# 5.Patrol

## 1. Program function description

After the program is started, open the patrol route set by the dynamic parameter setter and click "switch" on the GUI interface.Muto moves according to the set patrol route. During operation, the lidar works at the same time. If an obstacle is detected within the detection range, it will stop. After the controller program is turned on, you can also press the R2 button to pause/continue Muto movement. The controller program is turned on by default.

## 2. Program code reference path

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

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_laser/yahboomcar_laser/patrol_a1.py       #a1雷达
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_laser/yahboomcar_laser/patrol_4ROS.py     #4ros雷达
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_laser/launch/laser_patrol_launch.py
```

## 3. Configuration before use

Note: Since the Muto series robots are equipped with multiple lidar devices, the factory system has been configured with routines for multiple devices. However, since the product cannot be automatically recognized, the lidar model needs to be manually set.

After entering the container: Make the following modifications according to the lidar type:

```
root@ubuntu:/# cd
root@ubuntu:~# vim .bashrc
```

```
# env
alias python=python3
export ROS_DOMAIN_ID=26

export ROBOT_TYPE=Muto
export RPLIDAR_TYPE=a1          # a1, 4ROS
echo "-----------------------------------"
echo -e "ROS_DOMAIN_ID: \033[32m$ROS_DOMAIN_ID\033[0m"
echo -e "my_robot_type: \033[32m$ROBOT_TYPE\033[0m | my_lidar: \033[32m$RPLIDAR_TYPE\033[0m"
echo "-----------------------------------"

#colcon_cd
source /usr/share/colcon_cd/function/colcon_cd.sh
export _colcon_cd_root=/root/yahboomcar_ros2_ws/yahboomcar_ws
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash

#ros2
source /opt/ros/foxy/setup.bash
source /root/yahboomcar_ros2_ws/yahboomcar_ws/install/setup.bash
source /root/yahboomcar_ros2_ws/software/library_ws/install/setup.bash
```

After the modification is completed, save and exit vim, and then execute:

```
root@jetson-desktop:~# source .bashrc
------------------------------------
ROS_DOMAIN_ID: 26
my_robot_type: Muto | my_lidar: a1
------------------------------------
root@jetson-desktop:~#
```

You can see the current modified lidar type.

# 4. Program startup

## 4.1. Start command

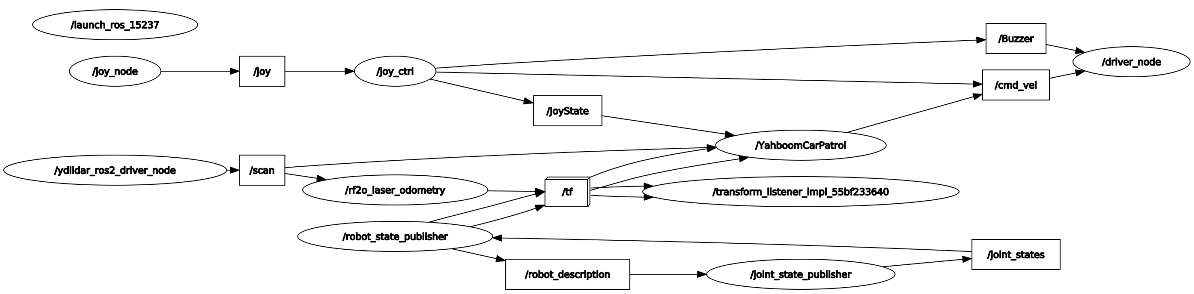
After entering the docker container, enter in the terminal

```
#publish odometer
ros2 launch rf2o_laser_odometry rf2o_laser_odometry.launch.py
#Start patrol program
ros2 launch yahboomcar_laser laser_patrol_launch.py

#Open the dynamic parameter setter (it is recommended to open it with a virtual machine and configure multi-machine communication in advance)
ros2 run rqt_reconfigure rqt_reconfigure

#For the set patrol route, click "switch" on the rqt_reconfigure GUI interface to start patrolling.
- LengthTest: Straight Line Test
- Circle: circular route patrol
- Square: square line patrol
- Triangle: triangular route patrol
```

## 4.2. View topic communication node graph

docker terminal input

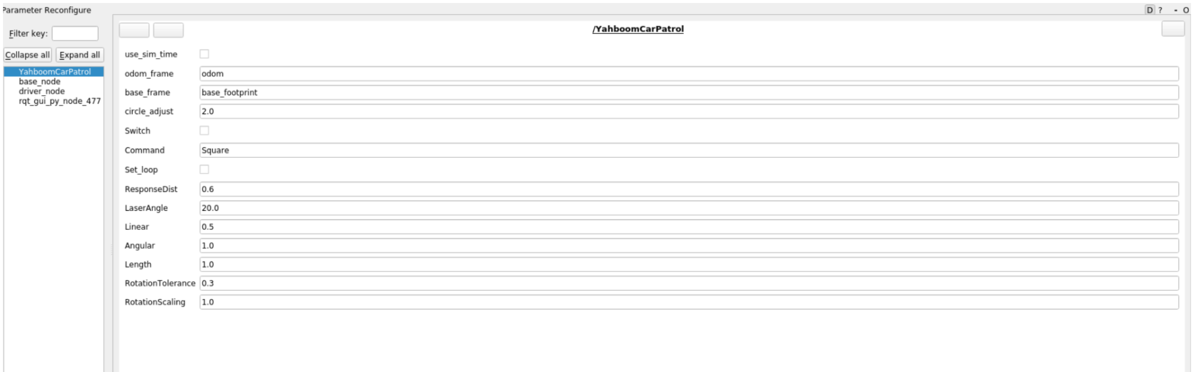
```
ros2 run rqt_graph rqt_graph
```




Set the parameter size through the dynamic parameter adjuster, terminal input

```
ros2 run rqt_reconfigure rqt_reconfigure
```

Each parameter of the dynamic parameter adjuster is described as follows:




| Parameter name | Parameter meaning |
|---|---|
| odom_frame | Odometer coordinate system |
| base_frame | Base coordinate system |
| circle_adjust | Rotation angular speed adjustment coefficient |
| Switch | Game switch |
| Command | Patrol route |
| Set_loop | Set loop |
| ResponseDist | Lidar obstacle avoidance response distance |
| LaserAngle | Lidar scan angle |
| Linear | Line speed |
| Angular | Angular speed |
| Length | Linear test distance |
| RotationTolerance | Steering error tolerance |
| RotationScaling | Angle proportional coefficient |

After the program starts, in the GUI interface of the dynamic parameter adjuster interface, enter any of the following routes in the Comand column:

- LengthTest： straight line test
- Circle： circular route patrol
- Square： square route patrol
- Triangle： triangle route patrol

After selecting the route, click on the blank space to write the parameters, and then click the Switch button to start patrolling. If loop is set, you can loop the last route for patrol. If loop is false, it will stop after completing the patrol.

## 5. Core source code analysis

The implementation source code of this code is to subscribe to the TF transformation of odom and base_footprintf, so that you can know "how long you have walked" at any time, and then issue speed instructions according to the set route. Taking Triangle as an example, we will analyze it here.

```python
#Set the patrol route and enter the self.Triangle function
self.command_src = "Triangle"
triangle = self.Triangle()
#Parsed with part of the self.Triangle code,
def Triangle(self):
    if self.index == 0:
        print("Length")
        step1 = self.advancing(self.Length) #Start with a straight line and walk
through one side of the triangle
        #sleep(0.5)
        if step1 == True:
            #self.distance = 0.0
            self.index = self.index + 1;
            self.Switch  =
rclpy.parameter.Parameter('Switch',rclpy.Parameter.Type.BOOL,True)
            all_new_parameters = [self.Switch]
            self.set_parameters(all_new_parameters)
    elif self.index == 1:
            print("Spin")
            step2 = self.Spin(120)#Then change the direction and turn to 120,
triangle 3*120=360
            #sleep(0.5)
            if step2 == True:
                self.index = self.index + 1;
                self.Switch  =
rclpy.parameter.Parameter('Switch',rclpy.Parameter.Type.BOOL,True)
                all_new_parameters = [self.Switch]
                self.set_parameters(all_new_parameters)
#After completing the following three loops, the triangle patrol is completed,
mainly looking at the self.advancing and self.Spin functions. After these two
functions are executed, True will be returned.
 def advancing(self,target_distance):
    #The following is to obtain the xy coordinates, calculate them with the
coordinates at the previous moment, and calculate how far you have traveled
     #The way to obtain xy coordinates is to monitor the tf transformation of
odom and base_footprint. For this part, please refer to the self.get_position()
function.
    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
    print("distance: ",self.distance)
    self.error = self.distance - target_distance
    move_cmd.linear.x = self.Linear
    if abs(self.error) < self.LineTolerance :
        print("stop")
        self.distance = 0.0
        self.pub_cmdVel.publish(Twist())
        self.x_start = self.position.x;
        self.y_start = self.position.y;
        self.Switch  =
rclpy.parameter.Parameter('Switch',rclpy.Parameter.Type.BOOL,False)
        all_new_parameters = [self.Switch]
```

```
            self.set_parameters(all_new_parameters)
            return True
        else:
            if self.Joy_active or self.warning > 10:
                if self.moving == True:
                    self.pub_cmdVel.publish(Twist())
                    self.moving = False
                    print("obstacles")
                 else:
                    #print("Go")
                    self.pub_cmdVel.publish(move_cmd)
                self.moving = True
                return False

def Spin(self,angle):
        self.target_angle = radians(angle)
        #The following is to obtain the pose and calculate how many degrees you
have turned. To obtain the pose, you can refer to the self.get_odom_angle
function. It is also obtained by monitoring the TF transformation of odom and
base_footprint.
        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.Switch  =
rclpy.parameter.Parameter('Switch',rclpy.Parameter.Type.BOOL,False)
            all_new_parameters = [self.Switch]
            self.set_parameters(all_new_parameters)'''
            return True
        if self.Joy_active or self.warning > 10:
            if self.moving == True:
                self.pub_cmdVel.publish(Twist())
                self.moving = False
                print("obstacles")
        else:
            if self.Command == "Square" or self.Command == "Triangle":
                #move_cmd.linear.x = 0.2
                move_cmd.angular.z = copysign(self.Angular, self.error)
            elif self.Command == "Circle":
                length = self.Linear * self.circle_adjust / self.Length#The
circle_adjust here is the coefficient of the rotation angle. From the
calculation, it can be understood that the larger the length, the larger the
radius of the circle.
                #print("length: ",length)
                move_cmd.linear.x = self.Linear
                move_cmd.angular.z = copysign(length, self.error)
                #print("angular: ",move_cmd.angular.z)
```

```
            '''move_cmd.linear.x = 0.2
            move_cmd.angular.z = copysign(2, self.error)'''
        self.pub_cmdVel.publish(move_cmd)
    self.moving = True
```