Radar patrol

Note: The virtual machine needs to be in the same LAN as the car, and the ROS_DOMAIN_ID needs to be the same. You can check [Must-read before use] to set the IP and ROS_DOMAIN_ID on the board.

1. Program function description

The car connects to the proxy, runs the program, sets the patrol route in the dynamic parameter regulator, and then clicks Start. The car will move according to the set patrol route. At the same time, the radar on the car will scan whether there are obstacles within the set radar angle and the set obstacle detection distance. If there are obstacles, it will stop and the buzzer will sound. If there are no obstacles, it will continue to patrol.

```
Note: Before running the program, the car needs to be restarted in a stable
standing position to ensure that all sensors are reset
```

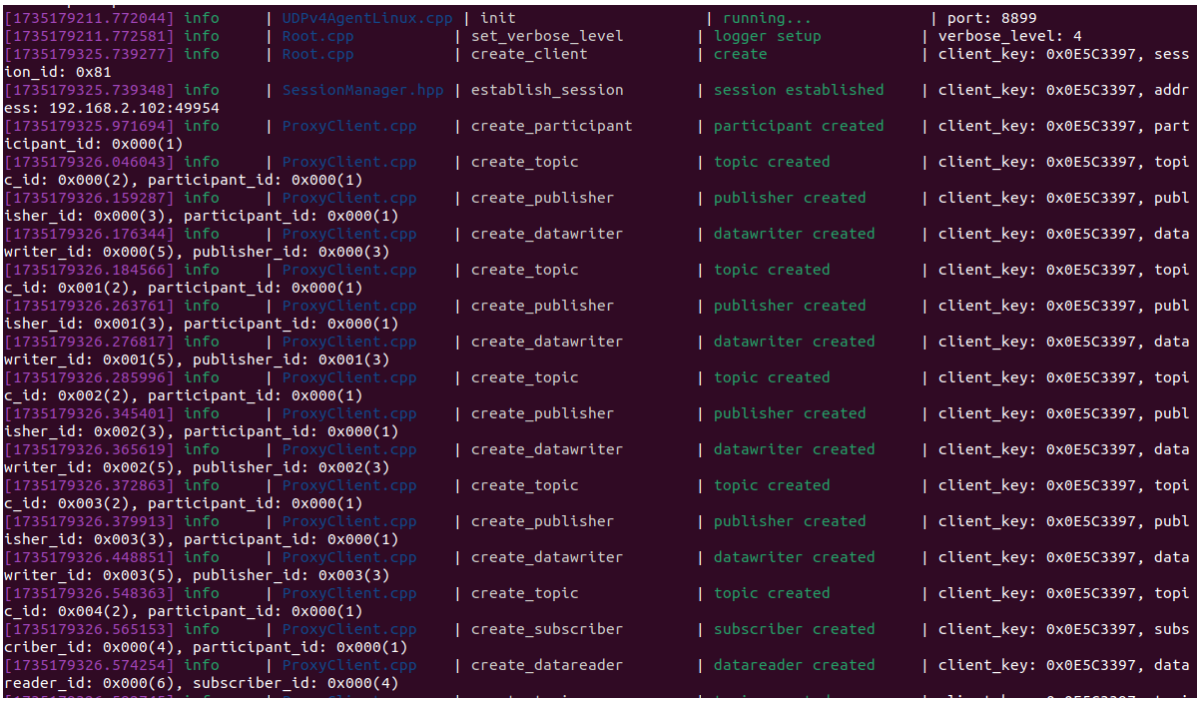
2. Start and connect the agent

Take the supporting virtual machine as an example, enter the following command to start the agent,

```
sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --
net=host microros/micro-ros-agent:humble udp4 --port 8899 -v4
```

```
yahboom@yahboom-VM:~$ sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm
 --privileged --net=host microros/micro-ros-agent:humble udp4 --port 8899 -v4
[1735179211.772044] info     | UDPv4AgentLinux.cpp | init                     |
running...             | port: 8899
[1735179211.772581] info     | Root.cpp           | set_verbose_level        | l
ogger setup           | verbose_level: 4
```

Then, turn on the car switch and wait for the car to connect to the proxy. The connection is successful as shown in the figure below.



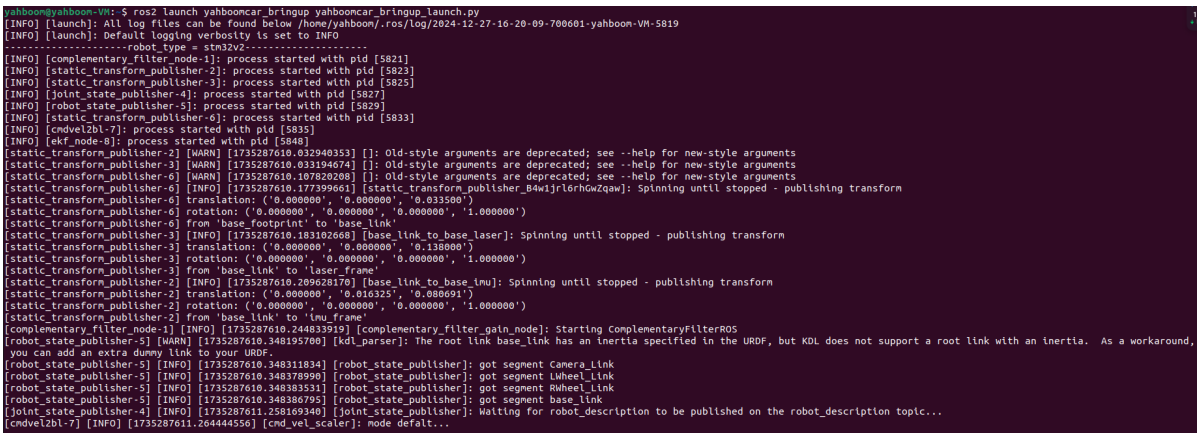
3. Start the program

3.1 Run the command

Take the matching virtual machine as an example, input in the terminal,

```
ros2 launch yahboomcar_bringup yahboomcar_bringup_launch.py
```

First, start the car to process the underlying data program. The program will publish the TF transformation of odom->base_footprint. With this TF transformation, we can calculate "how far the car has walked".



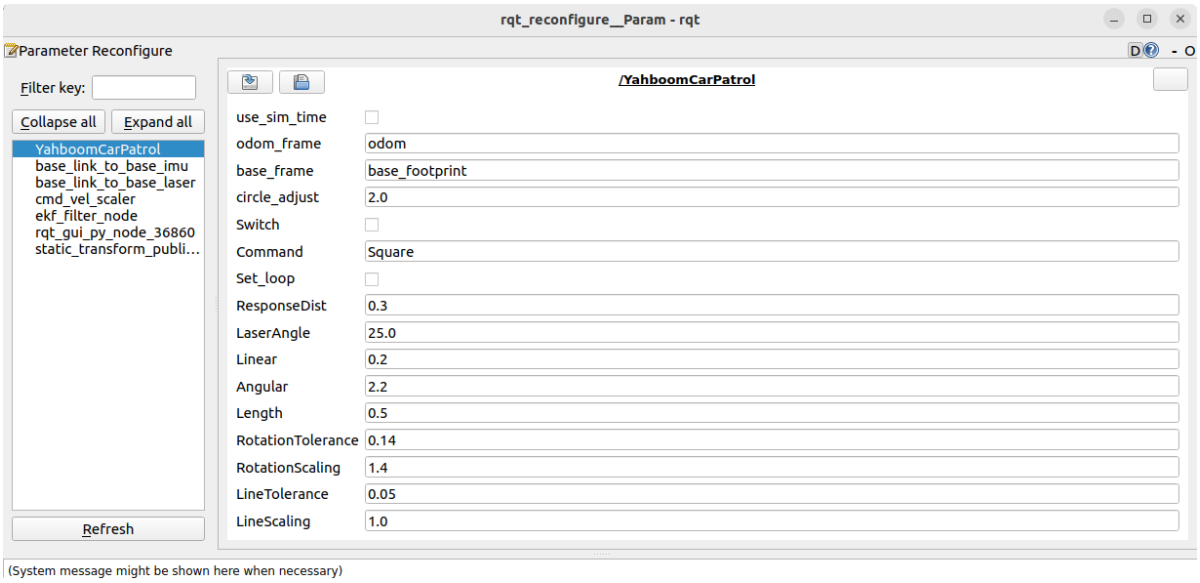
Then, start the radar patrol program and input in the terminal.

```
ros2 run yahboomcar_bringup patrol
```

```
yahboom@yahboom-VM:~$ ros2 run yahboomcar_bringup patrol
import finish
create done
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
Switch False
```

Open the parameter regulator, give the patrol route, and input in the terminal.

```
ros2 run rqt_reconfigure rqt_reconfigure
```



Note: The above nodes may not appear when you first open the application. Click Refresh to see all nodes. The YahboomCarPatrol displayed is the patrol node.

If conversion or other errors occur, this is due to network delay. The coordinate conversion of odom is not successful. You need to restart the car and connect to the agent again.

```
[INFO] [1735358962.891464389] [YahboomCarPatrol]: transform not ready
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0))

publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0))
```

Under normal network conditions, the frequency of the odometer

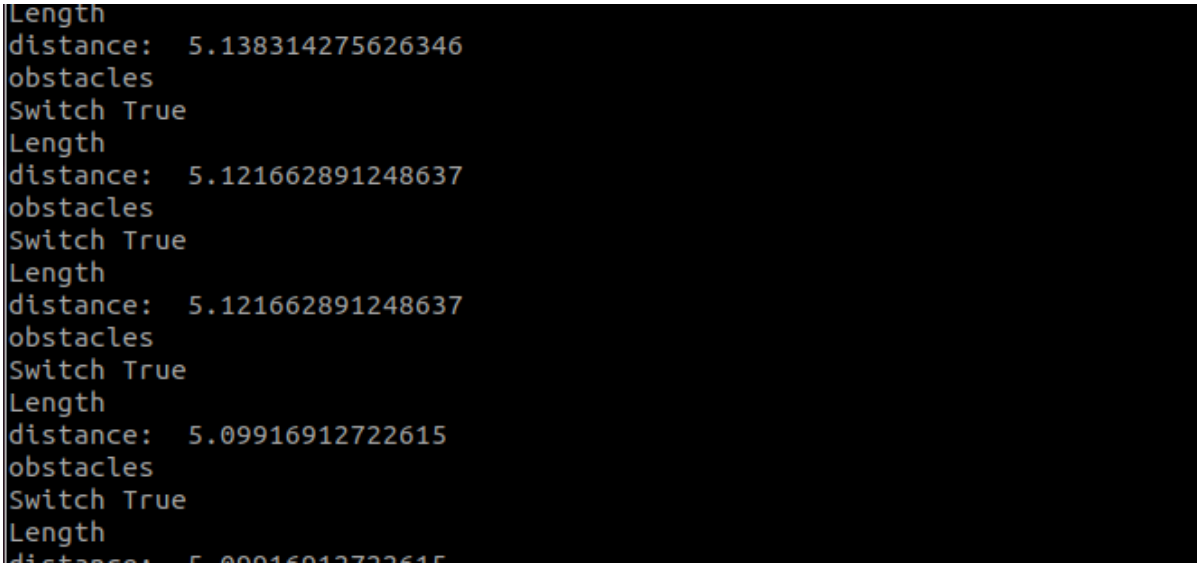
```
yahboom@yahboom-VM:~$ ros2 topic hz /odom
average rate: 16.758
        min: 0.032s max: 0.101s std dev: 0.02551s window: 19
average rate: 15.230
        min: 0.032s max: 0.167s std dev: 0.03232s window: 33
average rate: 14.526
        min: 0.032s max: 0.167s std dev: 0.03142s window: 46
average rate: 14.521
```

4. Start patrol

In the rqt interface, find the YahboomCarPatrol node. The [Command] inside is the patrol route set. Here, take the square patrol route as an example. The following will explain which patrol routes there are. After setting the route in [Command], click Switch to start patrolling.

```
distance:  0.6335016035139822
Switch True
Length
distance:  0.6335016035139822
Switch True
Length
distance:  0.7284602368836334
Switch True
Length
distance:  0.7284602368836334
Switch True
Length
distance:  0.7284602368836334
Switch True
Length
distance:  0.8070901854409693
Switch True
Length
distance:  0.8070901854409693
Switch True
Length
distance:  0.8070901854409693
Switch True
Length
```

Take a square as an example, walk in a straight line first, then rotate 90 degrees, walk in a straight line again, then rotate 90 degrees, and so on, until the route you walk is a square. If you encounter an obstacle during the walk, you will stop and the buzzer will sound.



As shown in the figure above, obstacles will be printed when you encounter an obstacle.

Other parameters in the rqt interface are as follows:

- odom_frame: the name of the odometer coordinate system
- base_frame: the name of the base coordinate system
- circle_adjust: when the patrol route is circular, this value can be used as a coefficient to adjust the size of the circle, see the code for details
- Switch: gameplay switch
- Command: patrol route, there are the following routes: [LengthTest]- straight patrol, [Circle]- circle patrol, [Square]- square patrol, [Triangle]- triangle patrol.
- Set_loop: re-patrol development, after setting, it will continue to patrol in a loop according to the set route
- ResponseDist: obstacle detection distance
- LaserAngle: radar detection angle
- Linear: linear velocity
- Angular: angular velocity
- Length: linear motion distance
- RotationTolerance: tolerance value allowed for rotation error
- RotationScaling: rotation scale coefficient

After modifying the above parameters, you need to click on the blank space to pass the parameters into the program. You can also record the parameters, modify the source code directly and recompile and update the environment.

5. Code analysis

Source code reference path (taking the supporting virtual machine as an example):

```
/home/yahboom/yahboomcar_ws/src/yahboomcar_bringup/yahboomcar_bringup
```

patrol.py, the core code is as follows,

Create radar and remote control data subscribers

```
self.sub_scan =
self.create_subscription(LaserScan,"/scan",self.LaserScanCallback,1)
self.sub_joy = self.create_subscription(Bool,"/JoyState",self.JoyStateCallback,1)
```

Create speed and buzzer data publisher

```
self.pub_cmdVel = self.create_publisher(Twist,"cmd_vel_bl",5)
self.pub_Buzzer = self.create_publisher(UInt16,'/beep',1)
```

Monitor the TF transformation of odom and base_footprint, calculate the current XY coordinates and the rotation angle,

```
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

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]
        return angle_rot
```

There are two important functions, linear advancing and rotational spin. All patrol routes are just a combination of linear and rotational movements.

advancing,

```
def advancing(self,target_distance):

    self.position.x = self.get_position().transform.translation.x
    self.position.y = self.get_position().transform.translation.y
    move_cmd = Twist()
    self.distance = sqrt(pow((self.position.x - self.x_start), 2) +
                         pow((self.position.y - self.y_start), 2))
    self.distance *= self.LineScaling
    self.error = self.distance - target_distance
    print("Length error: ",self.error)
    move_cmd.linear.x = self.Linear
    move_cmd.linear.x = move_cmd.linear.x *45
    if abs(self.error) < self.LineTolerance :
```

```
        print("stop")
        self.distance = 0.0
        self.error = 0.0
        self.x_start = self.get_position().transform.translation.x;
        self.y_start = self.get_position().transform.translation.y;
        return True
    else:
        if self.Joy_active or self.warning > 10:
            if self.moving == True:
                self.pub_cmdVel.publish(Twist())
                self.moving = False
                b = UInt16()
                b.data = 1
                self.pub_Buzzer.publish(b)
                print("obstacles")
        else:
            #print("Go")
            b = UInt16()
            b.data = 0
            self.pub_Buzzer.publish(UInt16())
            self.pub_cmdVel.publish(move_cmd)
        self.moving = True
        return False
```

Spin

```
def Spin(self,angle):
    self.target_angle = radians(angle)
    self.odom_angle = self.get_odom_angle()
    self.delta_angle = self.RotationScaling *
self.normalize_angle(self.odom_angle - self.last_angle)
    self.turn_angle -= self.delta_angle
    print("turn_angle: ",self.turn_angle)
    self.error =  self.target_angle + self.turn_angle
    print("error: ",self.error)
    self.last_angle = self.odom_angle
    move_cmd = Twist()
    if abs(self.error) < self.RotationTolerance or self.Switch==False :
        #self.pub_cmdVel.publish(Twist())
        self.turn_angle = 0.0
        self.last_angle = self.get_odom_angle()
        return True
    if self.Joy_active or self.warning > 10:
        if self.moving == True:
            self.pub_cmdVel.publish(Twist())
            self.moving = False
            b = UInt16()
            b.data = 1
            self.pub_Buzzer.publish(b)
            print("obstacles")
    else:
        b = UInt16()
        b.data = 0
        self.pub_Buzzer.publish(UInt16())
        if self.Command == "Square" or self.Command == "Triangle":
```

```
            length = self.Linear * self.circle_adjust / self.Length *200
            move_cmd.linear.x = self.Linear *45
            move_cmd.angular.z = copysign(length, self.error)
        elif self.Command == "Circle":
            length = self.Linear * self.circle_adjust / self.Length *200
            move_cmd.linear.x = self.Linear *45
            move_cmd.angular.z = copysign(length, self.error)
            '''move_cmd.linear.x = 0.2
            move_cmd.angular.z = copysign(2, self.error)'''
        self.pub_cmdVel.publish(move_cmd)
    self.moving = True
```

Now that we have the functions for walking in a straight line and rotating, we can arrange and combine them according to the patrol route. Taking a square as an example,

```
def Square(self):
    if self.index == 0:
        #print("Length")
        step1 = self.advancing(self.Length)
        #sleep(0.5)
        if step1 == True:
            #self.distance = 0.0
            self.index = self.index + 1;
            self.x_start = self.get_position().transform.translation.x;
            self.y_start = self.get_position().transform.translation.y;
            self.Switch  =
rclpy.parameter.Parameter('Switch',rclpy.Parameter.Type.BOOL,True)
            all_new_parameters = [self.Switch]
            self.set_parameters(all_new_parameters)
            return False
    elif self.index == 1:
        print("Spin")
        step2 = self.Spin(90)
        #sleep(0.5)
        if step2 == True:
            self.index = self.index + 1;
            self.last_angle = self.get_odom_angle()
            self.x_start = self.get_position().transform.translation.x;
            self.y_start = self.get_position().transform.translation.y;
            print("x_start: ",self.x_start)
            print("y_start: ",self.y_start)
            print("last_angle: ",self.last_angle)
            self.pub_cmdVel.publish(Twist())
            return False
.....
```