5. Robot calibration

1. Program function description

After the program is run, the parameters are adjusted here through the dynamic parameter adjuster to calibrate the linear speed and angular speed of the car. Taking the X3 model as an example, the intuitive expression of the calibrated linear speed is to give the car an instruction to walk straight forward 1 meter to see how far it actually ran and whether it is within the error range; the intuitive expression of the calibrated angular speed is to let the car rotate 360 degrees and see Whether the angle of the car's rotation is within the error range.

2. Program code reference path

After entering the docker container, the location of the source code of this function is:

```
#Calibration linear speed source code
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_bringup/yahboomcar_bringup/
calibrate_linear_X3.py
#Calibration angular velocity source code
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_bringup/yahboomcar_bringup/
calibrate_angular_X3
```

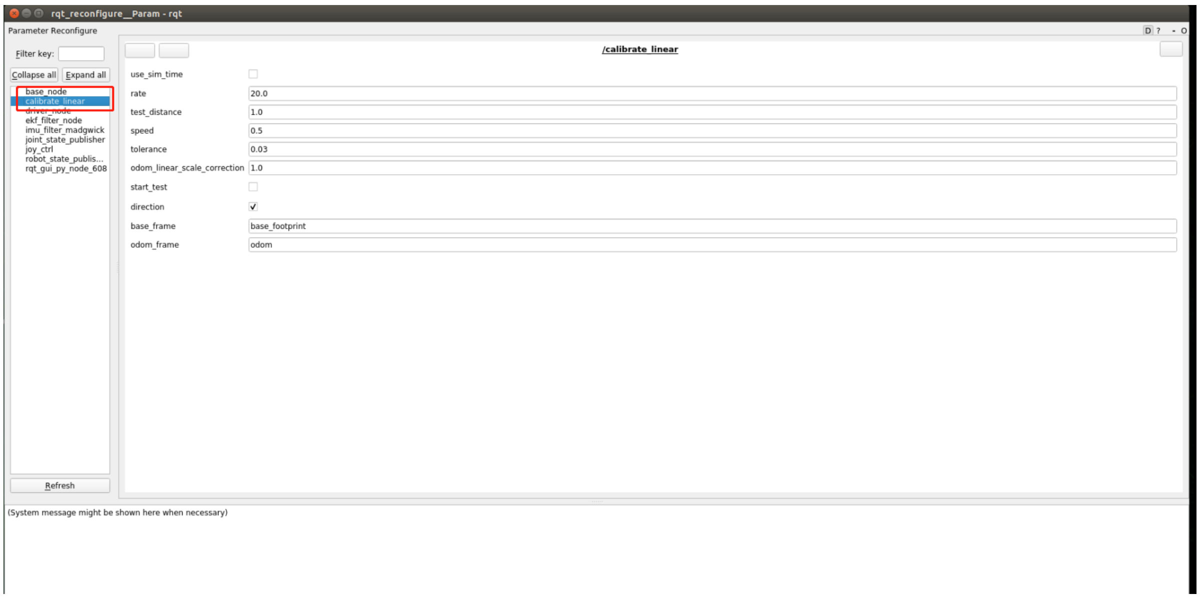
3. Program startup

After entering the docker container, according to the actual car model, taking X3 as an example, enter in the terminal,

```
#chassisdrive
ros2 launch yahboomcar_bringup yahboomcar_bringup_X3_launch.py

#Linear velocity and angular velocity run separately
#calibration line speed
ros2 run yahboomcar_bringup calibrate_linear_X3
#calibration angular velocity
ros2 run yahboomcar_bringup calibrate_angular_X3

#Dynamic parameter adjustment
ros2 run rqt_reconfigure rqt_reconfigure
```



use_sim_time	☐
rate	20.0
test_distance	1.0
speed	0.5
tolerance	0.03
odom_linear_scale_correction	1.0
start_test	☐
direction	✓
base_frame	base_footprint
odom_frame	odom

(System message might be shown here when necessary)

Taking 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 has moved 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 blank. , the program will automatically write. If the distance the car moves exceeds the acceptable error range (the value of the tolerance variable), then the value of odom_linear_scale_correction is set. The following is the meaning of each parameter,

Parameters	Meaning
rate	publishing frequency (no need to modify)
test_distance	Distance to test linear speed
speed	linear speed
tolerance	acceptable error value
odom_linear_scale_correction	Scale coefficient
start_test	Start testing
direction	direction (line speed test X (1) Y (0) direction)
base_frame	Monitor the parent coordinates of TF transformation
odom_frame	Monitor the sub-coordinates of TF transformation

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

4. Program 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 can know "how far I have walked now/how many degrees I have turned now."

Taking calibrate_linear_X3.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
```

calibrate_angular_X3 core code is as follows,

```
#Here we also monitor the TF transformation and obtain the current pose
information, but we also perform a conversion here, converting the quaternion to
Euler angle conversion, and then return
def get_odom_angle(self):
try:
now = rclpy.time.Time()
rot = self.tf_buffer.lookup_transform(self.odom_frame,self.base_frame,now)
#print("oring_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)
except (LookupException, ConnectivityException, ExtrapolationException):
self.get_logger().info('transform not ready')
return
#Calculate the 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 at the base_node node, and the code path is,

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_base_node/src/base_node_X3
```

This node will receive /vel_raw data, publish odom data through mathematical calculations, and also publish TF transformation. The core code is as follows,

```
#Calculate the xy coordinates and xyzw quaternion values. The xy two-point
coordinates represent the position, and the xyzw quaternion represents the
attitude.
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);
```