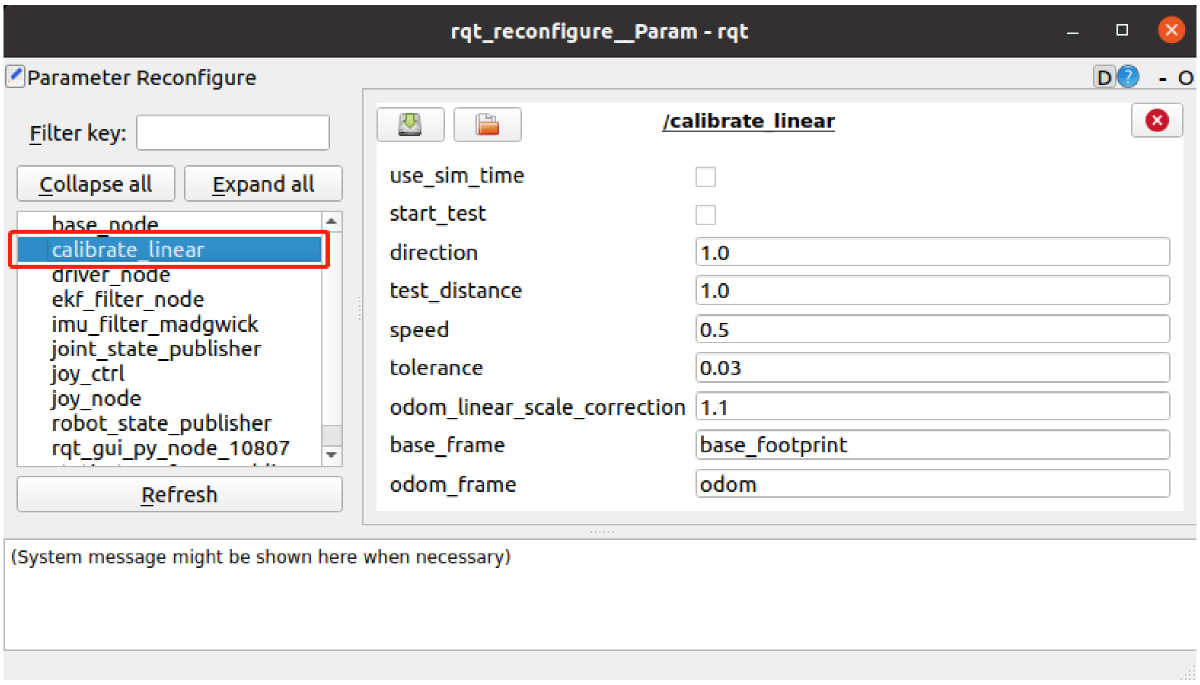
5.3, Program startup

After SSH connects to the car, enter in the terminal,

```
#Start chassis
ros2 launch yahboomcar_bringup yahboomcar_bringup_launch.py
#Calibrate linear speed
ros2 run yahboomcar_bringup calibrate_linear
#Calibrate angular speed
ros2 run yahboomcar_bringup calibrate_angular
```

Open the virtual machine terminal and enter,

```
#Dynamic parameter adjustment
ros2 run rqt_reconfigure rqt_reconfigure
```



base_node
calibrate_linear
driver_node
ekf_filter_node
imu_filter_madgwick
joint_state_publisher
joy_ctrl
joy_node
robot_state_publisher
rqt_gui_py_node_10807

use_sim_time	☐
start_test	☐
direction	1.0
test_distance	1.0
speed	0.5
tolerance	0.03
odom_linear_scale_correction	1.1
base_frame	base_footprint
odom_frame	odom

(System message might be shown here when necessary)

Take the calibration of linear speed as an example, click [start_test] to calibrate the linear speed of the car in the x direction, and observe whether the car moves the test_distance distance. The default setting here is 1m. You can customize the test distance before calibration. It must be a decimal. After setting, click the blank space and the program will automatically write it. If the distance moved by the car exceeds the acceptable error range (the value of the tolerance variable), then set the value of odom_linear_scale_correction. The following are the meanings of each parameter.

Parameter	Meaning
rate	Release frequency (can be left unchanged)
test_distance	Test distance of linear speed
speed	Linear speed
tolerance	Acceptable error value
odom_linear_scale_correction	Scale factor
start_test	Start test
direction	Direction (Linear speed test X(1)Y(0) direction)
base_frame	Parent coordinates of TF transformation
odom_frame	Child coordinates of TF transformation

The variable settings for testing angular velocity are roughly the same, except that test_distance becomes test_angle test angle and speed becomes angular velocity.

5.4, Core source code analysis

This program is mainly implemented by using TF to monitor the transformation between coordinates. By monitoring the coordinate transformation between base_footprint and odom, the robot knows "how far I have walked now/how many degrees I have turned now".

Taking calibrate_linear.py as an example, the core code is as follows,

```
#Listen to TF transformation
def get_position(self):
try:
now = rclpy.time.Time()
trans = self.tf_buffer.lookup_transform(self.odom_frame,self.base_frame,now)
return trans
except (LookupException, ConnectivityException, ExtrapolationException):
self.get_logger().info('transform not ready')
raise
return

#Get the current xy coordinates and calculate the distance based on the previous
xy coordinates
self.position.x = self.get_position().transform.translation.x
self.position.y = self.get_position().transform.translation.y
print("self.position.x: ",self.position.x)
print("self.position.y: ",self.position.y)
distance = sqrt(pow((self.position.x - self.x_start), 2) +
pow((self.position.y - self.y_start), 2))
distance *= self.odom_linear_scale_correction
```

The core code of calibrate_angular.py is as follows,

```
#Here, TF transformation is also monitored to obtain the current posture
information, but here a conversion is also made, the quaternion is converted to
Euler angle, and then returned
def get_odom_angle(self):
try:
now = rclpy.time.Time()
rot = self.tf_buffer.lookup_transform(self.odom_frame,self.base_frame,now)
print("orig_rot: ",rot.transform.rotation)
cacl_rot = PyKDL.Rotation.Quaternion(rot.transform.rotation.x,
rot.transform.rotation.y, rot.transform.rotation.z, rot.transform.rotation.w)
#print("cacl_rot: ",cacl_rot) angle_rot = cacl_rot.GetRPY()[2] #print("angle_rot:
",angle_rot) return angle_rot except (LookupException, ConnectivityException,
ExtrapolationException): self.get_logger().info('transform not ready') raise
return #Calculate rotation angle self.odom_angle = self.get_odom_angle()
self.delta_angle = self.odom_angular_scale_correction *
self.normalize_angle(self.odom_angle - self.first_angle)
```

The published TF transformation is published in the base_node node, and the code path is,

```
/home/sunrise/yahboomcar_ws/src/yahboomcar_base_node/src/base_node.cpp
```

This node will receive the data of /vel_raw, publish the odom data through mathematical calculations, and publish the TF transformation at the same time. The core code is as follows,

```
#Calculate the values of xy coordinates and xyzw quaternion. The xy coordinates
represent the position, and the xyzw quaternion represents the posture
double delta_heading = angular_velocity_z_ * vel_dt_; //radians
double delta_x = (linear_velocity_x_ * cos(heading_)-
linear_velocity_y_*sin(heading_)) * vel_dt_; //m
double delta_y = (linear_velocity_x_ *
sin(heading_)+linear_velocity_y_*cos(heading_)) * vel_dt_; //m x_pos_ += delta_x;
y_pos_ += delta_y; heading_ += delta_heading; tf2::Quaternion myQuaternion;
geometry_msgs::msg::Quaternion odom_quat; myQuaternion.setRPY(0.00,0.00,heading_
); #Publish TF transformation geometry_msgs::msg::TransformStamped t;
rclcpp::Time now = this->get_clock()->now(); t.header.stamp = now;
t.header.frame_id = "odom"; t.child_frame_id = "base_footprint";
t.transform.translation.x = x_pos_; t.transform.translation.y = y_pos_;
t.transform.translation.z = 0.0; t.transform.rotation.x = myQuaternion.x();
t.transform.rotation.y = myQuaternion.y(); t.transform.rotation.z =
myQuaternion.z(); t.transform.rotation.w = myQuaternion.w(); tf_broadcaster_-
>sendTransform(t); ```
```