# 5、Robot calibration

## 1、Program function description

After the program is run, the linear and angular velocity of the trolley is calibrated by means of a dynamic parameter adjuster, which here adjusts the parameters. Taking the X3 model as an example, the intuitive embodiment of the calibration line speed allows the car to go straight forward by 1 meter to see how far it actually ran and whether it is within the error range; The intuitive embodiment of calibrating angular velocity allows the car to rotate 360 degrees to see if the angle of the trolley rotation is within the error range.

## 2、Program code reference path

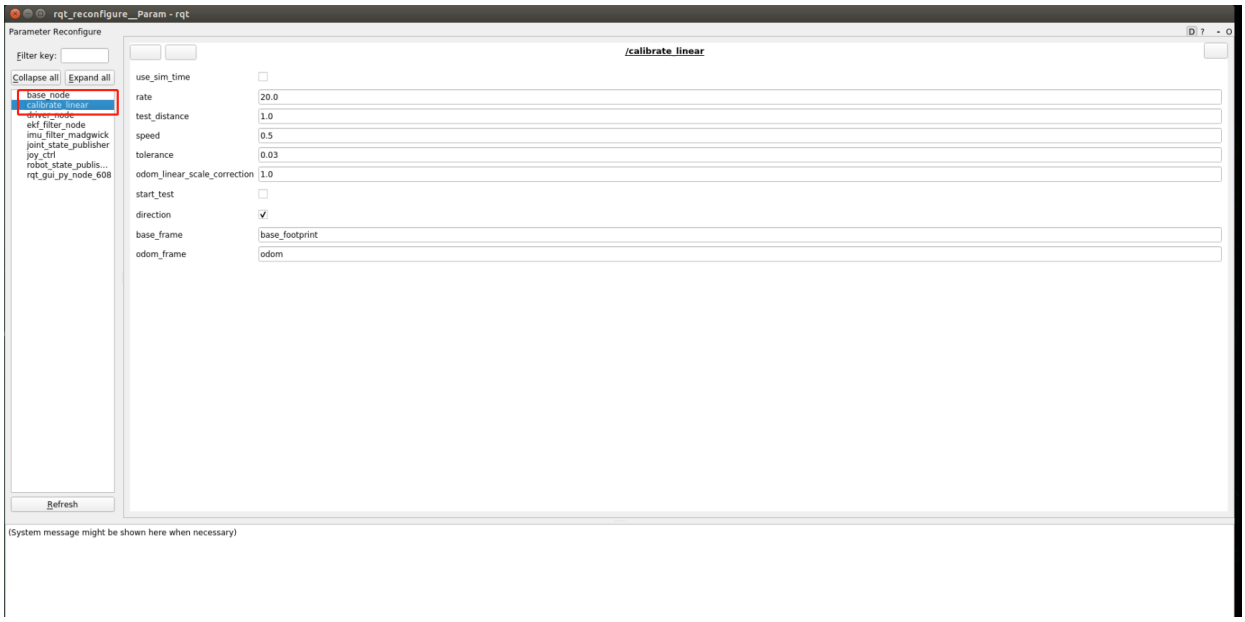
After entering the docker container, the location of the source code of this function is located at,

```
#Calibrate the line speed source code
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_bringup/yahboomcar_bringup/
calibrate_linear_X3.py
#Calibrate angular velocity source code
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_bringup/yahboomcar_bringup/
calibrate_angular_X3
```

## 3、The program starts

After entering the docker container, according to the actual model, take X3 as an example, the terminal input,

```
#Chassis drive
ros2 launch yahboomcar_bringup yahboomcar_bringup_X3_launch.py
#Calibrate line speed
ros2 run yahboomcar_bringup calibrate_linear_X3
#Calibrate angular velocity
ros2 run yahboomcar_bringup calibrate_angular_X3
#Dynamic parameter adjustment
ros2 run rqt_reconfigure rqt_reconfigure
```




Take the calibration line speed as an example, click the "start_test" trolley to calibrate the linear speed in the x direction, observe whether the trolley has moved test_distance distance, here the default setting is 1m, you can customize the test distance before calibration, it must be a decimal, after setting the click on the blank, the program will automatically write. If the trolley travels more than the range of acceptable error (the value of the tolerance variable), then the value of the odom_linear_scale_correction is set. The following are the meanings of each parameter,

| parameter | meaning |
| --- | --- |
| rate | Release frequency (can be unmodified) |
| test_distance | The distance at which the linear speed is tested |
| speed | The magnitude of the line speed |
| tolerance | The error value that can be received |
| odom_linear_scale_correction | Scale factor |
| start_test | Start testing |
| direction | Direction (linear velocity test X(1)Y(0)direction) |
| base_frame | Listen for the parent coordinates of the TF transformation |
| odom_frame | Listen for the sub-coordinates of the TF transformation |

The variable settings for testing angular velocity are generally the same, but the test_distance becomes the test_angle test angle and the speed becomes the angular velocity magnitude.

## 4、Analysis of the core source code of the program

This program mainly uses TF to monitor the transformation between coordinates, and let the robot know "how far I have gone now/how many degrees I have turned now" by listening to the coordinate transformation between the base_footprint and odom.

Taking calibrate_linear_X3.py as an example, the core code is as follows,

```
#Listen for TF transformations
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 the core code is as follows,

```
#Here, too, the TF transformation is monitored to obtain the current pose
information, but here it is also converted, converting the quaternion to the Euler
angle, and then back
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 transform is published base_node this node, and the code path is,

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

The node will receive/vel_raw data, through mathematical calculation, release odom data, and release TF transformation, the core code is as follows:

```
#Calculate the xy coordinates as well as the value of the xyzw quaternion, the xy
two-point coordinate represents the position, and the xyzw quaternion represents the
pose
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 transform
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);
```