به نام خدا

دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر

اصول علم ربات

پروژه نهایی

نام و نام خانوادگی	مریم کرمانشاهی، نارین حصاری
شماره دانشجویی	۹۷۲۳۰۷۳ و ۹۸۲۹۰۱۴
تاریخ ارسال گزارش	۱۴۰۲/۴/۱۰

فهرست گزارش سوالات

سناریو ۱

مقدمه

الگوریتم VFH یکی از روش‌های پرکاربرد و موثر در هدایت ربات‌ها و خودروهای هوش مصنوعی در محیط‌های پویا و ناهموار است. نام VFH مخفف عبارت "Vector Field Histogram" می‌باشد که به اشاره به مبدل برداری هیستوگرام موانع می‌باشد. این الگوریتم به منظور مسیریابی اتوماتیک و ایمن ربات‌ها و خودروها در محیط‌های پر از موانع مورد استفاده قرار می‌گیرد.

در محیط‌های پویا و پر از موانع، انتخاب بهترین مسیر برای رسیدن به هدف می‌تواند چالش‌های زیادی را به دنبال داشته باشد. الگوریتم VFH با استفاده از حسگرهای لیزری یا سونار، اطلاعات موانع اطراف را به دست می‌آورد و با استفاده از هیستوگرام و اطلاعات جهت هدف، مسیری ایمن و بهینه برای ربات تعیین می‌کند.

همانطور که در کد ارائه شده نیز مشاهده می‌شود، الگوریتم VFH با محاسبه هیستوگرام موانع اطراف ربات، به دنبال دره‌ها (valleys) در هیستوگرام می‌گردد که می‌تواند مسیر راه‌حل ممکنی به هدف ارائه دهد. انتخاب دره مناسب و هدایت ربات به سمت آن، امکان حرکت ایمن و پیش‌رفت به هدف را فراهم می‌کند.

در این الگوریتم، معیارهای مختلفی نظیر آستانه‌های محدودیتی، سرعت خطی و زاویه‌ی چرخش ربات قابل تنظیم هستند که با توجه به نیاز و محیط مسیریابی می‌توانند تغییر کنند.

الگوریتم VFH به دلیل قابلیت انطباق به شرایط محیطی مختلف، عدم وابستگی به نقشه‌ی ساختاری محیط و قدرت محاسباتی مناسب، یکی از روش‌های پرطرفدار در هدایت و مسیریابی ربات‌ها و خودروهای هوشمند می‌باشد.

کد vfh_node

```
#!/usr/bin/python3

import tf
import math
import rospy
```

```
import numpy as np
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan


class VFH():

    def __init__(self):

        rospy.init_node('vfh_node', anonymous = False)
        self.cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size = 10)

        self.a = 1
        self.b = 0.25
        self.l = 2
        self.s_max = 15
        self.thresh = 6
        self.path_length = 0.5
        self.sector_size = 5
        self.goal_index = -1
        self.linear_epsilon = 0.3
        self.ang_epsilon = 0.1
        self.linear_vel = 0.1
        self.ang_vel = 0.1


    def run(self):
        goal_x_arr = [4.5, 3.0, 0.5 , 1.5, 3.5, 6, 7, 8, 13]
        goal_y_arr = [0.5 , 4.5, 2.0 , 5.5, 6.5, 6, 3, 7, 7]
        for i in range(len(goal_x_arr)):
            goal_x = goal_x_arr[i]
            goal_y = goal_y_arr[i]

            while not rospy.is_shutdown():
                vfh_class.guidance(goal_x, goal_y)
                current_position = self.get_pose()[0]
                rospy.loginfo(f"Current goal: ({goal_x}, {goal_y})")

                if (abs(current_position.x - goal_x) < 0.3) and
(abs(current_position.y - goal_y) < 0.3):
                    rospy.loginfo(f"Robot reached current goal.")
                    break


    def guidance(self, curr_goal_x, curr_goal_y):

        sectors = self.find_h()
```

```
        goal_sector = self.get_goal_sector(curr_goal_x, curr_goal_y)
        selected_sectors = self.calculate_thresh(sectors)

        if sectors[goal_sector] < self.thresh:
            best_sector = goal_sector

        else:
            best_sector = self.select_valley(selected_sectors, goal_sector)

        if best_sector > 36:
            best_sector -= 72

        angle = math.radians(best_sector * 5)
        self.vfh_controller(angle)


    def get_pose(self):
        odom = rospy.wait_for_message("/odom", Odometry)
        orientation = odom.pose.pose.orientation
        pose = odom.pose.pose.position
        roll, pitch, yaw = tf.transformations.euler_from_quaternion((
            orientation.x, orientation.y, orientation.z, orientation.w
        ))

        return pose, yaw


    def find_h(self):
       h = []
       self.laser_scan = rospy.wait_for_message("/scan", LaserScan)
       self.sector_num = int(len(self.laser_scan.ranges) /
self.sector_size)

       for i in range(self.sector_num):
           sum_m = 0
           for j in range(i * self.sector_size, (i + 1)*self.sector_size):
               d = min(6, self.laser_scan.ranges[j])
               m = self.a - self.b * d
               sum_m += m

           h.append(sum_m)

       return self.get_h_prime(h)


    def get_h_prime(self, sectors):
        h_prime = []
```

```
        for i in range(self.sector_num):
            total_h = 0
            for j in range(-2,3):
                if abs(j) == 2:
                    k = 1

                else:
                    k = 2

                if  self.sector_num <= i + j:
                    j = j * -1
                    i  = self.sector_num - i - 1

                total_h += k * sectors[i+j]

            total_h = total_h / (2*self.l + 1)
            h_prime.append(total_h)

        return h_prime


    def calculate_thresh(self ,sectors):

        thresh_arr = []
        for i in range(self.sector_num):

            if sectors[i] < self.thresh:
                thresh_arr.append(i)

        return thresh_arr


    def get_goal_sector(self, curr_goal_x, curr_goal_y):

        pose, yaw = self.get_pose()
        angle = math.atan2(curr_goal_y - pose.y, curr_goal_x - pose.x)

        if angle < 0:
            angle += 2 * math.pi
        dif = angle - yaw

        if dif < 0:
            dif += 2 * math.pi

        goal_idx = int(math.degrees(dif) / self.sector_size)
        goal_sector_ = goal_idx % self.sector_num
        return goal_sector_
```

```
    def calculate_valley_arr(self, selected_sectors):
        valley_arr = []
        curr_valley=[]
        for i in range(len(selected_sectors)):
            j = i - 1

            if i == 0 :
                curr_valley.append(selected_sectors[i])
                continue

            if selected_sectors[i] - selected_sectors[j] > 1:
                valley_arr.append(curr_valley)
                curr_valley = []

            curr_valley.append(selected_sectors[i])

        valley_arr.append(curr_valley)
        curr_valley = []

        if  valley_arr[-1][-1] == (self.sector_num -1) and valley_arr[0][0]
== 0:
            curr_valley = valley_arr.pop(0)
            for i in curr_valley:
                valley_arr[-1].append(i)

        return valley_arr


    def select_valley(self, selected_sectors, goal_sector):
        curr_idx = 0
        curr_min= math.inf
        valley_arr = self.calculate_valley_arr(selected_sectors)
        for i in range(len(valley_arr)):

            for j in range(len(valley_arr[i])):

                dist = abs(goal_sector - valley_arr[i][j])
                if dist > 36:
                    dist = 72 - dist

                if dist < curr_min:
                    curr_idx = i
                    curr_min = dist

        nearest_valley = valley_arr[curr_idx]

        if len(nearest_valley) <= self.s_max:
```

```
            candidate = nearest_valley[int(len(nearest_valley) / 2)]
            return candidate

        else :
            candidate = nearest_valley[curr_idx + int(self.s_max / 2)]
            return candidate


    def vfh_controller(self, angle):
        prev_angle = self.get_pose()[1]
        remaining = angle

        rospy.sleep(1)
        sign = 1
        if angle > math.pi:
            angle -= 2 * math.pi

        if angle < -math.pi:
            angle += 2 *math.pi

        if angle < 0:
            sign = -1

        twist = Twist()
        twist.angular.z = sign * self.ang_vel
        self.cmd_vel.publish(twist)

        while  self.ang_epsilon <= abs(remaining):

            curr_angle = self.get_pose()[1]
            delta = curr_angle - prev_angle

            if abs(delta) < 0.2:
                remaining -= delta

            prev_angle = curr_angle


        twist.angular.z = 0
        self.cmd_vel.publish(twist)
        remaining = self.path_length
        prev_pose = self.get_pose()[0]

        rospy.sleep(1)

        twist = Twist()
        twist.linear.x = self.linear_vel
        self.cmd_vel.publish(twist)
```

```
        while  self.linear_epsilon <= remaining:
            curr_pose = self.get_pose()[0]
            delta = np.linalg.norm([curr_pose.x - prev_pose.x, curr_pose.y
- prev_pose.y])
            remaining -= delta
            remaining = abs(remaining)
            prev_pose = curr_pose

        twist.linear.x = 0
        self.cmd_vel.publish(twist)
        rospy.sleep(1)
        self.cmd_vel.publish(Twist())


if __name__ == '__main__':
    try:
        vfh_class = VFH()
        vfh_class.run()

    except rospy.ROSInterruptException:
        pass
```

کد لانچ فایل

```
<launch>

  <node pkg="final_project" type="vfh.py" name="vfh_node"
output="screen"></node>

  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type
[burger, waffle, waffle_pi]"/>
  <arg name="world_name_file" default="empty.world"/>
  <arg name="x_pos" default="0.0"/>
  <arg name="y_pos" default="0.0"/>
  <arg name="z_pos" default="0.0"/>
  <arg name="yaw" default="0.0"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find
turtlebot3_gazebo)/worlds/updated_maze.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
```

```
    <arg name="debug" value="false"/>
  </include>

  <param name="robot_description" command="$(find xacro)/xacro --inorder
$(find turtlebot3_description)/urdf/turtlebot3_$(arg model).urdf.xacro" />

  <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf" args="-urdf -
model turtlebot3_$(arg model) -x $(arg x_pos) -y $(arg y_pos) -z $(arg
z_pos) -Y $(arg yaw) -param robot_description" />

</launch>
```

توضیح توابع

این یک پیاده‌سازی Python از الگوریتم Vector Field Histogram (VFH) است که برای مسیریابی ربات در ROS (سیستم عامل ربات) استفاده می‌شود. الگوریتم VFH به ربات کمک می‌کند تا به مواضع هدف تعیین شده پیشینه خود حرکت کند و در عین حال از موانع در مسیر خودداری کند.

- کلاس VFH: در این قسمت، نود ROS مقدمه‌ای را راه‌اندازی می‌کند، یک publisher برای تاپیک /cmd_vel را برای کنترل حرکت ربات تنظیم می‌کند و پارامترهای مختلفی را برای الگوریتم VFH تعیین می‌کند.

- run : این تابع حلقه اصلی الگوریتم VFH برای هدایت ربات به مجموعه‌ای از مواضع هدف پیشینه را ایجاد می‌کند. این تابع از طریق لیستی از مواضع هدف حرکت راهنما را برای رسیدن ربات به هر هدف فراخوانی می‌کند تا ربات به موقعیت هدف برسد.

- guidance(curr_goal_x, curr_goal_y): این تابع هسته الگوریتم VFH است که دستورات هدایتی را برای ربات برای رسیدن به هدف مشخص شده ایجاد می‌کند. ابتدا، با استفاده از تابع find_h ، هیستوگرام موانع اطراف ربات را محاسبه می‌کند. سپس، براساس هدف هدف کنونی، به کمک توابع get_goal_sector و calculate_thresh و select_valley ، بهترین بخش یا درهای را جهت هدایت به سمت هدف انتخاب می‌کند. در نهایت، سرعت زاویه‌ای لازم برای

هدایت ربات به جهت انتخاب شده را با استفاده از تابع vfh_controller محاسبه می‌کند و دستورات مربوطه را به ربات ارسال می‌کند.

- get_pose : این تابع، موقعیت و جهت فعلی ربات را با استفاده از پیام Odometry دریافت می‌کند و بازگشت می‌دهد.

- find_h : این تابع محاسبه هیستوگرام موانع اطراف ربات را با استفاده از پیام LaserScan انجام می‌دهد.

- get_h_prime(sectors): این تابع، هیستوگرام بخش‌هایی از موانع اطراف ربات را با استفاده از تابع get_h_prime محاسبه می‌کند.

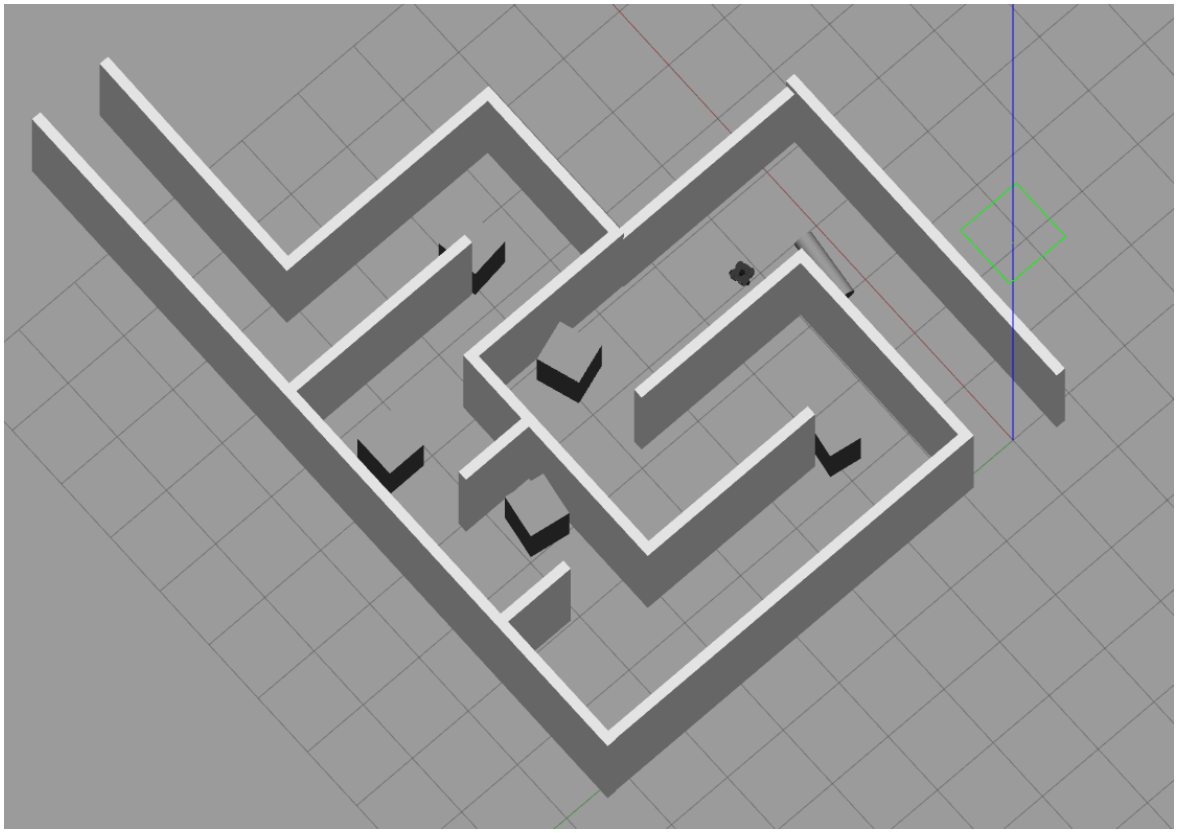
- calculate_thresh(sectors): این تابع، فهرستی از بخش‌هایی را که مقدار آن‌ها در هیستوگرام کمتر از آستانه معینی است، محاسبه می‌کند.

- get_goal_sector(curr_goal_x, curr_goal_y): این تابع بر اساس هدف هدف کنونی و موقعیت و جهت فعلی ربات، بخش هدف را برای هدایت مشخص می‌کند.

- calculate_valley_arr(selected_sectors): این تابع بر اساس بخش‌های انتخاب شده، درها را محاسبه می‌کند.

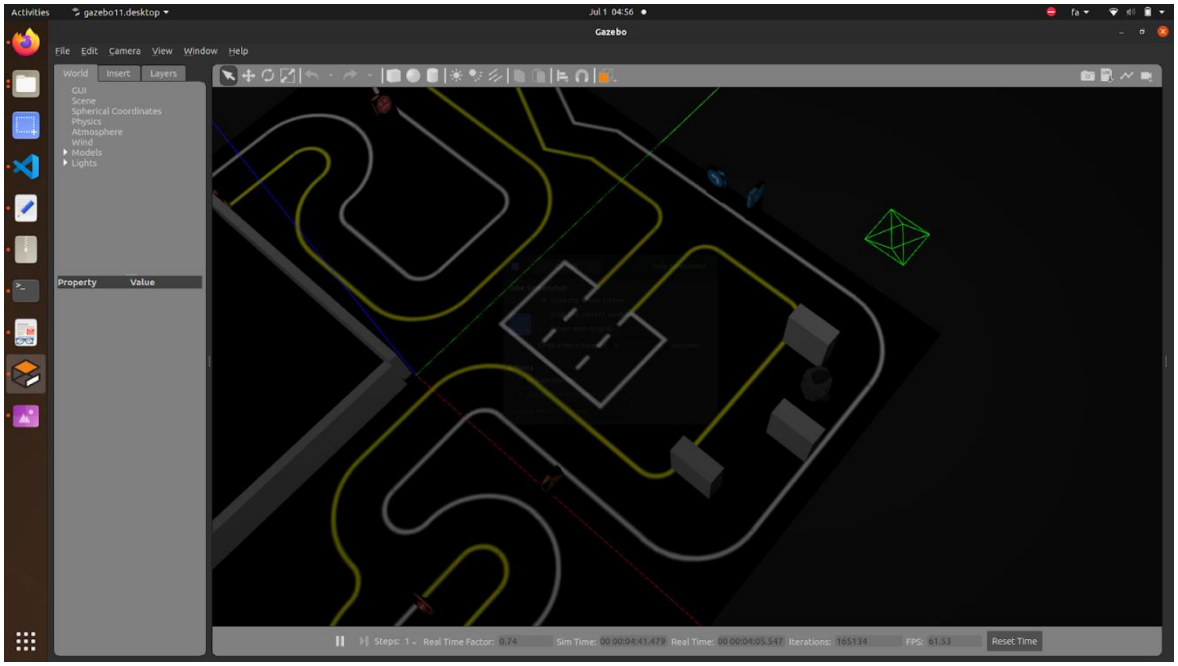
- select_valley(selected_sectors, goal_sector): این تابع، بین درهای موجود، درهای نزدیکترین به هدف را انتخاب می‌کند و بهترین دره را برمی‌گرداند.

- vfh_controller(angle): این تابع با استفاده از دستورات کنترل سرعت زاویه‌ای و خطی ربات، ربات را به سمت جهت مشخص شده هدایت می‌کند.
- در نهایت، در بخش '__main__ == __name__ if__'، یک نمونه از کلاس VFH ایجاد می‌شود و تابع run فراخوانی می‌شود تا الگوریتم VFH روی ربات اجرا شود.

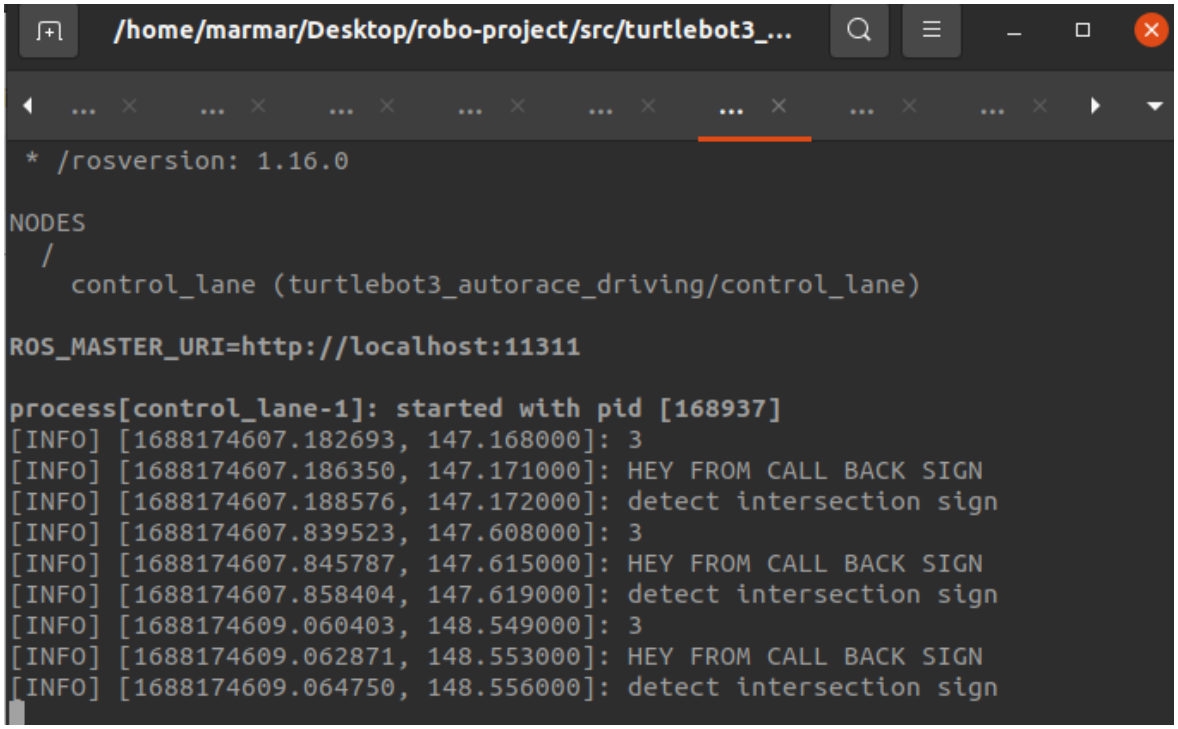


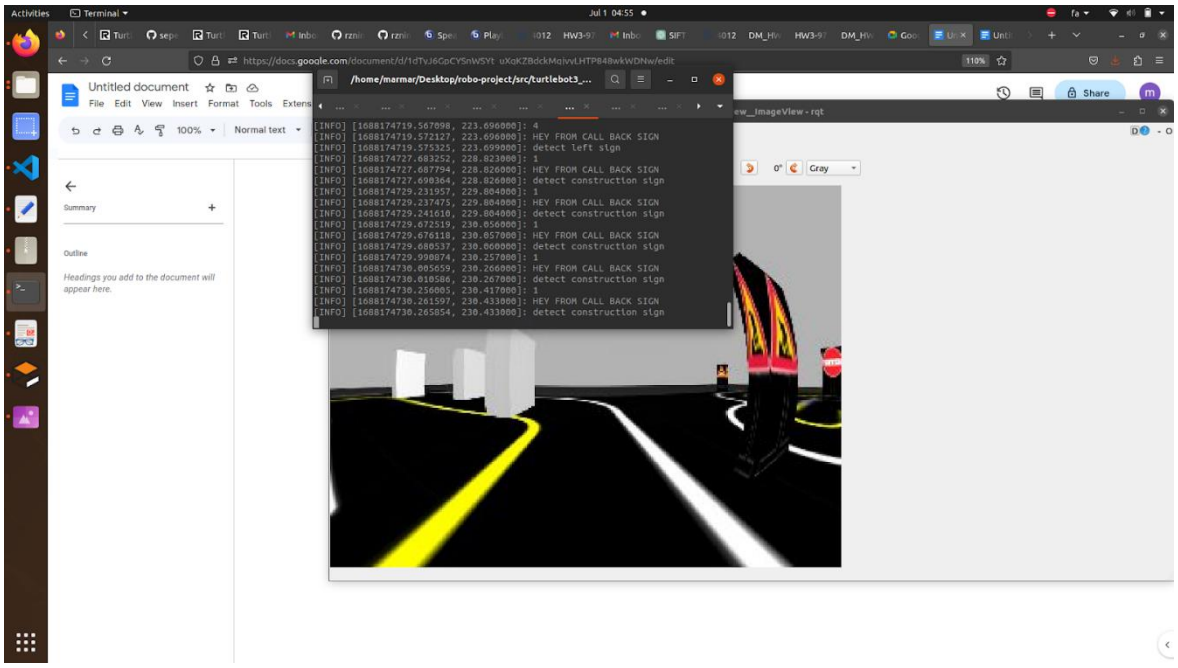
سوال ۲ – Lane Detection & Sign Detection

ما در اینجا دو قسمت برای پیاده سازی داریم . در قسمت اول باید پکیج ها را دانلود کنیم و برنامه را در حالت عادی ران کنیم و بعد از آن موقع برخورد با مانع باید کاری کنیم از آنها عبور کند. که این کار را انجام دادیم. برای قسمت رد شدن از مانع اینگونه عمل کردیم که با توجه به قرار گرفتن موانعِ به صورت خودکار از کنار مانع اول رد میشود. بعد از دیدن مانع دوم (ما توسط لیزر و چک کردن فاصله با مانع می دانیم چه زمان به مانع نزدیکیم.). سپس بعد از دیدن مانع دوم به سرعت به سمت چپ میرویم. و بعد ایستاده به ست عکس چرخیده مستقیم میرویم و دوباره به سمت چپ برمیگردیم. ضرایب کاملا به صورت تجربی به دست می آیند.در عکس زیر مشاهده میکنیم که موانع را رد میکند.



برای قسمت دوم تشخیص علامت ها نیز ما ابتدا در کلاس هایی که پسوندsign در اخرشان در پکیج detect داشتند را کمی تغییر دادیم. اینام های داخلشان رو طوری تغییر دادیم که همه ی آن علامت هایی که میخواهیم را در بر بگیرند. بعد ازآن روی تاپیک مربوطه در کلاس کنترل سابسکرایب کردیم. و میبینیم که میتواند ساین ها را تشخیص دهد.همچنین برای بهتر دیدن تابلو ها بعضی عکس ها را هم تغییر دادم.


/home/marmar/Desktop/robo-project/src/turtlebot3_...
* /rosversion: 1.16.0
NODES
/
control_lane (turtlebot3_autorace_driving/control_lane)
ROS_MASTER_URI=http://localhost:11311
process[control_lane-1]: started with pid [168937]
[INFO] [1688174607.182693, 147.168000]: 3
[INFO] [1688174607.186350, 147.171000]: HEY FROM CALL BACK SIGN
[INFO] [1688174607.188576, 147.172000]: detect intersection sign
[INFO] [1688174607.839523, 147.608000]: 3
[INFO] [1688174607.845787, 147.615000]: HEY FROM CALL BACK SIGN
[INFO] [1688174607.858404, 147.619000]: detect intersection sign
[INFO] [1688174609.060403, 148.549000]: 3
[INFO] [1688174609.062871, 148.553000]: HEY FROM CALL BACK SIGN
[INFO] [1688174609.064750, 148.556000]: detect intersection sign


Activities
Terminal
Untitled document
/home/marmar/Desktop/robo-project/src/turtlebot3_...
Gray