4.Lidar follow

This chapter needs to be used with a car chassis and other sensors to operate. This is just an explanation of the implementation method. In fact, it cannot be run. It needs to be used with the company's rosmaster-X3 car to achieve this function.

If you need to transplant it to your own motherboard, you need to install other dependency files.

1. Function Description

After the program is started, the radar scans the nearest object and then locks on it. The object moves and the car moves with it.

2. Code path

```
~/oradar_ws/src/yahboomcar_laser/yahboomcar_laser/laser_tracker.py
```

3. Start up

Input following command:

```
ros2 launch yahboomcar_laser laser_tracker_launch.py
```

4. Core code analysis

laser_tracker_launch.py

```
from launch import LaunchDescription
from launch_ros.actions import Node

import os
from ament_index_python.packages import get_package_share_directory
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource

def generate_launch_description():
    lidar_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
         get_package_share_directory('rplidar_ros'), 'launch'),
         '/lidar.launch.py'])
      )

    laser_tracker_node = Node(
        package='yahboomcar_laser',
        executable='laser_tracker',
    )

    bringup_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
        get_package_share_directory('yahboomcar_bringup'), 'launch'),
        '/yahboomcar_bringup_X3_launch.py'])
```

```
    )

    launch_description = LaunchDescription([
        lidar_node,
        laser_tracker_node,
        bringup_node
        ])
    return launch_description
```

Start the radar lidar_node, chassis bringup_node and tracking function section laser_tracker_node.

laser_tracker.py

The radar callback function explains how to obtain the obstacle distance information at each angle. Then find the nearest point, then determine the distance, then calculate the speed data, and finally publish it.

```
    def registerScan(self, scan_data):
        if not isinstance(scan_data, LaserScan): return
        if self.Joy_active == True or self.Switch == False:
            if self.Moving == True:
                self.pub_vel.publish(Twist())
                self.Moving = not self.Moving
            return

        self.Moving = True

        frontDistList = []
        frontDistIDList = []
        minDistList = []
        minDistIDList = []
        ranges = np.array(scan_data.ranges)
        for i in range(len(ranges)):
            angle = (scan_data.angle_min + scan_data.angle_increment * i) *
RAD2DEG
            if abs(angle) > (180 - self.priorityAngle):
                if ranges[i] < (self.ResponseDist + offset):
                    frontDistList.append(ranges[i])
                    frontDistIDList.append(angle)
            elif (180 - self.LaserAngle) < angle < (180 - self.priorityAngle):
                minDistList.append(ranges[i])
                minDistIDList.append(angle)
            elif (self.priorityAngle - 180) < angle < (self.LaserAngle - 180):
                minDistList.append(ranges[i])
                minDistIDList.append(angle)

        if len(frontDistIDList) != 0:
            minDist = min(frontDistList)
            minDistID = frontDistIDList[frontDistList.index(minDist)]
        else:
            minDist = min(minDistList)
            minDistID = minDistIDList[minDistList.index(minDist)]

        velocity = Twist()
```

```
        if abs(minDist - self.ResponseDist) < 0.1:
            velocity.linear.x = 0.0
        else:
            velocity.linear.x = -self.lin_pid.pid_compute(self.ResponseDist,
minDist)

        angle_pid_compute = self.ang_pid.pid_compute(minDistID / 72, 0)
        if abs(angle_pid_compute) < 0.02:
            velocity.angular.z = 0.0
        else:
            velocity.angular.z = angle_pid_compute
        self.pub_vel.publish(velocity)
```