FACULTY OF ENGINEERING AND APPLIED SCIENCE
METE 4300U : Introduction to Mobile Robotics
Winter 2024

# ROS PROJECT

## DEVELOPMENT OF AN AUTONOMOUS VEHICLE SYSTEM

*MILESTONE #3 REPORT*

Course Instructor : Scott Nokleby
TA : Christopher Baird

Group # 9

| LAB GROUP MEMBERS | | | | |
|---|---|---|---|---|
| # | Surname | Name | ID | Signature |
| 1 | Lowrie | Jonathan | 100741369 | J.L |
| 2 | Isaeva | Yulia | 100743213 | Y.I |

**Table of Contents**



Lab 1:
* Detect orange,
* Swap yellow to white and white to yellow lane
* Swap lanes works
*swapped yellow for orange
*traffic cone detected nicely
*disable all red light detection
* editing detect_traffic _light

## Introduction

The Robot Operating System (ROS) is an open source index of libraries that aid in robot applications. These libraries include drivers and different algorithms that have been developed to make programming robots easier. This project uses ROS in order to program and control an autonomous mobile robot, the TurtleBot3 burger. The main objective for this project and the robot is to be able to autonomously navigate a road network and perform various tasks [1]. These tasks are separated into three milestones over the course of this project. After each milestone a demonstration(demo) will be held to validate the completion of the requirements in each milestone. The milestones are:

- **Milestone #1** : Autonomous Driving
- **Milestone #2** : Traffic Signal Detection
- **Milestone #3** : Obstacle Detection and Avoidance

This report solely focuses on Milestone #3 :Obstacle Detection and Avoidance with an additional explanation of Milestone #1. As autonomous driving is used within the 3rd milestone it is necessary to explain the previous work from Milestone #1. Likewise, the traffic signal system used in Milestone #2 is not a requirement in Milestone #3 Therefore its section is omitted from this report.

### Milestone 1: Autonomous Driving

For Phase A, the robot had to be able to autonomously navigate the road network which is shown in *fig 1 & 2*.The robot, following Canada traffic laws, must drive in the right lane of the course in either direction from its starting point. The square intersection in the road network also poses a challenge in lane detection as when the robot must cross through it the yellow line disappears. This milestone requires the robot to complete these tasks and challenges quickly and safely. The demo for this milestone requires that the robot autonomously drive in its designated lane from its starting point and return to the same starting point after completing both loops.

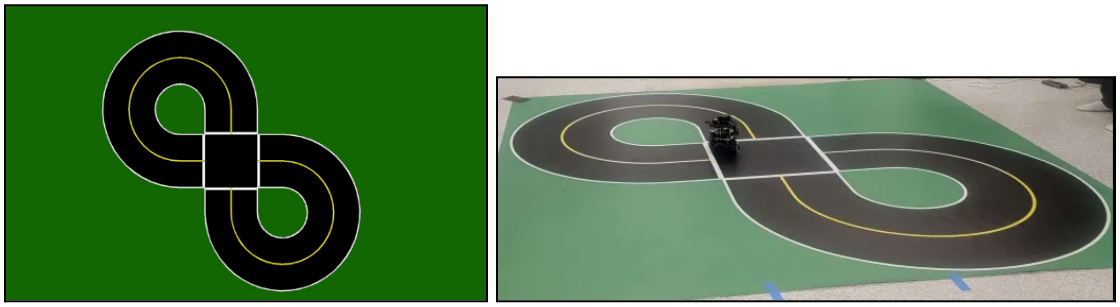


**Fig 1 & 2.** Road Network BluePrint & Lab Road Network with Turtlebot3 [1]

**Milestone 2: Traffic Light Detection**

As the last milestone does not include the traffic light system the details of this milestone are not discussed when in relation to milestone #3.

**Milestone 3: Obstacle Detection and Avoidance**

For this milestone, orange traffic cones are placed on the road network. This is to simulate a construction environment similar to real-life situations. The cones are placed on the right side of the track at both loops. The objective of the robot is to detect the orange traffic cone and then switch lanes and drive until the traffic cones are no longer detected , which then the robot should switch back to the original lane. The robot is required to make a full continuous run of the track in order to complete the milestone. The track and orange traffic cones can be viewed in figure *3*.

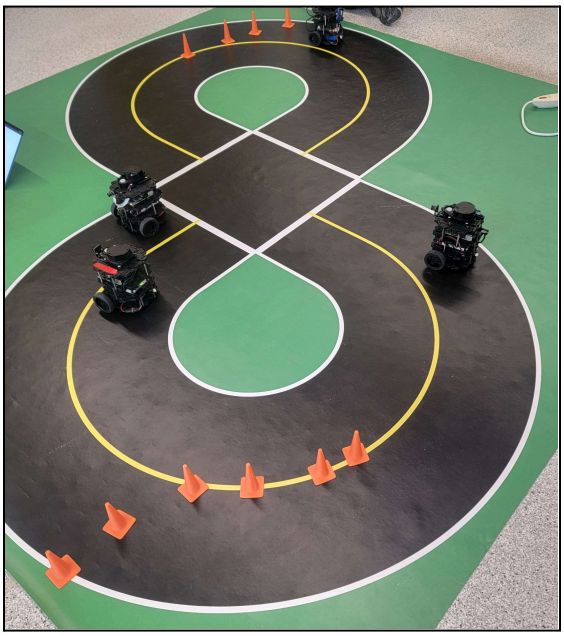


**Fig 3.** Road Network with Construction Cone Placements

## Milestone Requirements & Engineering Specifications

### Requirements

In order to successfully complete Milestone #3 there were a variety of requirements that the robot had to pass. If one or more of these requirements are not met during the demo it will hinder the overall evaluation and performance of the milestone. The requirements for Milestone #3 are separated into functional and physical requirements. Functional requirements represent the completion of tasks necessary for the milestone, while the physical requirements are the components necessary for the robot to complete the functional requirements.

**Table 1. Functional Requirements for Milestone 3**

| | |
|---|---|
| F1 | Robot must be able to identify the orange traffic cones |
| F2 | Robot must be able to switch driving lanes to avoid the cone obstacles |
| F3 | Robot must be able to switch back to the right lane after obstacles have passed |
| F4 | Robot must be able to cross the intersection quickly and safely |
| F5 | Robot must be able to travel in either direction |

**Table 2. Physical Requirements for Milestone 3**

| | |
|---|---|
| P1 | Robot must have an operational & focused camera |
| P2 | Robot must have a charged Li-Po battery for continuous power supply |
| P3 | Robot must be connected to a network |
| P4 | Traffic cones must be placed onto the track |

These requirements serve as a necessary guideline that will aid in completing Milestone #3 efficiently. The turtlebot is shown in *figure 4* below as an example of the complete assembly for the physical requirements.

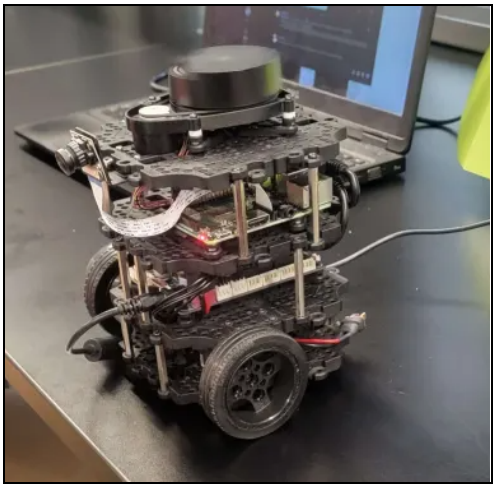


**Fig 4.** TurtleBot3 used in Milestone

**Engineering Specifications**

Following the functional and physical milestone requirements, the engineering specifications for the robot must also be met in order to complete the milestone. The engineering specifications include detailed technical criteria that are critical to the operation of the system/robot. *Table 3* lists the engineering specifications of the robot in order to complete the requirements that were listed in *Table 1&2*.

**Table 3. Engineering Specifications for Milestone 3**

| | |
|---|---|
| E1 | Must have an operational PC |
| E2 | Robot & PC must be configured to the appropriate network |
| E3 | Robot should not stop for more than 1 minute |
| E4 | Cones/obstacles should be detected at an appropriate distance |
| E5 | Lane transition should be smooth |
| E6 | Noise from detection method should be minimalized |

## Background Research

To obtain a better understanding of the operational dynamic of our robot system concerning obstacle detection and avoidance, a comprehensive exploration of the existing ROS packages and detection methods was undertaken. However, it should be noted that for the turtlebot the AutoRace package has already been decided as the package that will be utilized for the completion of this milestone. This is because of its straightforward integration with the TurtleBot. The research conducted is for the main purpose of exploring the different techniques that are available for obstacle detection and avoidance.

### Methods | Obstacle Detection

Autonomous Robots have a variety of available methods for obstacle detection. A few methods will be outlined for a better understanding of alternate paths that could be taken to complete this objective. All of these methods are in relation to the hardware that is available to the robot for obstacle detection.

LiDAR(Light Detection and Ranging): lidar is a remote sensing method that uses light in the form of a pulsed laser to measure ranges from objects in the environment. Some LiDARS have the capability of three dimensionally mapping its environment for more extensive object and environment detection.Since this sensor is able to map all types of environments with accuracy and precision, most Lidar systems are more costly than other methods of obstacle detection. [1]

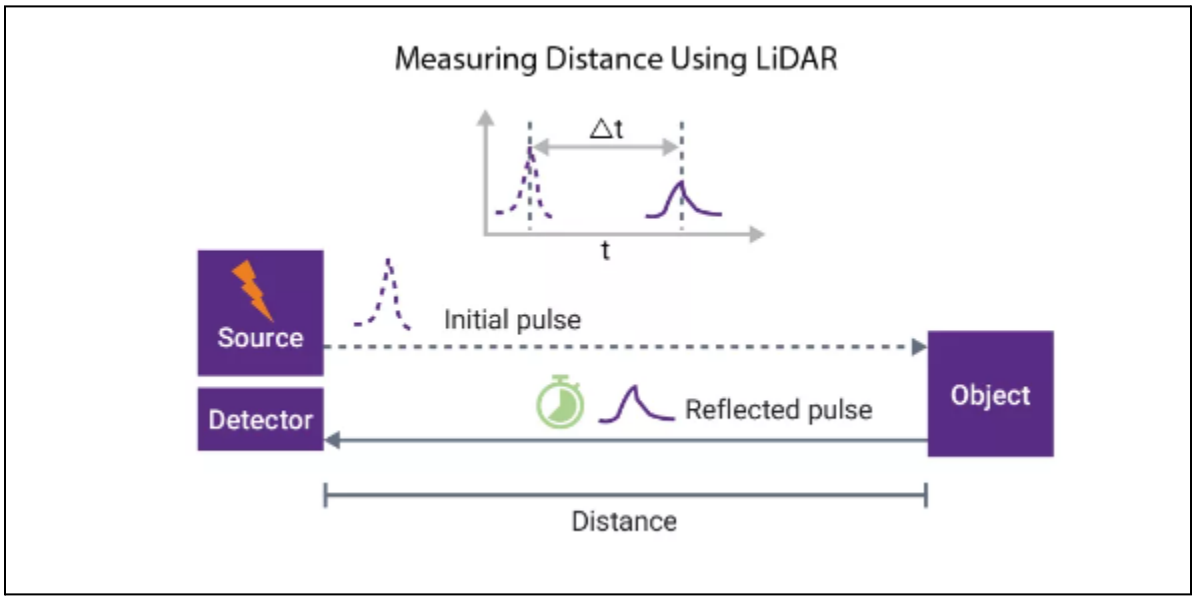



**Fig 5.** LiDAR Decomposition Diagram [1]

Depth Cameras : These cameras capture the distance information for each pixel, creating a depth map of the environment. This camera can automatically detect the presence of any object nearby. This allows the robot to automatically make decisions due to the surrounding environment. There

are a variety of depth technologies available (Stereo vision, time of flight, structured light) all for different and useful applications [2].

**Fig 6.** 3D Time of Flight Camera Example [2]

Ultrasonic Sensors : Similar to lidar sensors these emit sound waves instead of light that measure the time it takes for the echo to return, determining the distance to an obstacle. These can be bought at a range of values depending on the accuracy of the data. A significant limitation to this method is the limited field of view and minimal interference can disrupt the accuracy of the sensor's data.

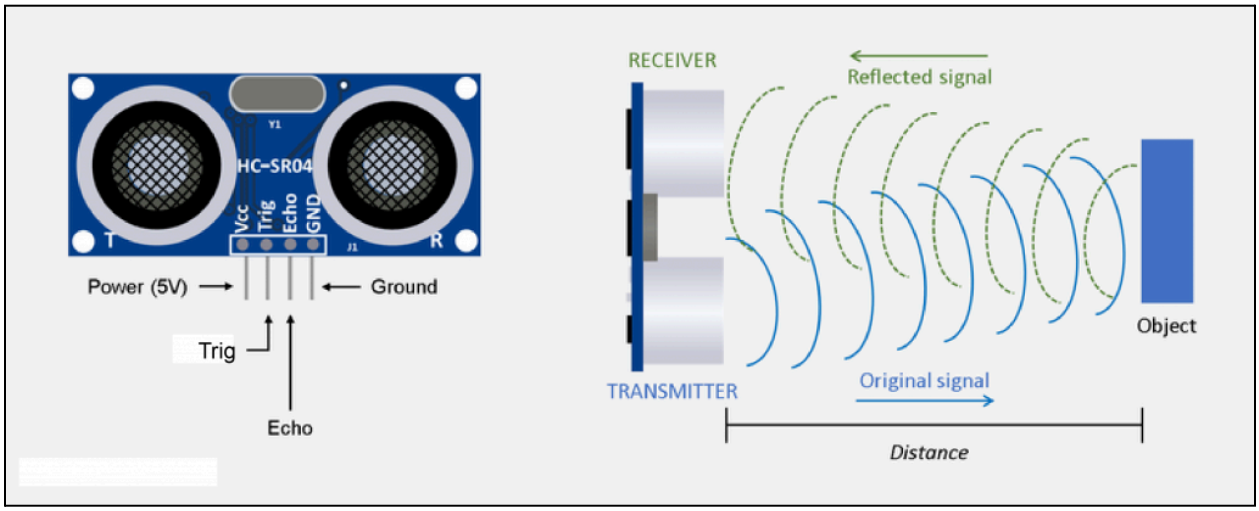



**Fig 7.** Ultrasonic Sensor Decomposition [3]

Cameras: Standard cameras with the included application of computer vision algorithms can capture the environment and detect objects in its field of view. This method is versatile and budget friendly but requires a certain amount of computing power and can be challenged with different lighting conditions. This is the method of detection the TurtleBots will be using from this milestone.

**Methods | Obstacle Avoidance**

Obstacle avoidance relies heavily on the ROS package used in autonomous mobile robots. There are a variety of packages available, two will be discussed in this section that can be integrated with the Turtlebot robots.

obstacle-avoidance-ROS [4] : This package utilizes the turtlebots LiDAR sensor to process the environment data and find the clearest path forward. It essentially steers the robot into the right direction by finding the direction where the distance to the nearest object exceeded a defined threshold. When more than one path is available the algorithm will choose the one with the least required movement. This package node can be implemented into another package to utilize it.

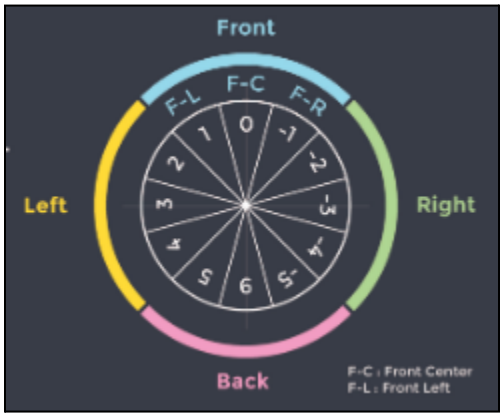



**Fig 8.** LiDAR Coverage

Reactive_assistance [5] : This package contains a "gap-based" algorithm approach for obstacle avoidance. This method allows the robot to navigate through spaces and avoid obstacles. Thus, this algorithm is better suited for larger scale environments as the robot would find a way through the obstacle instead of around it. This method also uses an implementation of a LiDAR sensor and can be used with the TurtleBot robot.

## Functional Decomposition of Developed System and Software Design

As previously mentioned, for this milestone the *turtlebot3_autorace_2020* metapackage will be used to complete the functional requirements F3 & F4. The noetic version was installed to ensure that it corresponded to the turtlebot. On Top of the AutoRace package additional dependent packages were also installed. This section of the report outlines general ROS package structure and functionality as well as the AutoRace2020 metapackage.

### ROS

ROS, or Robot Operating System, is a large framework designed to make development of robot software easier and open sourced. ROS is not a full operating system but acts in the middle of communication between the hardware and software of a robotics system. The architecture uses nodes and topics that can be subscribed or published from to perform various tasks.

### ROS Architecture

The architecture of ROS has many fundamental components, each playing a different but significant role. ROS is organized into packages, these packages contain libraries and other necessary executable files that will enable the robot to function appropriately. Packages contain a set of nodes that work together to accomplish the specific task.

Nodes: nodes are individual programs that perform a specific task within that package. They can be written in various languages but for the nodes in this milestone they are written in Python. Nodes communicate with each other by publishing or subscribing messages to topics or each other.

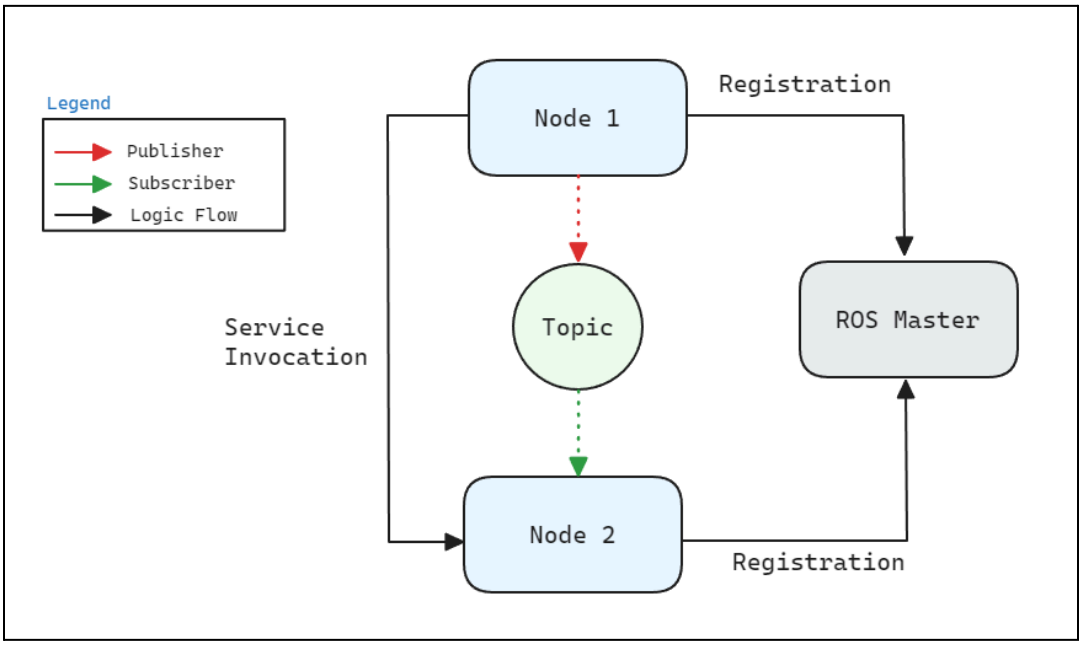



**Fig 9.** ROS Node Communication

Topics: Topics are communication channels where nodes can exchange information. They act as a placeholder where different nodes can subscribe or publish to that same topic.

Publishers & Subscribers: Publishers are nodes that produce information to a topic. They contain data or sensor readings from the robot, allowing any other node to subscribe to that information. Subscribers are the opposite in where they subscribe to topics of interest or data needed from publishers and use that data in their algorithms. This method provides simple communication inside ROS packages that can be altered to specific tasks and needs of the robot.

**ROS Package | AutoRace2020 MetaPackage**

This metapackage is designed specifically for the Turtlebot3. It provides tools for autonomous navigation, including traffic light detection and lane detection. The metapackage also includes various other ROS packages that handle different aspects of the robot's functionality.

Specifically, the *detect_traffic_light* node is designed to detect traffic lights (red,yellow and green) from a camera feed; this node was edited in order to detect only the orange from the traffic cones. The node functionality is outlined below with a full explanation of the autonomous driving and obstacle detection and avoidance architectures.

Autonomous Driving Decomposition

This section goes over the autonomous driving subsection of this milestone. Though its was completed in Milestone #1, the package and nodes control_lane and detect_lane were still used to complete Milestone #3. The system and network architecture for this section is displayed in Fig.10.

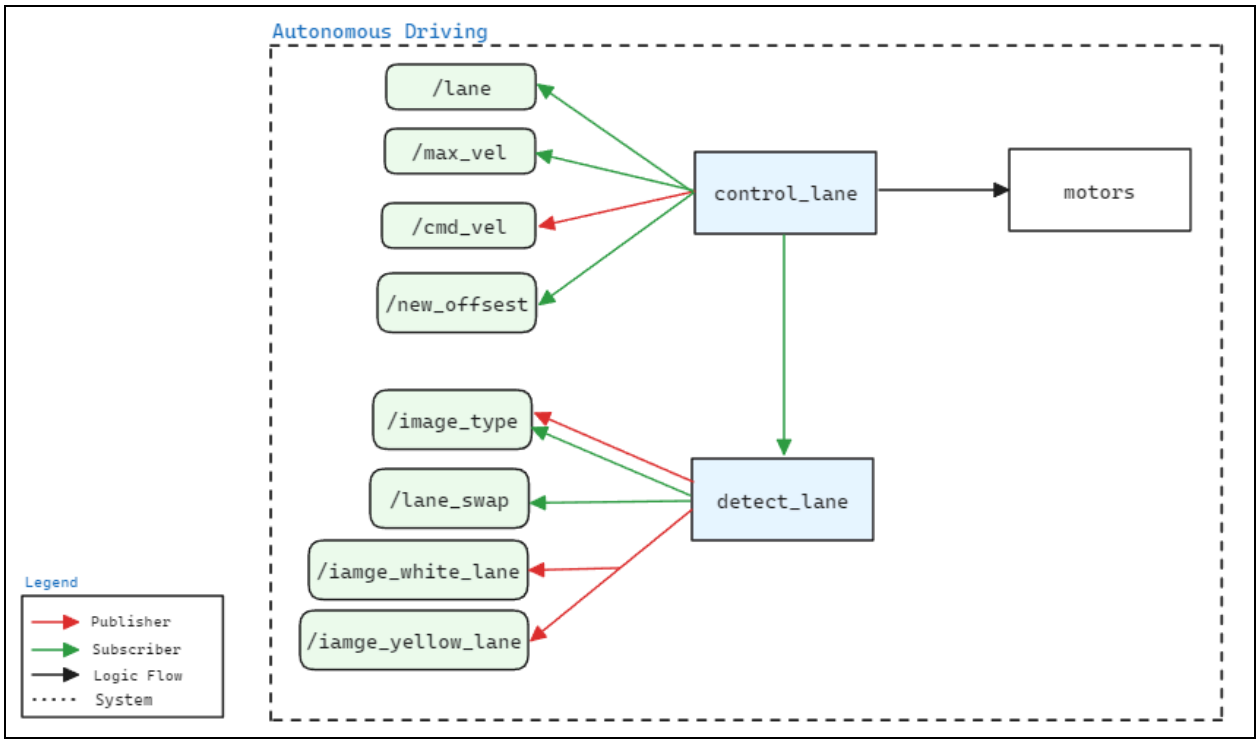



**Fig 10.** Autonomous Driving Subsystem Decomposition

The control logic for this ensures that a path is generated in the right lane and the mac_vel controls the movement of the robot. The lane parameters were previously configured and were not changed for this milestone.

Traffic Light Decomposition

This subsystem is the main obstacle detection and avoidance system. By using an altered detect_traffic_light node, the robot is able to detect the orange traffic cone and use an offset and lane swap to change course and avoid the construction cone. The architecture is outlined in Fig 11.

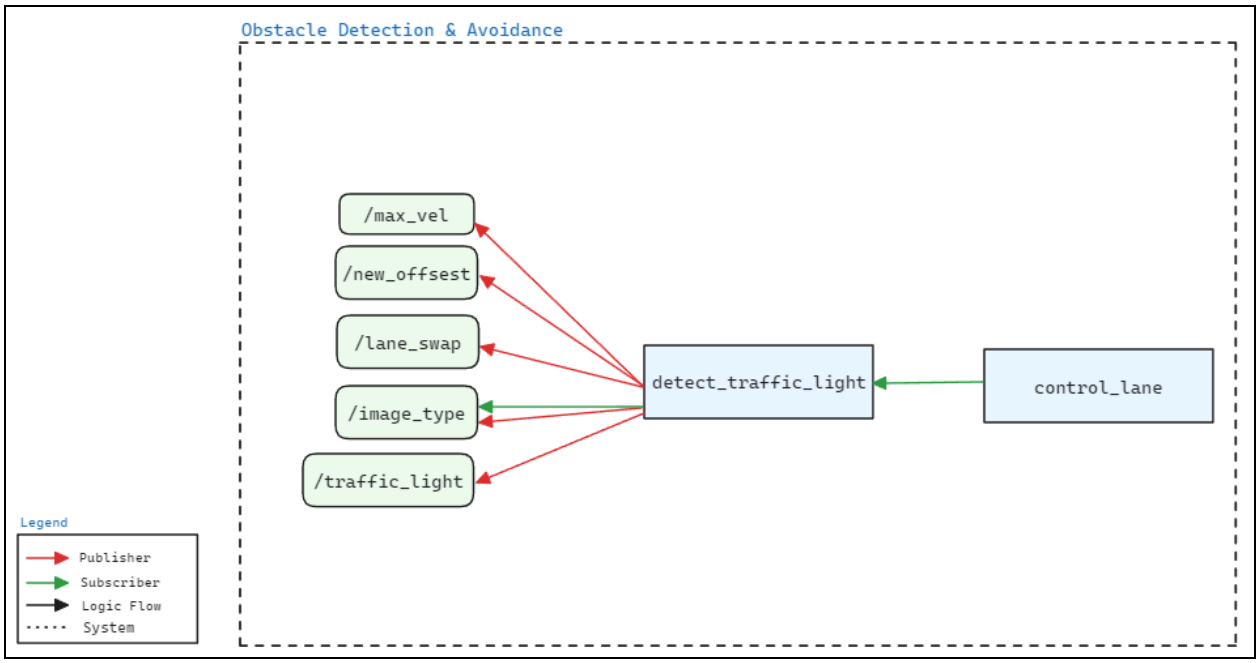



**Fig 11.** Obstacle Detection & Avoidance Subsystem Decomposition

Complete Milestone Decomposition

This section outlines the whole milestone decomposition in logic and system architecture with the following figures. However the architecture diagram in Fig 12 includes the topics that were utilized the most in this milestone. Fig 13. Outline the logic that was needed for the robot to complete the milestone successfully.

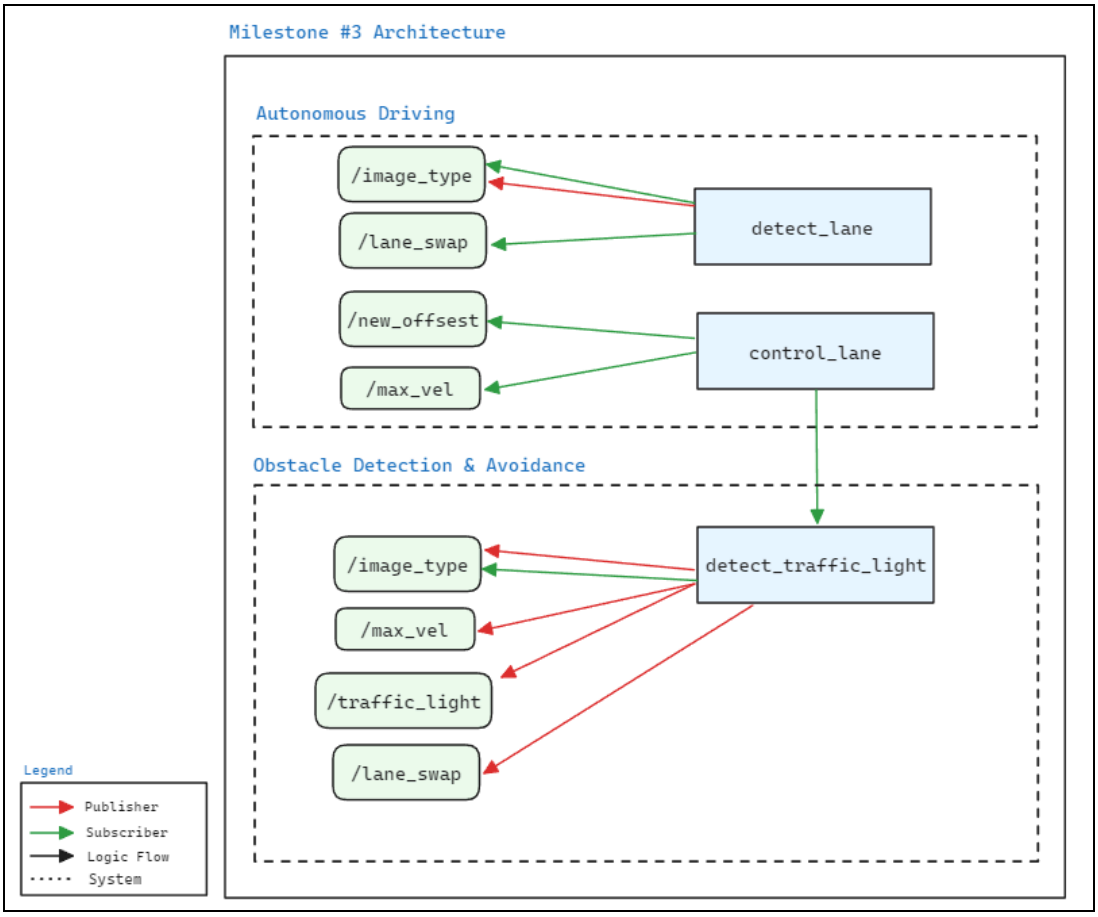



**Fig 12**. Milestone #3 Summarized System Architecture

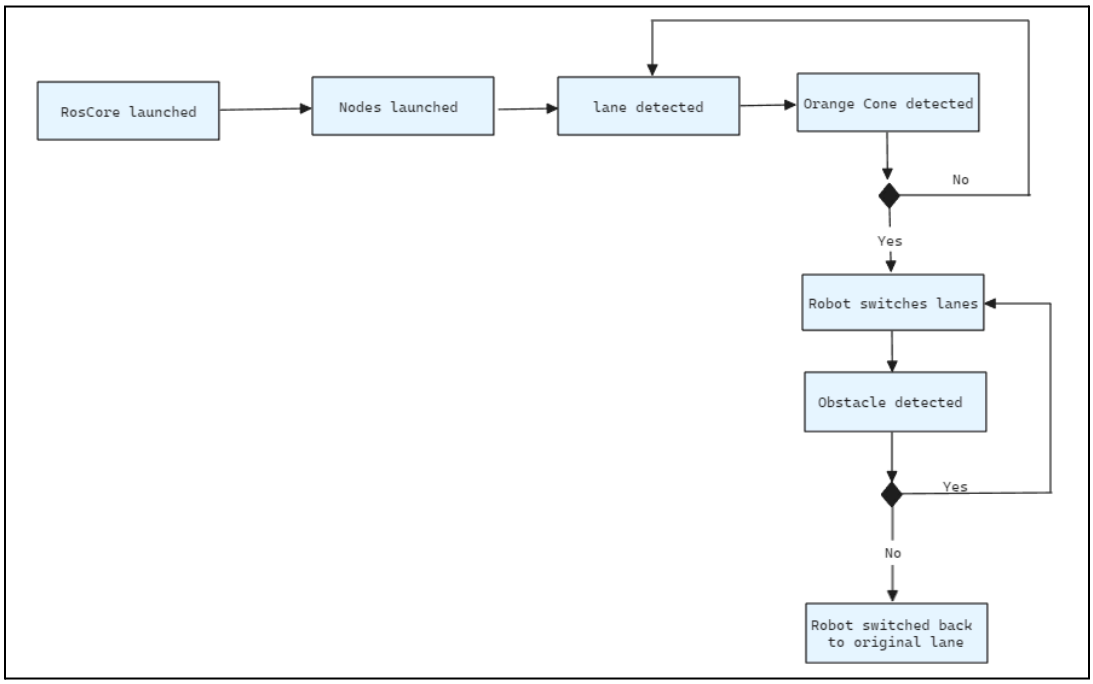



**Fig 13.** Milestone System Logic

Package & Node Edits

A summary of all the changes made in the package can be found below in *Table 4*, where each modified node is outlined including its change. The full edited code for the nodes that were modified can be found in the appendix of this report.

**Table 4. Summary of Changes In AutoRace Package**

| **Node: detect_traffic_light** | |
|---|---|
| Change Made: | #status3 = self.fnFindCircleOfTrafficLight(cv_image_mask, 'red')<br># if status3 == 3:<br># rospy.loginfo("detect RED")<br># self.red_count += 1<br># self.pub_max_vel.publish(0.0) |
| | #self.pub_max_vel.publish(0.05) |
| Reason for Change : | Commented out the red detection and green max_velocity to only look for the yellow light which was adjusted to detect the orange cones.This change makes sure that red wont be detected and if green was detected no motor control would be published. |
| **Node: detect_traffic_light** | |
| Change Made: | rospy.loginfo("detect YELLOW")<br>self.yellow_count += 1<br>if self.cone_seen == False:<br>self.cone_seen = True<br>if self.yellow_percent > 10:<br>self.pub_lane_swap.publish(True)<br>self.pub_new_offset.publish(400)<br>current_offset = 400<br>elif self.yellow_percent > 8:<br>self.pub_new_offset.publish(50)<br>elif self.yellow_percent > 7:<br>self.pub_new_offset.publish(100)<br>elif self.yellow_percent > 6:<br>self.pub_new_offset.publish(150)<br>elif self.yellow_percent > 5:<br>self.pub_new_offset.publish(200)<br>elif self.yellow_percent > 4:<br>self.pub_new_offset.publish(250)<br>elif self.yellow_percent > 3:<br>self.pub_new_offset.publish(300)<br>elif self.yellow_percent > 2:<br>self.pub_new_offset.publish(350)<br>else: |

| | |
|---|---|
| | self.pub_new_offset.publish(400)<br><br>self.last_cone_seen = 0<br>else:<br>self.yellow_count = 0<br>if self.cone_seen == True:<br>self.last_cone_seen += 1 |
| Reason for Change : | To check the Percentage of the screen that is the orange of the cone before changing where it should be in the lane |
| **Node: detect_traffic_light** | |
| Change Made: | if self.last_cone_seen > 100:<br>if self.cone_seen == true:<br>current_offset = 0<br>self.cone_seen = False<br>self.pub_lane_swap.publish(False)<br>if current_offset < 400:<br>current_offset = current_offset + 1<br>self.pub_new_offset.publish(current_offset) |
| Reason for Change : | To reset the system after it has not seen a cone for a set time |
| **Node: control_lane** | |
| Change Made: | error = center - self.offset |
| Reason for Change : | This allows for the system to change where it is driving within the lane |

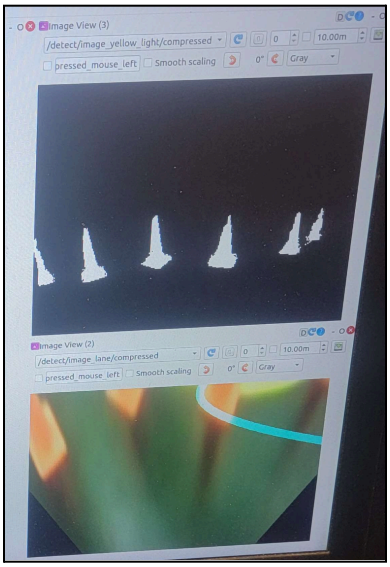
## Test plan

In order to validate the developed system that was previously outlined, a test plan was created and followed to ensure the full completion of the milestone. The test plan uses four sections interchangeably to perfect the robot for completing the milestone and demo.These four sections are:

- Calibrating Traffic Cone Detection
- Obstacle Avoidance
- Network Configuration
- Battery Optimization

Each section is critical to the robot's functionality. These four sections were carried out during lab time and personal time on the road network. Both the lab network and the extra road network were used for testing. It should be noted that the extra road network that was available outside of lab time had a different lighting environment which extended the calibration time. The four sections of the test plan are outlined below in *Table 5*.

**Table 5. Test Plan for Milestone #2**

| **Calibrating Traffic Cone Detection** | |
|---|---|
| Problem: | The traffic cones need to be detected in order for the robot to avoid the obstacle and switch lanes. The traffic cones are orange. |
| Solution: | In the detect_traffic_lane node the hue parameters for yellow were adjusted to detect the orange color. |
| Working Obstacle Detection |  **Fig 14.** Cone Detection |

| | |
|---|---|
| Final Parameter Values | hue_red_l = 0<br>hue_red_h = 26<br>saturation_red_l = 239<br>saturation_red_h = 255<br>lightness_red_l = 123<br>lightness_red_h = 250<br>hue_yellow_l = 0<br>hue_yellow_h = 21<br>saturation_yellow_l = 110<br>saturation_yellow_h = 228<br>lightness_yellow_l = 123<br>lightness_yellow_h = 255<br>hue_green_l = 40<br>hue_green_h = 113<br>saturation_green_l = 210<br>saturation_green_h = 255<br>lightness_green_l = 131<br>lightness_green_h = 255 |
| Results: | The robot was able to efficiently detect the traffic cones and did not detect background noise. The adjustment was successful. |
| **Obstacle Avoidance** | |
| Problem: | The robot is required to switch lanes to avoid the obstacles and then switch back to the original lane once the obstacles are no longer detected, marking the end of the construction block. |
| Solution: | The nodes were edited to adjust the offset with respect to how much obstacle is detected . This offset would slowly switch lanes to avoid the construction cones. The full edits can be viewed in Table 4. |
| Results: | Regrettably, though the obstacles were detected the offset was inconsistent. In some cases it would turn and resume driving, but the majority of the test runs were unsuccessful. |
| **Network Calibration** | |
| Problem: | The laboratory in which the road network was placed, has its own network titled ‘Mobile Robotics’ that is for the specific use of this course. However, as more turtlebots are connected to the same network it was observed that many obtained connectivity issues and slower processing time. This caused disruptive issues in our early testing plans as it was difficult to re-calibrate the lane detection when the *rqt* image windows would not update fast enough or at all. |

| | |
|---|---|
| Solution: | Using a personal router instead of directly connecting to the provided network. |
| Results: | By using the personal router instead of directly connecting to the provided network, the robot was able to run tasks without lag or any other interference. This enabled us to calibrate the lane detection and camera parameters more accurately, thus, given us a better chance to complete the milestone. |
| **Battery Optimization** | |
| Problem: | The Li-Po battery has a duration of approximately 2 hours when running continuously.When doing long tests on the road network it was a common issue that the battery would run out before the testing was finished. |
| Solution: | The battery was charged fully before each test day and time management was incorporated to make sure that it always had a full charge when testing. |
| Results: | The battery optimization plan worked well and testing went smoother. There were still a few instances when it ran out sooner than anticipated but other batteries from adjacent groups loaned their batteries when not in use. |

## Final Results

In order to validate the test plan in the previous section, the robot had to complete the demo run perfectly. The below table discusses what occurred during the demo day and what was changed.

**Table 6. DEMO Analyzation**

| **DEMO** | |
|---|---|
| Pre-Demo: | Continued tuning the obstacle detection as the robot was still inconsistently switching lanes before the demo. The robot would proceed to turn too much or not at all when detecting the obstacle. |
| **Final Results** | |
| Demo Attempt #1 | The cone detection was successful, however, switching lanes was unsuccessful and the robot proceeded to run into the cones. |
| Demo Attempt #2 | The second attempt was similar to the first, lane detection and obstacle detection were successful but the robot once more ran into the cones. |

## Final Results

Despite the successfulness of the obstacle detection, the obstacle avoidance object faced some challenges in regards to lane switching leading to collisions with the cones during both attempts and through testing. Though continuous tuning of the obstacle avoidance algorithms was done by attempting to change the offset according to the amount of obstacles detected, the robot was unsuccessful in completing the full objectives of this milestone.

## Conclusions and Future Work

The results from the final demo attempts, highlights the importance of refining obstacle detection to achieve reliable autonomous navigation that ensures safety. Though not all the necessary objectives of this milestone were completed, many knowledgeable lessons were learned about modifying ROS nodes and writing code to change those nodes in order to complete similar tasks. In the future, though the milestones have ended, the knowledge gained from this project will continue to be useful.

## References


1 "What is LiDAR and How Does it Work? | Synopsys." https://www.synopsys.com/glossary/what-is-lidar.html

2 P. Kumar and P. Kumar, "What are depth-sensing cameras? How do they work?," *E-con Systems*, Apr. 04, 2024. https://www.e-consystems.com/blog/camera/technology/what-are-depth-sensing-cameras-how-do-they-work/

3 "Micro bit Lesson — Using the Ultrasonic Module « osoyoo.com," Sep. 18, 2018. https://osoyoo.com/2018/09/18/micro-bit-lesson-using-the-ultrasonic-module/

4 Rad-Hi, "GitHub - Rad-hi/Obstacle-Avoidance-ROS: A ROS node that allows for a naive obstacle avoidance behavior based on laser scans from a Lidar (Gazebo simulation).," *GitHub*. https://github.com/Rad-hi/Obstacle-Avoidance-ROS

5 Mazrk, "GitHub - mazrk7/reactive_assistance: A ROS package of a reactive obstacle avoidance method for navigation of mobile robots.," *GitHub*. https://github.com/mazrk7/reactive_assistance


## Appendix

Commented code

**Detect lane**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

################################################################################
####
# Copyright 2018 ROBOTIS CO., LTD.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
################################################################################
```

```
####

# Authors: Leon Jung, Gilbert, Ashe Kim, Special Thanks : Roger Sacchelli

import rospy
import numpy as np
import cv2
from cv_bridge import CvBridge
from std_msgs.msg import UInt8, Float64, Bool
from sensor_msgs.msg import Image, CompressedImage
from dynamic_reconfigure.server import Server
from turtlebot3_autorace_detect.cfg import DetectLaneParamsConfig

class DetectLane():
    def __init__(self):
        self.hue_white_l = rospy.get_param("~detect/lane/white/hue_l", 0)
        self.hue_white_h = rospy.get_param("~detect/lane/white/hue_h", 179)
        self.saturation_white_l = rospy.get_param("~detect/lane/white/saturation_l", 0)
        self.saturation_white_h = rospy.get_param("~detect/lane/white/saturation_h", 70)
        self.lightness_white_l = rospy.get_param("~detect/lane/white/lightness_l", 105)
        self.lightness_white_h = rospy.get_param("~detect/lane/white/lightness_h", 255)

        self.hue_yellow_l = rospy.get_param("~detect/lane/yellow/hue_l", 10)
        self.hue_yellow_h = rospy.get_param("~detect/lane/yellow/hue_h", 127)
        self.saturation_yellow_l = rospy.get_param("~detect/lane/yellow/saturation_l", 70)
        self.saturation_yellow_h = rospy.get_param("~detect/lane/yellow/saturation_h", 255)
        self.lightness_yellow_l = rospy.get_param("~detect/lane/yellow/lightness_l", 95)
        self.lightness_yellow_h = rospy.get_param("~detect/lane/yellow/lightness_h", 255)

        self.is_calibration_mode = rospy.get_param("~is_detection_calibration_mode", False)
        if self.is_calibration_mode == True:
            srv_detect_lane = Server(DetectLaneParamsConfig, self.cbGetDetectLaneParam)

        self.sub_image_type = "raw"         # you can choose image type "compressed", "raw"
        self.pub_image_type = "compressed"  # you can choose image type "compressed", "raw"
        self.sub_lane_swap = rospy.Subscriber('/control/swap_lane', Bool, self.swap_lanes,
queue_size =1)
        if self.sub_image_type == "compressed":
            # subscribes compressed image
            self.sub_image_original = rospy.Subscriber('/detect/image_input/compressed',
CompressedImage, self.cbFindLane, queue_size = 1)
        elif self.sub_image_type == "raw":
            # subscribes raw image
            self.sub_image_original = rospy.Subscriber('/detect/image_input', Image,
self.cbFindLane, queue_size = 1)
```

```
        if self.pub_image_type == "compressed":
            # publishes lane image in compressed type
            self.pub_image_lane = rospy.Publisher('/detect/image_output/compressed',
CompressedImage, queue_size = 1)
        elif self.pub_image_type == "raw":
            # publishes lane image in raw type
            self.pub_image_lane = rospy.Publisher('/detect/image_output', Image, queue_size = 1)

        if self.is_calibration_mode == True:
            if self.pub_image_type == "compressed":
                # publishes lane image in compressed type
                self.pub_image_white_lane =
rospy.Publisher('/detect/image_output_sub1/compressed', CompressedImage, queue_size = 1)
                self.pub_image_yellow_lane =
rospy.Publisher('/detect/image_output_sub2/compressed', CompressedImage, queue_size = 1)
            elif self.pub_image_type == "raw":
                # publishes lane image in raw type
                self.pub_image_white_lane = rospy.Publisher('/detect/image_output_sub1', Image,
queue_size = 1)
                self.pub_image_yellow_lane = rospy.Publisher('/detect/image_output_sub2', Image,
queue_size = 1)

        self.pub_lane = rospy.Publisher('/detect/lane', Float64, queue_size = 1)

        # subscribes state : yellow line reliability
        self.pub_yellow_line_reliability = rospy.Publisher('/detect/yellow_line_reliability', UInt8,
queue_size=1)

        # subscribes state : white line reliability
        self.pub_white_line_reliability = rospy.Publisher('/detect/white_line_reliability', UInt8,
queue_size=1)

        self.cvBridge = CvBridge()

        self.counter = 1

        self.left_lane = False


        self.window_width = 1000.
        self.window_height = 600.

        self.reliability_white_line = 100
        self.reliability_yellow_line = 100

    def cbGetDetectLaneParam(self, config, level):
```

```
    rospy.loginfo("[Detect Lane] Detect Lane Calibration Parameter reconfigured to")
    rospy.loginfo("hue_white_l : %d", config.hue_white_l)
    rospy.loginfo("hue_white_h : %d", config.hue_white_h)
    rospy.loginfo("saturation_white_l : %d", config.saturation_white_l)
    rospy.loginfo("saturation_white_h : %d", config.saturation_white_h)
    rospy.loginfo("lightness_white_l : %d", config.lightness_white_l)
    rospy.loginfo("lightness_white_h : %d", config.lightness_white_h)
    rospy.loginfo("hue_yellow_l : %d", config.hue_yellow_l)
    rospy.loginfo("hue_yellow_h : %d", config.hue_yellow_h)
    rospy.loginfo("saturation_yellow_l : %d", config.saturation_yellow_l)
    rospy.loginfo("saturation_yellow_h : %d", config.saturation_yellow_h)
    rospy.loginfo("lightness_yellow_l : %d", config.lightness_yellow_l)
    rospy.loginfo("lightness_yellow_h : %d", config.lightness_yellow_h)

    self.hue_white_l = config.hue_white_l
    self.hue_white_h = config.hue_white_h
    self.saturation_white_l = config.saturation_white_l
    self.saturation_white_h = config.saturation_white_h
    self.lightness_white_l = config.lightness_white_l
    self.lightness_white_h = config.lightness_white_h

    self.hue_yellow_l = config.hue_yellow_l
    self.hue_yellow_h = config.hue_yellow_h
    self.saturation_yellow_l = config.saturation_yellow_l
    self.saturation_yellow_h = config.saturation_yellow_h
    self.lightness_yellow_l = config.lightness_yellow_l
    self.lightness_yellow_h = config.lightness_yellow_h

    return config

def cbFindLane(self, image_msg):
    # Change the frame rate by yourself. Now, it is set to 1/3 (10fps).
    # Unappropriate value of frame rate may cause huge delay on entire recognition process.
    # This is up to your computer's operating power.
    if self.counter % 3 != 0:
        self.counter += 1
        return
    else:
        self.counter = 1

    if self.sub_image_type == "compressed":
        #converting compressed image to opencv image
        np_arr = np.frombuffer(image_msg.data, np.uint8)
        cv_image = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
    elif self.sub_image_type == "raw":
        cv_image = self.cvBridge.imgmsg_to_cv2(image_msg, "bgr8")
```

```
    # find White and Yellow Lanes
    if self.left_lane == False:
       white_fraction, cv_white_lane = self.maskWhiteLane(cv_image)
       yellow_fraction, cv_yellow_lane = self.maskYellowLane(cv_image)
    else:
       yellow_fraction, cv_yellow_lane = self.maskWhiteLane(cv_image)
       white_fraction, cv_white_lane = self.maskYellowLane(cv_image)


    try:
       if yellow_fraction > 3000:
          self.left_fitx, self.left_fit = self.fit_from_lines(self.left_fit, cv_yellow_lane)
          self.mov_avg_left = np.append(self.mov_avg_left,np.array([self.left_fit]), axis=0)

       if white_fraction > 3000:
          self.right_fitx, self.right_fit = self.fit_from_lines(self.right_fit, cv_white_lane)
          self.mov_avg_right = np.append(self.mov_avg_right,np.array([self.right_fit]),
axis=0)
    except:
       if yellow_fraction > 3000:
          self.left_fitx, self.left_fit = self.sliding_windown(cv_yellow_lane, 'left')
          self.mov_avg_left = np.array([self.left_fit])

       if white_fraction > 3000:
          self.right_fitx, self.right_fit = self.sliding_windown(cv_white_lane, 'right')
          self.mov_avg_right = np.array([self.right_fit])

    MOV_AVG_LENGTH = 5

    self.left_fit = np.array([np.mean(self.mov_avg_left[::-1][:, 0][0:MOV_AVG_LENGTH]),
                  np.mean(self.mov_avg_left[::-1][:, 1][0:MOV_AVG_LENGTH]),
                  np.mean(self.mov_avg_left[::-1][:, 2][0:MOV_AVG_LENGTH])])
    self.right_fit = np.array([np.mean(self.mov_avg_right[::-1][:,
0][0:MOV_AVG_LENGTH]),
                  np.mean(self.mov_avg_right[::-1][:, 1][0:MOV_AVG_LENGTH]),
                  np.mean(self.mov_avg_right[::-1][:, 2][0:MOV_AVG_LENGTH])])

    if self.mov_avg_left.shape[0] > 1000:
       self.mov_avg_left = self.mov_avg_left[0:MOV_AVG_LENGTH]

    if self.mov_avg_right.shape[0] > 1000:
       self.mov_avg_right = self.mov_avg_right[0:MOV_AVG_LENGTH]

    self.make_lane(cv_image, white_fraction, yellow_fraction)
```

```
def swap_lanes(self, lane):
    if lane.data == True:
        self.left_lane = True
    else:
        self.left_lane = False

def maskWhiteLane(self, image):
    # Convert BGR to HSV
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    Hue_l = self.hue_white_l
    Hue_h = self.hue_white_h
    Saturation_l = self.saturation_white_l
    Saturation_h = self.saturation_white_h
    Lightness_l = self.lightness_white_l
    Lightness_h = self.lightness_white_h

    # define range of white color in HSV
    lower_white = np.array([Hue_l, Saturation_l, Lightness_l])
    upper_white = np.array([Hue_h, Saturation_h, Lightness_h])

    # Threshold the HSV image to get only white colors
    mask = cv2.inRange(hsv, lower_white, upper_white)

    # Bitwise-AND mask and original image
    res = cv2.bitwise_and(image, image, mask = mask)

    fraction_num = np.count_nonzero(mask)

    if self.is_calibration_mode == False:
        if fraction_num > 35000:
            if self.lightness_white_l < 250:
                self.lightness_white_l += 5
        elif fraction_num < 5000:
            if self.lightness_white_l > 50:
                self.lightness_white_l -= 5

    how_much_short = 0

    for i in range(0, 600):
        if np.count_nonzero(mask[i,::]) > 0:
            how_much_short += 1

    how_much_short = 600 - how_much_short

    if how_much_short > 100:
```

```
            if self.reliability_white_line >= 5:
                self.reliability_white_line -= 5
        elif how_much_short <= 100:
            if self.reliability_white_line <= 99:
                self.reliability_white_line += 5

        msg_white_line_reliability = UInt8()
        msg_white_line_reliability.data = self.reliability_white_line
        self.pub_white_line_reliability.publish(msg_white_line_reliability)

        if self.is_calibration_mode == True:
            if self.pub_image_type == "compressed":
                # publishes white lane filtered image in compressed type

self.pub_image_white_lane.publish(self.cvBridge.cv2_to_compressed_imgmsg(mask, "jpg"))

            elif self.pub_image_type == "raw":
                # publishes white lane filtered image in raw type
                self.pub_image_white_lane.publish(self.cvBridge.cv2_to_imgmsg(mask, "bgr8"))

        return fraction_num, mask

    def maskYellowLane(self, image):
        # Convert BGR to HSV
        hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        Hue_l = self.hue_yellow_l
        Hue_h = self.hue_yellow_h
        Saturation_l = self.saturation_yellow_l
        Saturation_h = self.saturation_yellow_h
        Lightness_l = self.lightness_yellow_l
        Lightness_h = self.lightness_yellow_h

        # define range of yellow color in HSV
        lower_yellow = np.array([Hue_l, Saturation_l, Lightness_l])
        upper_yellow = np.array([Hue_h, Saturation_h, Lightness_h])

        # Threshold the HSV image to get only yellow colors
        mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

        # Bitwise-AND mask and original image
        res = cv2.bitwise_and(image, image, mask = mask)

        fraction_num = np.count_nonzero(mask)

        if self.is_calibration_mode == False:
```

```
            if fraction_num > 35000:
                if self.lightness_yellow_l < 250:
                    self.lightness_yellow_l += 20
            elif fraction_num < 5000:
                if self.lightness_yellow_l > 90:
                    self.lightness_yellow_l -= 20

        how_much_short = 0

        for i in range(0, 600):
            if np.count_nonzero(mask[i,::]) > 0:
                how_much_short += 1

        how_much_short = 600 - how_much_short

        if how_much_short > 100:
            if self.reliability_yellow_line >= 5:
                self.reliability_yellow_line -= 5
        elif how_much_short <= 100:
            if self.reliability_yellow_line <= 99:
                self.reliability_yellow_line += 5

        msg_yellow_line_reliability = UInt8()
        msg_yellow_line_reliability.data = self.reliability_yellow_line
        self.pub_yellow_line_reliability.publish(msg_yellow_line_reliability)

        if self.is_calibration_mode == True:
            if self.pub_image_type == "compressed":
                # publishes yellow lane filtered image in compressed type

self.pub_image_yellow_lane.publish(self.cvBridge.cv2_to_compressed_imgmsg(mask, "jpg"))

            elif self.pub_image_type == "raw":
                # publishes yellow lane filtered image in raw type
                self.pub_image_yellow_lane.publish(self.cvBridge.cv2_to_imgmsg(mask, "bgr8"))

        return fraction_num, mask

    def fit_from_lines(self, lane_fit, image):
        nonzero = image.nonzero()
        nonzeroy = np.array(nonzero[0])
        nonzerox = np.array(nonzero[1])
        margin = 100
        lane_inds = ((nonzerox > (lane_fit[0] * (nonzeroy ** 2) + lane_fit[1] * nonzeroy +
lane_fit[2] - margin)) & (
        nonzerox < (lane_fit[0] * (nonzeroy ** 2) + lane_fit[1] * nonzeroy + lane_fit[2] +
```

```
margin)))

        # Again, extract line pixel positions
        x = nonzerox[lane_inds]
        y = nonzeroy[lane_inds]

        # Fit a second order polynomial to each
        lane_fit = np.polyfit(y, x, 2)

        # Generate x and y values for plotting
        ploty = np.linspace(0, image.shape[0] - 1, image.shape[0])
        lane_fitx = lane_fit[0] * ploty ** 2 + lane_fit[1] * ploty + lane_fit[2]

        return lane_fitx, lane_fit

    def sliding_windown(self, img_w, left_or_right):
        histogram = np.sum(img_w[int(img_w.shape[0] / 2):, :], axis=0)

        # Create an output image to draw on and visualize the result
        out_img = np.dstack((img_w, img_w, img_w)) * 255

        # Find the peak of the left and right halves of the histogram
        # These will be the starting point for the left and right lines
        midpoint = np.int(histogram.shape[0] / 2)

        if left_or_right == 'left':
            lane_base = np.argmax(histogram[:midpoint])
        elif left_or_right == 'right':
            lane_base = np.argmax(histogram[midpoint:]) + midpoint

        # Choose the number of sliding windows
        nwindows = 20

        # Set height of windows
        window_height = np.int(img_w.shape[0] / nwindows)

        # Identify the x and y positions of all nonzero pixels in the image
        nonzero = img_w.nonzero()
        nonzeroy = np.array(nonzero[0])
        nonzerox = np.array(nonzero[1])

        # Current positions to be updated for each window
        x_current = lane_base

        # Set the width of the windows +/- margin
        margin = 50
```

```
    # Set minimum number of pixels found to recenter window
    minpix = 50

    # Create empty lists to receive lane pixel indices
    lane_inds = []

    # Step through the windows one by one
    for window in range(nwindows):
        # Identify window boundaries in x and y
        win_y_low = img_w.shape[0] - (window + 1) * window_height
        win_y_high = img_w.shape[0] - window * window_height
        win_x_low = x_current - margin
        win_x_high = x_current + margin

        # Draw the windows on the visualization image
        cv2.rectangle(out_img, (win_x_low, win_y_low), (win_x_high, win_y_high), (0, 255,
0), 2)

        # Identify the nonzero pixels in x and y within the window
        good_lane_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) & (nonzerox
>= win_x_low) & (
            nonzerox < win_x_high)).nonzero()[0]

        # Append these indices to the lists
        lane_inds.append(good_lane_inds)

        # If you found > minpix pixels, recenter next window on their mean position
        if len(good_lane_inds) > minpix:
            x_current = np.int(np.mean(nonzerox[good_lane_inds]))

    # Concatenate the arrays of indices
    lane_inds = np.concatenate(lane_inds)

    # Extract line pixel positions
    x = nonzerox[lane_inds]
    y = nonzeroy[lane_inds]

    # Fit a second order polynomial to each
    try:
        lane_fit = np.polyfit(y, x, 2)
        self.lane_fit_bef = lane_fit
    except:
        lane_fit = self.lane_fit_bef

    # Generate x and y values for plotting
```

```
        ploty = np.linspace(0, img_w.shape[0] - 1, img_w.shape[0])
        lane_fitx = lane_fit[0] * ploty ** 2 + lane_fit[1] * ploty + lane_fit[2]

        return lane_fitx, lane_fit

    def make_lane(self, cv_image, white_fraction, yellow_fraction):
        # Create an image to draw the lines on
        warp_zero = np.zeros((cv_image.shape[0], cv_image.shape[1], 1), dtype=np.uint8)

        color_warp = np.dstack((warp_zero, warp_zero, warp_zero))
        color_warp_lines = np.dstack((warp_zero, warp_zero, warp_zero))

        ploty = np.linspace(0, cv_image.shape[0] - 1, cv_image.shape[0])

        if yellow_fraction > 3000:
            pts_left = np.array([np.flipud(np.transpose(np.vstack([self.left_fitx, ploty])))])
            cv2.polylines(color_warp_lines, np.int_([pts_left]), isClosed=False, color=(0, 0, 255),
thickness=25)

        if white_fraction > 3000:
            pts_right = np.array([np.transpose(np.vstack([self.right_fitx, ploty]))])
            cv2.polylines(color_warp_lines, np.int_([pts_right]), isClosed=False, color=(255, 255,
0), thickness=25)

        self.is_center_x_exist = True

        if self.reliability_white_line > 50 and self.reliability_yellow_line > 50:
            if white_fraction > 3000 and yellow_fraction > 3000:
                centerx = np.mean([self.left_fitx, self.right_fitx], axis=0)
                pts = np.hstack((pts_left, pts_right))
                pts_center = np.array([np.transpose(np.vstack([centerx, ploty]))])

                cv2.polylines(color_warp_lines, np.int_([pts_center]), isClosed=False, color=(0,
255, 255), thickness=12)

                # Draw the lane onto the warped blank image
                cv2.fillPoly(color_warp, np.int_([pts]), (0, 255, 0))

            if white_fraction > 3000 and yellow_fraction <= 3000:
                centerx = np.subtract(self.right_fitx, 320)
                pts_center = np.array([np.transpose(np.vstack([centerx, ploty]))])

                cv2.polylines(color_warp_lines, np.int_([pts_center]), isClosed=False, color=(0,
255, 255), thickness=12)

            if white_fraction <= 3000 and yellow_fraction > 3000:
```

```
            centerx = np.add(self.left_fitx, 320)
            pts_center = np.array([np.transpose(np.vstack([centerx, ploty]))])

            cv2.polylines(color_warp_lines, np.int_([pts_center]), isClosed=False, color=(0,
255, 255), thickness=12)

        elif self.reliability_white_line <= 50 and self.reliability_yellow_line > 50:
            centerx = np.add(self.left_fitx, 320)
            pts_center = np.array([np.transpose(np.vstack([centerx, ploty]))])

            cv2.polylines(color_warp_lines, np.int_([pts_center]), isClosed=False, color=(0, 255,
255), thickness=12)

        elif self.reliability_white_line > 50 and self.reliability_yellow_line <= 50:
            centerx = np.subtract(self.right_fitx, 320)
            pts_center = np.array([np.transpose(np.vstack([centerx, ploty]))])

            cv2.polylines(color_warp_lines, np.int_([pts_center]), isClosed=False, color=(0, 255,
255), thickness=12)

        else:
            self.is_center_x_exist = False
            pass

        # Combine the result with the original image
        final = cv2.addWeighted(cv_image, 1, color_warp, 0.2, 0)
        final = cv2.addWeighted(final, 1, color_warp_lines, 1, 0)

        if self.pub_image_type == "compressed":
            if self.is_center_x_exist == True:
                # publishes lane center
                msg_desired_center = Float64()
                msg_desired_center.data = centerx.item(350)
                self.pub_lane.publish(msg_desired_center)

            self.pub_image_lane.publish(self.cvBridge.cv2_to_compressed_imgmsg(final, "jpg"))

        elif self.pub_image_type == "raw":
            if self.is_center_x_exist == True:
                # publishes lane center
                msg_desired_center = Float64()
                msg_desired_center.data = centerx.item(350)
                self.pub_lane.publish(msg_desired_center)

            self.pub_image_lane.publish(self.cvBridge.cv2_to_imgmsg(final, "bgr8"))
```

```
    def main(self):
        rospy.spin()

if __name__ == '__main__':
    rospy.init_node('detect_lane')
    node = DetectLane()
    node.main()
```

**Control Lane**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#############################################################################
####
# Copyright 2018 ROBOTIS CO., LTD.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#############################################################################
####

# Author: Leon Jung, Gilbert, Ashe Kim


import rospy
import numpy as np
from std_msgs.msg import Float64
from geometry_msgs.msg import Twist

my_var = 0.0

class ControlLane():
    def __init__(self):
        self.sub_lane = rospy.Subscriber('/control/lane', Float64, self.cbFollowLane, queue_size =
```

```
1)
        self.sub_max_vel = rospy.Subscriber('/control/max_vel', Float64, self.cbGetMaxVel,
queue_size = 1)
        self.pub_cmd_vel = rospy.Publisher('/control/cmd_vel', Twist, queue_size = 1)
        self.sub_new_offset = rospy.Subscriber('/control/new_offset', Float64, self.new_offset,
queue_size =1)

        self.lastError = 0
        self.MAX_VEL = 0.1
        self.offset = 400
        self.new_offset = 400
        rospy.on_shutdown(self.fnShutDown)

    def cbGetMaxVel(self, max_vel_msg):
        self.MAX_VEL = max_vel_msg.data

    def new_offset(self):
        self.offset = self.new_offset

    def cbFollowLane(self, desired_center):
        center = desired_center.data

        error = center - self.offset



        Kp = 0.0025
        Kd = 0.007

        angular_z = Kp * error + Kd * (error - self.lastError)
        self.lastError = error

        twist = Twist()
        # twist.linear.x = 0.05
        twist.linear.x = min(self.MAX_VEL * ((1 - abs(error) / 500) ** 2.2), 0.05)
        twist.linear.y = 0
        twist.linear.z = 0
        twist.angular.x = 0
        twist.angular.y = 0
        if self.MAX_VEL != 0:
            twist.angular.z = -max(angular_z, -2.0) if angular_z < 0 else -min(angular_z, 2.0)
        else:
            twist.angular.z = 0
        self.pub_cmd_vel.publish(twist)

    def fnShutDown(self):
```

```
        rospy.loginfo("Shutting down. cmd_vel will be 0")

        twist = Twist()
        twist.linear.x = 0
        twist.linear.y = 0
        twist.linear.z = 0
        twist.angular.x = 0
        twist.angular.y = 0
        twist.angular.z = 0
        self.pub_cmd_vel.publish(twist)

    def main(self):
        rospy.spin()

if __name__ == '__main__':
    rospy.init_node('control_lane')
    node = ControlLane()
    node.main()
```

**Detect Traffic Light**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#################################################################################
####
# Copyright 2018 ROBOTIS CO., LTD.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#################################################################################
####

# Author: Leon Jung, Gilbert, Ashe Kim

import rospy
```

```
import numpy as np
import cv2
from enum import Enum
from std_msgs.msg import UInt8, Float64, Bool
from sensor_msgs.msg import Image, CompressedImage
from cv_bridge import CvBridge, CvBridgeError
from dynamic_reconfigure.server import Server
from turtlebot3_autorace_detect.cfg import DetectTrafficLightParamsConfig

class DetectTrafficLight():
    def __init__(self):
        self.pub_max_vel = rospy.Publisher('/control/max_vel', Float64, queue_size = 1)
        self.pub_lane_swap = rospy.Publisher('/control/swap_lane', Bool, queue_size =1)
        self.pub_new_offset = rospy.Publisher('/control/new_offset', Float64,  queue_size =1)
        self.hue_red_l = rospy.get_param("~detect/lane/red/hue_l", 0)
        self.hue_red_h = rospy.get_param("~detect/lane/red/hue_h", 26)
        self.saturation_red_l = rospy.get_param("~detect/lane/red/saturation_l", 239)
        self.saturation_red_h = rospy.get_param("~detect/lane/red/saturation_h", 255)
        self.lightness_red_l = rospy.get_param("~detect/lane/red/lightness_l", 123)
        self.lightness_red_h = rospy.get_param("~detect/lane/red/lightness_h", 250)

        self.hue_yellow_l = rospy.get_param("~detect/lane/yellow/hue_l", 19)
        self.hue_yellow_h = rospy.get_param("~detect/lane/yellow/hue_h", 33)
        self.saturation_yellow_l = rospy.get_param("~detect/lane/yellow/saturation_l", 237)
        self.saturation_yellow_h = rospy.get_param("~detect/lane/yellow/saturation_h", 255)
        self.lightness_yellow_l = rospy.get_param("~detect/lane/yellow/lightness_l", 231)
        self.lightness_yellow_h = rospy.get_param("~detect/lane/yellow/lightness_h", 255)

        self.hue_green_l = rospy.get_param("~detect/lane/green/hue_l", 40)
        self.hue_green_h = rospy.get_param("~detect/lane/green/hue_h", 113)
        self.saturation_green_l = rospy.get_param("~detect/lane/green/saturation_l", 210)
        self.saturation_green_h = rospy.get_param("~detect/lane/green/saturation_h", 255)
        self.lightness_green_l = rospy.get_param("~detect/lane/green/lightness_l", 131)
        self.lightness_green_h = rospy.get_param("~detect/lane/green/lightness_h", 255)

        self.is_calibration_mode = rospy.get_param("~is_detection_calibration_mode", False)
        if self.is_calibration_mode == True:
            srv_detect_lane = Server(DetectTrafficLightParamsConfig,
self.cbGetDetectTrafficLightParam)

        self.sub_image_type = "compressed"          # "compressed" / "raw"
        self.pub_image_type = "compressed"          # "compressed" / "raw"

        self.counter = 1

        if self.sub_image_type == "compressed":
```

```
            # subscribes compressed image
            self.sub_image_original = rospy.Subscriber('/detect/image_input/compressed',
CompressedImage, self.cbGetImage, queue_size = 1)
        elif self.sub_image_type == "raw":
            # subscribes raw image
            self.sub_image_original = rospy.Subscriber('/detect/image_input', Image,
self.cbGetImage, queue_size = 1)

        if self.pub_image_type == "compressed":
            # publishes compensated image in compressed type
            self.pub_image_traffic_light = rospy.Publisher('/detect/image_output/compressed',
CompressedImage, queue_size = 1)
        elif self.pub_image_type == "raw":
            # publishes compensated image in raw type
            self.pub_image_traffic_light = rospy.Publisher('/detect/image_output', Image,
queue_size = 1)

        if self.is_calibration_mode == True:
            if self.pub_image_type == "compressed":
                # publishes light image in compressed type
                self.pub_image_red_light =
rospy.Publisher('/detect/image_output_sub1/compressed', CompressedImage, queue_size = 1)
                self.pub_image_yellow_light =
rospy.Publisher('/detect/image_output_sub2/compressed', CompressedImage, queue_size = 1)
                self.pub_image_green_light =
rospy.Publisher('/detect/image_output_sub3/compressed', CompressedImage, queue_size = 1)
            elif self.pub_image_type == "raw":
                # publishes light image in raw type
                self.pub_image_red_light = rospy.Publisher('/detect/image_output_sub1', Image,
queue_size = 1)
                self.pub_image_yellow_light = rospy.Publisher('/detect/image_output_sub2', Image,
queue_size = 1)
                self.pub_image_green_light = rospy.Publisher('/detect/image_output_sub3', Image,
queue_size = 1)


        self.sub_traffic_light_finished = rospy.Subscriber('/control/traffic_light_finished', UInt8,
self.cbTrafficLightFinished, queue_size = 1)

        self.pub_traffic_light_return = rospy.Publisher('/detect/traffic_light_stamped', UInt8,
queue_size=1)
        self.pub_parking_start = rospy.Publisher('/control/traffic_light_start', UInt8, queue_size =
1)
        self.pub_traffic_light = rospy.Publisher('/detect/traffic_light', UInt8, queue_size=1)

        self.CurrentMode = Enum('CurrentMode', 'idle lane_following traffic_light')
```

```
        self.cvBridge = CvBridge()
        self.cv_image = None

        self.is_image_available = False
        self.is_traffic_light_finished = False

        self.green_count = 0
        self.yellow_count = 0
        self.red_count = 0
        self.stop_count = 0
        self.off_traffic = False
        self.cone_seen = False
        self.last_cone_seen = 0
        self.yellow_percent = 0
        rospy.sleep(1)

        loop_rate = rospy.Rate(10)
        while not rospy.is_shutdown():
            if self.is_image_available == True:
                if self.is_traffic_light_finished == False:
                    self.fnFindTrafficLight()

            loop_rate.sleep()

    def cbGetDetectTrafficLightParam(self, config, level):
        rospy.loginfo("[Detect Traffic Light] Detect Traffic Light Calibration Parameter
reconfigured to")
        rospy.loginfo("hue_red_l : %d", config.hue_red_l)
        rospy.loginfo("hue_red_h : %d", config.hue_red_h)
        rospy.loginfo("saturation_red_l : %d", config.saturation_red_l)
        rospy.loginfo("saturation_red_h : %d", config.saturation_red_h)
        rospy.loginfo("lightness_red_l : %d", config.lightness_red_l)
        rospy.loginfo("lightness_red_h : %d", config.lightness_red_h)

        rospy.loginfo("hue_yellow_l : %d", config.hue_yellow_l)
        rospy.loginfo("hue_yellow_h : %d", config.hue_yellow_h)
        rospy.loginfo("saturation_yellow_l : %d", config.saturation_yellow_l)
        rospy.loginfo("saturation_yellow_h : %d", config.saturation_yellow_h)
        rospy.loginfo("lightness_yellow_l : %d", config.lightness_yellow_l)
        rospy.loginfo("lightness_yellow_h : %d", config.lightness_yellow_h)

        rospy.loginfo("hue_green_l : %d", config.hue_green_l)
        rospy.loginfo("hue_green_h : %d", config.hue_green_h)
        rospy.loginfo("saturation_green_l : %d", config.saturation_green_l)
        rospy.loginfo("saturation_green_h : %d", config.saturation_green_h)
```

```
        rospy.loginfo("lightness_green_l : %d", config.lightness_green_l)
        rospy.loginfo("lightness_green_h : %d", config.lightness_green_h)

        self.hue_red_l = config.hue_red_l
        self.hue_red_h = config.hue_red_h
        self.saturation_red_l = config.saturation_red_l
        self.saturation_red_h = config.saturation_red_h
        self.lightness_red_l = config.lightness_red_l
        self.lightness_red_h = config.lightness_red_h

        self.hue_yellow_l = config.hue_yellow_l
        self.hue_yellow_h = config.hue_yellow_h
        self.saturation_yellow_l = config.saturation_yellow_l
        self.saturation_yellow_h = config.saturation_yellow_h
        self.lightness_yellow_l = config.lightness_yellow_l
        self.lightness_yellow_h = config.lightness_yellow_h

        self.hue_green_l = config.hue_green_l
        self.hue_green_h = config.hue_green_h
        self.saturation_green_l = config.saturation_green_l
        self.saturation_green_h = config.saturation_green_h
        self.lightness_green_l = config.lightness_green_l
        self.lightness_green_h = config.lightness_green_h

        return config


    def cbGetImage(self, image_msg):
        # drop the frame to 1/5 (6fps) because of the processing speed. This is up to your
computer's operating power.
        if self.counter % 3 != 0:
            self.counter += 1
            return
        else:
            self.counter = 1

        if self.sub_image_type == "compressed":
            np_arr = np.frombuffer(image_msg.data, np.uint8)
            self.cv_image = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
        else:
            self.cv_image = self.cvBridge.imgmsg_to_cv2(image_msg, "bgr8")

        self.is_image_available = True

    def fnFindTrafficLight(self):
        cv_image_mask = self.fnMaskGreenTrafficLight()
```

```
cv_image_mask = cv2.GaussianBlur(cv_image_mask,(5,5),0)
status1 = self.fnFindCircleOfTrafficLight(cv_image_mask, 'green')



if status1 == 1 or status1 == 5:
   rospy.loginfo("detect GREEN")
   self.stop_count = 0
   self.green_count += 1
   #self.pub_max_vel.publish(0.05)

else:
   self.green_count = 0

   cv_image_mask = self.fnMaskYellowTrafficLight()
   cv_image_mask = cv2.GaussianBlur(cv_image_mask,(5,5),0)

   status2 = self.fnFindCircleOfTrafficLight(cv_image_mask, 'yellow')
   if status2 == 2:
      rospy.loginfo("detect YELLOW")
      self.yellow_count += 1
      if self.cone_seen == False:
         self.cone_seen = True
         if self.yellow_percent > 10:
            self.pub_lane_swap.publish(True)
            self.pub_new_offset.publish(400)
            current_offset = 400
         elif self.yellow_percent > 8:
            self.pub_new_offset.publish(50)
         elif self.yellow_percent > 7:
            self.pub_new_offset.publish(100)
         elif self.yellow_percent > 6:
            self.pub_new_offset.publish(150)
         elif self.yellow_percent > 5:
            self.pub_new_offset.publish(200)
         elif self.yellow_percent > 4:
            self.pub_new_offset.publish(250)
         elif self.yellow_percent > 3:
            self.pub_new_offset.publish(300)
         elif self.yellow_percent > 2:
            self.pub_new_offset.publish(350)
         else:
            self.pub_new_offset.publish(400)


      self.last_cone_seen = 0
```

```
            else:
                self.yellow_count = 0
                if self.cone_seen == True:
                    self.last_cone_seen += 1
                #cv_image_mask = self.fnMaskRedTrafficLight()
                #cv_image_mask = cv2.GaussianBlur(cv_image_mask,(5,5),0)

                #status3 = self.fnFindCircleOfTrafficLight(cv_image_mask, 'red')
                # if status3 == 3:
                  #  rospy.loginfo("detect RED")
                  #   self.red_count += 1
                  #   self.pub_max_vel.publish(0.0)

                #elif status3 == 4:
                #     self.red_count = 0
                  #    self.stop_count += 1
                #     self.pub_red_light = 0.0
                # else:
                #     self.red_count = 0
                  #    self.stop_count = 0



        if self.last_cone_seen > 100:
            if self.cone_seen == true:
                current_offset = 0
            self.cone_seen = False
            self.pub_lane_swap.publish(False)
            if current_offset < 400:
                current_offset = current_offset + 1
            self.pub_new_offset.publish(current_offset)
        if self.green_count >= 3:
            rospy.loginfo("GREEN")
            cv2.putText(self.cv_image,"GREEN", (self.point_col, self.point_low),
cv2.FONT_HERSHEY_DUPLEX, 0.5, (80, 255, 0))
            msg_sign = UInt8()
            msg_sign.data = self.CurrentMode.lane_following.value
            self.pub_traffic_light.publish(msg_sign)
            self.is_traffic_light_finished = False #EDIT


        if self.yellow_count >= 3:
            rospy.loginfo("YELLOW")
            cv2.putText(self.cv_image,"YELLOW", (self.point_col, self.point_low),
cv2.FONT_HERSHEY_DUPLEX, 0.5, (0, 255, 255))
```

```
        if self.red_count >= 3:
            rospy.loginfo("RED")
            cv2.putText(self.cv_image,"RED", (self.point_col, self.point_low),
cv2.FONT_HERSHEY_DUPLEX, 0.5, (0, 0, 255))


        if self.stop_count >= 8:
            rospy.loginfo("STOP") #FALSE TRail
            self.off_traffic = False
            cv2.putText(self.cv_image,"STOP", (self.point_col, self.point_low),
cv2.FONT_HERSHEY_DUPLEX, 0.5, (0, 0, 255))

        if self.pub_image_type == "compressed":
            # publishes traffic light image in compressed type

self.pub_image_traffic_light.publish(self.cvBridge.cv2_to_compressed_imgmsg(self.cv_image
, "jpg"))

        elif self.pub_image_type == "raw":
            # publishes traffic light image in raw type
            self.pub_image_traffic_light.publish(self.cvBridge.cv2_to_imgmsg(self.cv_image,
"bgr8"))


    def fnMaskRedTrafficLight(self):
        image = np.copy(self.cv_image)

        # Convert BGR to HSV
        hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        Hue_l = self.hue_red_l
        Hue_h = self.hue_red_h
        Saturation_l = self.saturation_red_l
        Saturation_h = self.saturation_red_h
        Lightness_l = self.lightness_red_l
        Lightness_h = self.lightness_red_h

        # define range of red color in HSV
        lower_red = np.array([Hue_l, Saturation_l, Lightness_l])
        upper_red = np.array([Hue_h, Saturation_h, Lightness_h])

        # Threshold the HSV image to get only red colors
        mask = cv2.inRange(hsv, lower_red, upper_red)

        # Bitwise-AND mask and original image
        res = cv2.bitwise_and(image, image, mask = mask)
```

```
        if self.is_calibration_mode == True:
            if self.pub_image_type == "compressed":
                # publishes red light filtered image in compressed type
                self.pub_image_red_light.publish(self.cvBridge.cv2_to_compressed_imgmsg(mask,
"jpg"))

            elif self.pub_image_type == "raw":
                # publishes red light filtered image in raw type
                self.pub_image_red_light.publish(self.cvBridge.cv2_to_imgmsg(mask, "mono8"))

        mask = cv2.bitwise_not(mask)

        return mask

    def fnMaskYellowTrafficLight(self):
        image = np.copy(self.cv_image)

        # Convert BGR to HSV
        hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        Hue_l = self.hue_yellow_l
        Hue_h = self.hue_yellow_h
        Saturation_l = self.saturation_yellow_l
        Saturation_h = self.saturation_yellow_h
        Lightness_l = self.lightness_yellow_l
        Lightness_h = self.lightness_yellow_h

        # define range of yellow color in HSV
        lower_yellow = np.array([Hue_l, Saturation_l, Lightness_l])
        upper_yellow = np.array([Hue_h, Saturation_h, Lightness_h])

        # Threshold the HSV image to get only yellow colors
        mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

        h,w = mask.shape[:2]
        pixel_count = h*w

        # Bitwise-AND mask and original image
        res = cv2.bitwise_and(image, image, mask = mask)

        self.yellow_percent = 0
        self.yellow_percent = cv2.countNonZero(mask)
        self.yellow_percent = (self.yellow_percent/pixel_count)*100
        if self.is_calibration_mode == True:
            if self.pub_image_type == "compressed":
```

```
            # publishes yellow light filtered image in compressed type

self.pub_image_yellow_light.publish(self.cvBridge.cv2_to_compressed_imgmsg(mask,
"jpg"))

          elif self.pub_image_type == "raw":
             # publishes yellow light filtered image in raw type
             self.pub_image_yellow_light.publish(self.cvBridge.cv2_to_imgmsg(mask,
"mono8"))

       mask = cv2.bitwise_not(mask)


       rospy.loginfo(self.yellow_percent)



       return mask

    def fnMaskGreenTrafficLight(self):
       image = np.copy(self.cv_image)

       # Convert BGR to HSV
       hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

       Hue_l = self.hue_green_l
       Hue_h = self.hue_green_h
       Saturation_l = self.saturation_green_l
       Saturation_h = self.saturation_green_h
       Lightness_l = self.lightness_green_l
       Lightness_h = self.lightness_green_h

       # define range of green color in HSV
       lower_green = np.array([Hue_l, Saturation_l, Lightness_l])
       upper_green = np.array([Hue_h, Saturation_h, Lightness_h])

       # Threshold the HSV image to get only green colors
       mask = cv2.inRange(hsv, lower_green, upper_green)

       # Bitwise-AND mask and original image
       res = cv2.bitwise_and(image, image, mask = mask)

       if self.is_calibration_mode == True:
          if self.pub_image_type == "compressed":
             # publishes green light filtered image in compressed type
```

```
self.pub_image_green_light.publish(self.cvBridge.cv2_to_compressed_imgmsg(mask, "jpg"))

            elif self.pub_image_type == "raw":
                # publishes green light filtered image in raw type
                self.pub_image_green_light.publish(self.cvBridge.cv2_to_imgmsg(mask, "mono8"))

        mask = cv2.bitwise_not(mask)

        return mask

    def fnFindCircleOfTrafficLight(self, mask, find_color):
        status = 0

        params=cv2.SimpleBlobDetector_Params()
        # Change thresholds
        params.minThreshold = 0
        params.maxThreshold = 255

        # Filter by Area.
        params.filterByArea = True
        params.minArea = 50#EDIT original value 50
        #params.maxArea = 600

        # Filter by Circularity
        #params.filterByCircularity = False #EDIT
        #params.minCircularity = 0.4

        # Filter by Convexity
        #params.filterByConvexity = True
        #params.minConvexity = 0.6

        det=cv2.SimpleBlobDetector_create(params)
        keypts=det.detect(mask)

frame=cv2.drawKeypoints(self.cv_image,keypts,np.array([]),(0,255,255),cv2.DRAW_MATC
HES_FLAGS_DRAW_RICH_KEYPOINTS)

        col1 = 180
        col2 = 270
        col3 = 305

        low1 = 50
        low2 = 170
        low3 = 170

        # if detected more than 1 light
```

```
        for i in range(len(keypts)):
            self.point_col = int(keypts[i].pt[0])
            self.point_low = int(keypts[i].pt[1])
            if self.point_col > col1 and self.point_col < col2 and self.point_low > low1 and
self.point_low < low2:
                if find_color == 'green':
                    status = 1
                elif find_color == 'yellow':
                    status = 2
                elif find_color == 'red':
                    status = 3
            elif self.point_col > col2 and self.point_col < col3 and self.point_low > low1 and
self.point_low < low3:
                if find_color == 'red':
                    status = 4
                elif find_color == 'green':
                    status = 5
            else:
                status = 6

        return status

    def cbTrafficLightFinished(self, traffic_light_finished_msg):
        self.is_traffic_light_finished = False #EDIT

    def main(self):
        rospy.spin()

if __name__ == '__main__':
    rospy.init_node('detect_traffic_light')
    node = DetectTrafficLight()
    node.main()
```