OAKLAND UNIVERSITY™

School of Engineering and Computer Science

Project Report

System for Traffic Light Detection to Implement in Autonomous Vehicles

By
Karthik Challa, Paul Huch, Isha Kulkarni, Zhiming Qian

Course
ECE6460 – Autonomous Vehicle Systems

Professor
Dr. Micho Radovnikovich

TABLE OF CONTENTS

Abstract

Utilizing code written to control the activity of a simulated vehicle in Gazebo; the developed system is able to identify the location of the traffic lights, identify the color displayed by the light, and make a decision on vehicle activity based on displayed light (red, yellow, or green). This is completed by using source code that controls the speed and vehicle activity, identifying lanes that the vehicle will stay within bounds, and the pinhole camera model workshopped to be focused on traffic lights in the simulated environment.

Introduction

The major component used in this project is the vehicle camera to collect image data. To apply this into a working system to identify traffic lights and respond the camera is used to distinguish between the colors on the traffic light, send this data to the vehicle node to process and output the vehicle response based on what color is shown at the signal. If the light is green, then the vehicle will continue to follow the simulated pathway. If the light is yellow, based on the vehicle acceleration at the point of identification, the vehicle will continue or decelerate to stop. If the light is red, then the vehicle will decelerate to stop.

To achieve this the camera is set to a single position with the three simulated traffic lights always in its line of sight. As image data is sent to the vehicle node, the first step is to identify circles using Hough Circles.

Once these circles are identified then the color needs to be specified. The method used to recognize the different colors is the Hue Saturation Hex cone Model. With this method the color detection node will provide the specific color code to the vehicle control node. The complete process is outlined in the diagram below.

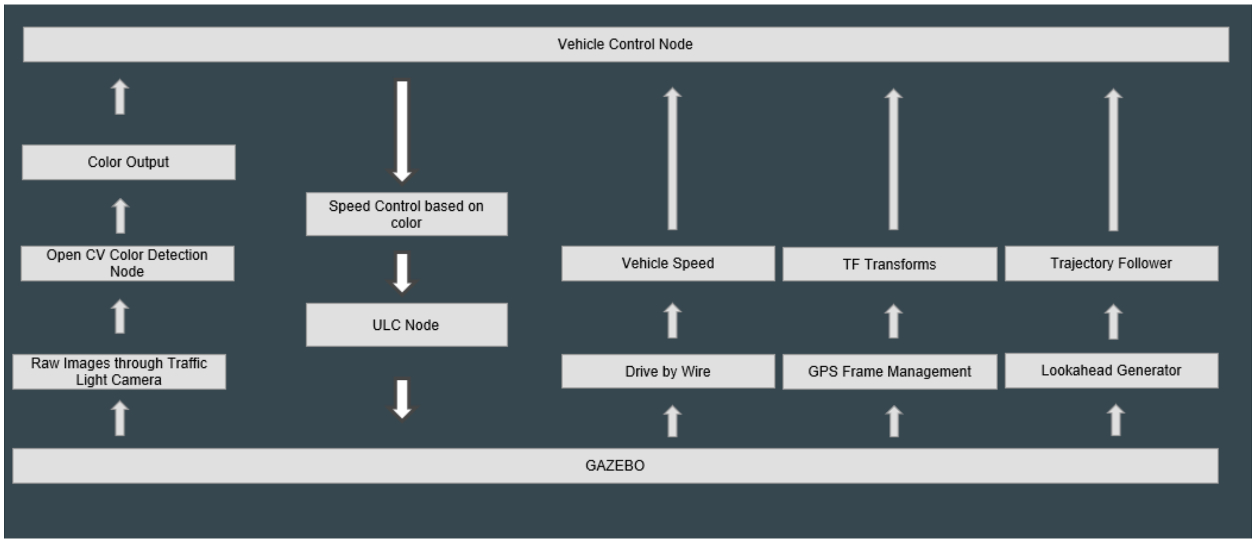


Diagram 1. System Overview A - Color Based Control

The diagram describes the following actions instigated by the vehicle control node in response to the color data input. The speed control is active based on the color, which changes acceleration value to have the vehicle cruise, decelerate to a stop, or ignore the traffic light. This is a straightforward direction in the case of green, where the simulated vehicle will not stop at the light and continue its cruise. In the case of red, the simulated vehicle will decelerate to stop. For the case with yellow there are three options: the vehicle will cruise, the vehicle will decelerate to stop, or the vehicle will ignore the traffic light. This operation for this case is determined by the current acceleration of the vehicle and is outlined in Diagram 2.

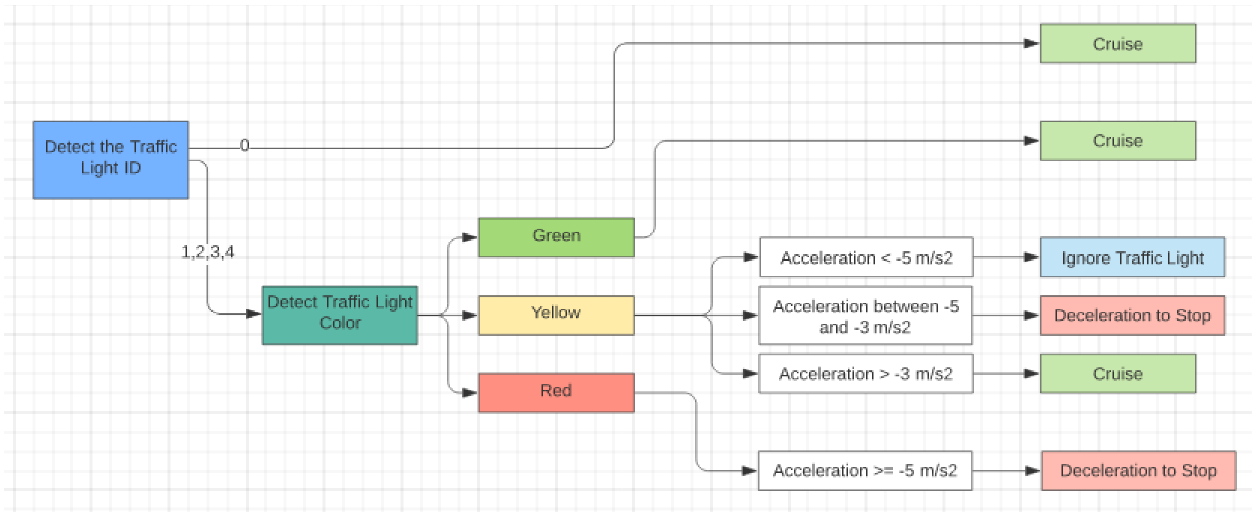


Diagram 2. Vehicle Control

Implementation

There are two major parts that have gone in to complete the project. First is the recognition of the traffic lights by the camera, and second is the control of the simulated vehicle in order for it to respond to the different colored lights.

Traffic Light Recognition

In terms of traffic light recognition, the main tool we use is OpenCV. The color detection part consists of two main codes, Hue Saturation Value Hex cone Model (HSV) function and Hough circle function. HSV is used to detect different colors and Hough circles are used to detect the circles.

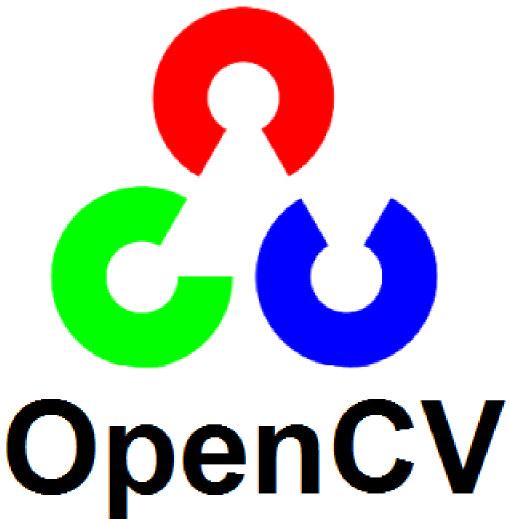


Diagram 3: Open CV Icon

1.Hue Saturation Value Hex cone Model

HSV is an intuitive color model for users. To start with a pure color, specify the color angle H, and let V=S=1, and then get the desired color by adding black and white to it. Increasing black can reduce V without changing S, and increasing white can reduce S without changing V. For example, to get dark blue, V=0.4 S=1 H=210 degrees. To get light blue, V=1 S=0.4 H=210 degrees.

The human eye can distinguish 128 different colors, 130 color saturations, and 23 shades. If 16Bit is used to represent HSV, then 7 bits can be used to store H, 4 bits to store S, and 5 bits to store V, overall 745 or 655 can meet our needs.

Since HSV is an intuitive color model, it is widely used in image editing tools, such as Photoshop (called HSB in Photoshop), but this also means that it is not suitable for use in lighting models. Many light hybrid operations, and light intensity operations, cannot be directly implemented using HSV.

Another intuitive color model is the HSL model, where the first two parameters are the same as HSV, and L represents brightness. Its three-dimensional representation is a double pyramid.

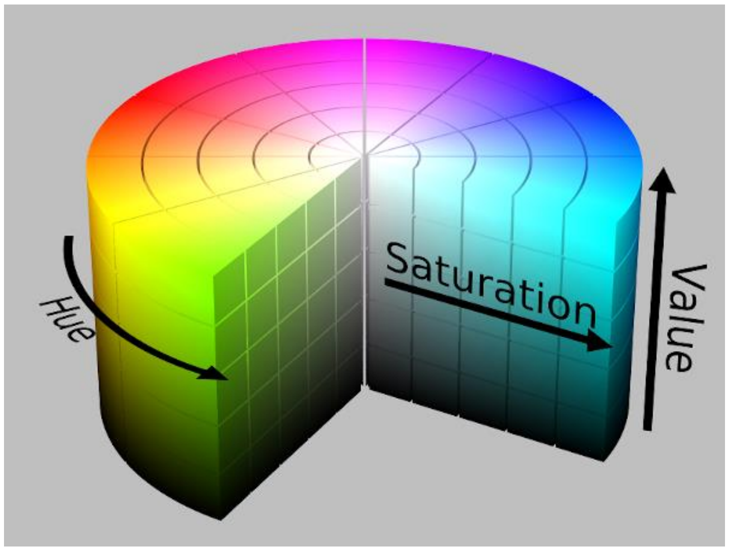
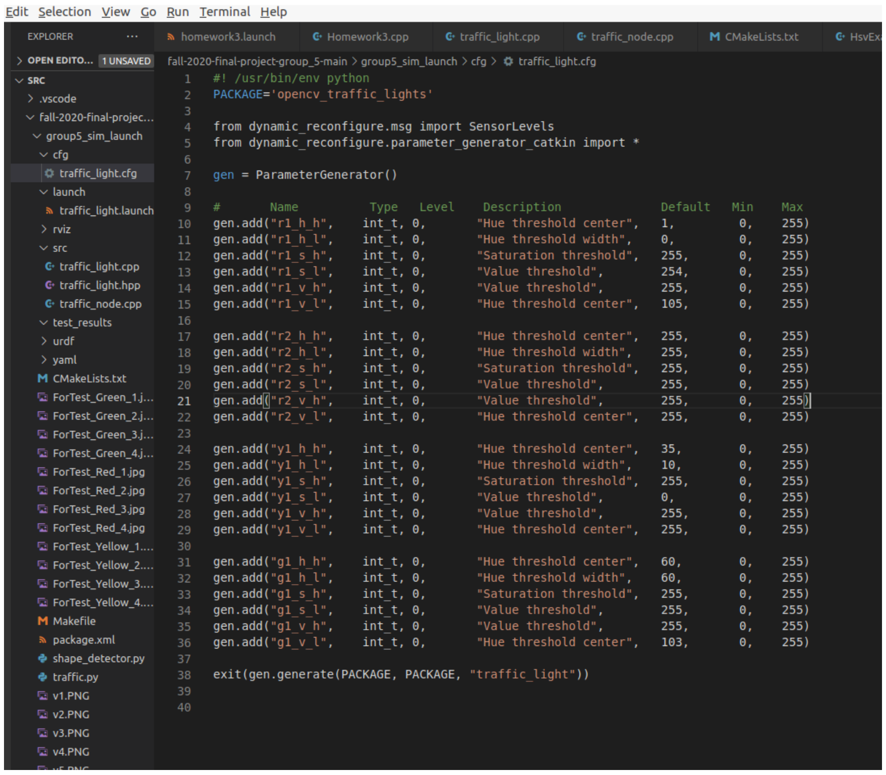


Diagram 4. Vehicle Control

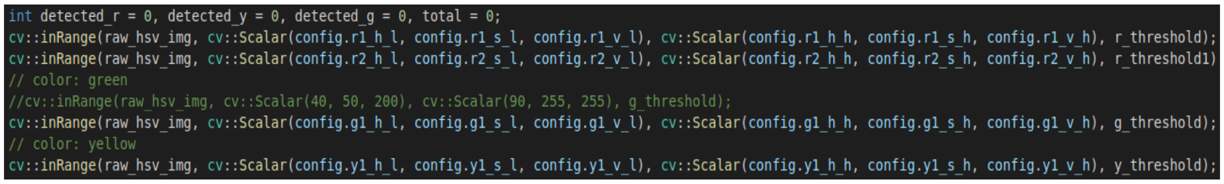
Above is the HSV model. What is needed for this part is to match correct parameters with corresponding colors. However, adjusting many independent variables is not simple. Dependent variables change with the independent variables. To overcome this, 24 parameters are named and saved in the cfg file shown below. They limited the range of red, yellow and green colors.


Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS 1 UNSAVED
homework3.launch
Homework3.cpp
traffic_light.cpp
traffic_node.cpp
CMakeLists.txt
fall-2020-final-project-group_5-main > group5_sim_launch > cfg > traffic_light.cfg

```
#! /usr/bin/env python
PACKAGE='opencv_traffic_lights'

from dynamic_reconfigure.msg import SensorLevels
from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()

#       Name          Type   Level     Description                  Default   Min    Max
gen.add("r1_h_h",    int_t, 0,       "Hue threshold center",   1,         0,    255)
gen.add("r1_h_l",    int_t, 0,       "Hue threshold width",    0,         0,    255)
gen.add("r1_s_h",    int_t, 0,       "Saturation threshold",   255,       0,    255)
gen.add("r1_s_l",    int_t, 0,       "Value threshold",        254,       0,    255)
gen.add("r1_v_h",    int_t, 0,       "Value threshold",        255,       0,    255)
gen.add("r1_v_l",    int_t, 0,       "Hue threshold center",   105,       0,    255)

gen.add("r2_h_h",    int_t, 0,       "Hue threshold center",   255,       0,    255)
gen.add("r2_h_l",    int_t, 0,       "Hue threshold width",    255,       0,    255)
gen.add("r2_s_h",    int_t, 0,       "Saturation threshold",   255,       0,    255)
gen.add("r2_s_l",    int_t, 0,       "Value threshold",        255,       0,    255)
gen.add("r2_v_h",    int_t, 0,       "Value threshold",        255,       0,    255)
gen.add("r2_v_l",    int_t, 0,       "Hue threshold center",   255,       0,    255)

gen.add("y1_h_h",    int_t, 0,       "Hue threshold center",   35,        0,    255)
gen.add("y1_h_l",    int_t, 0,       "Hue threshold width",    10,        0,    255)
gen.add("y1_s_h",    int_t, 0,       "Saturation threshold",   255,       0,    255)
gen.add("y1_s_l",    int_t, 0,       "Value threshold",        0,         0,    255)
gen.add("y1_v_h",    int_t, 0,       "Value threshold",        255,       0,    255)
gen.add("y1_v_l",    int_t, 0,       "Hue threshold center",   255,       0,    255)

gen.add("g1_h_h",    int_t, 0,       "Hue threshold center",   60,        0,    255)
gen.add("g1_h_l",    int_t, 0,       "Hue threshold width",    60,        0,    255)
gen.add("g1_s_h",    int_t, 0,       "Saturation threshold",   255,       0,    255)
gen.add("g1_s_l",    int_t, 0,       "Value threshold",        255,       0,    255)
gen.add("g1_v_h",    int_t, 0,       "Value threshold",        255,       0,    255)
gen.add("g1_v_l",    int_t, 0,       "Hue threshold center",   103,       0,    255)

exit(gen.generate(PACKAGE, PACKAGE, "traffic_light"))
```

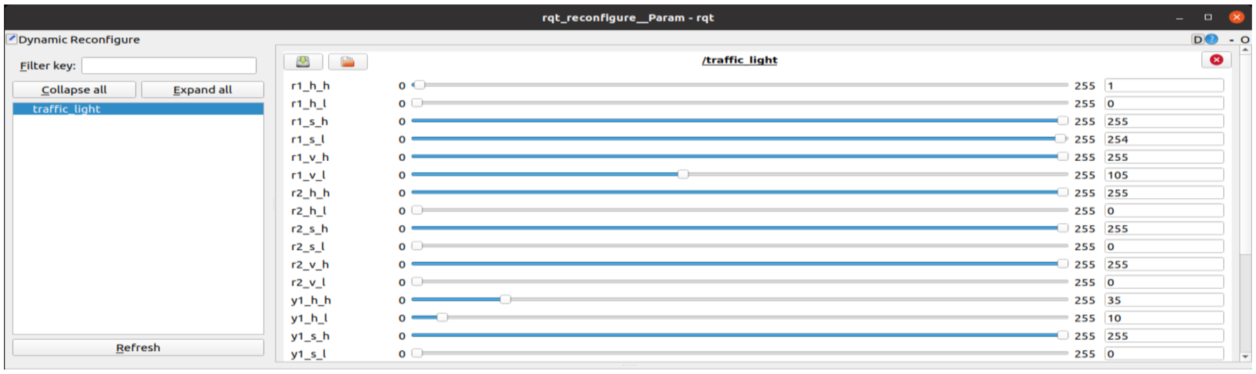
Diagram 5: Config file

Then the independent variables are input into the HSV main code, and made visible by using a sliding bar.



```
int detected_r = 0, detected_y = 0, detected_g = 0, total = 0;
cv::inRange(raw_hsv_img, cv::Scalar(config.r1_h_l, config.r1_s_l, config.r1_v_l), cv::Scalar(config.r1_h_h, config.r1_s_h, config.r1_v_h), r_threshold);
cv::inRange(raw_hsv_img, cv::Scalar(config.r2_h_l, config.r2_s_l, config.r2_v_l), cv::Scalar(config.r2_h_h, config.r2_s_h, config.r2_v_h), r_threshold1)
// color: green
//cv::inRange(raw_hsv_img, cv::Scalar(40, 50, 200), cv::Scalar(90, 255, 255), g_threshold);
cv::inRange(raw_hsv_img, cv::Scalar(config.g1_h_l, config.g1_s_l, config.g1_v_l), cv::Scalar(config.g1_h_h, config.g1_s_h, config.g1_v_h), g_threshold);
// color: yellow
cv::inRange(raw_hsv_img, cv::Scalar(config.y1_h_l, config.y1_s_l, config.y1_v_l), cv::Scalar(config.y1_h_h, config.y1_s_h, config.y1_v_h), y_threshold);
```

Diagram 6. Code for color detection



```
r1_h_h  0  255  1
r1_h_l  0  255  0
r1_s_h  0  255  255
r1_s_l  0  255  254
r1_v_h  0  255  255
r1_v_l  0  255  105
r2_h_h  0  255  255
r2_h_l  0  255  0
r2_s_h  0  255  255
r2_s_l  0  255  0
r2_v_h  0  255  255
r2_v_l  0  255  0
y1_h_h  0  255  35
y1_h_l  0  255  10
y1_s_h  0  255  255
y1_s_l  0  255  0
```

Diagram 7: Sliding bar

Instant changes and the trend of the picture are now visible as the parameter changes by sliding the bar. This way parameters can be adjusted to appropriate values and record these values as default values and fill them in the config file.

2.HoughCircles

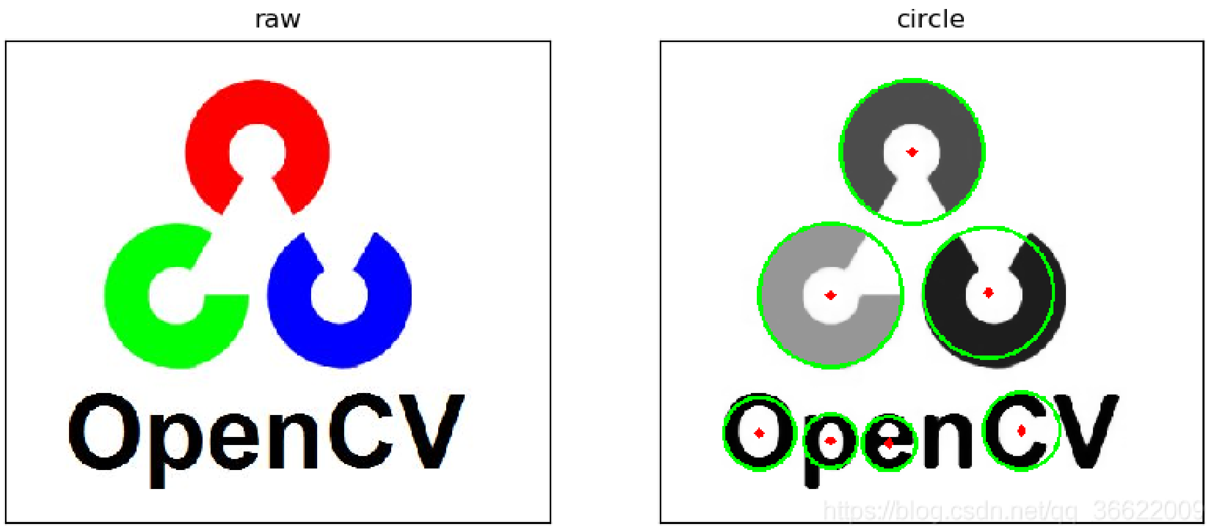


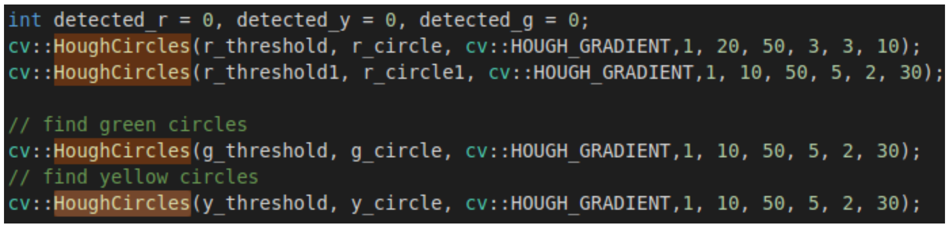
Diagram 8: OpenCV icon processed by Hough Circle

The second main function of the detection part is Hough circle, since the light bulb part of the traffic lights are mostly round. The Hough circle is a function to detect circular/rounded shapes.

The circle is expressed mathematically as $(x-x_c)^2 + (y-y_c)^2 = r^2$, where (x_c, y_c) is the center of the circle and r is the radius of the circle. From the equation, there are 3 parameters, so a 3D accumulator is needed to perform the Hough transform, which will be very ineffective. Therefore, OpenCV uses a more difficult method, the Hough gradient method using edge gradient information.

We use the following function: cv.HoughCircles (image, circle, method, dp, minDist, param1, param2, minRadius, maxRadius)
image : 8-bit, single-channel, grayscale input image.
circles : output vector of found circles(cv.CV_32FC3 type). Each vector is encoded as a 3-element floating-point vector (x,y,radius).
method : detection method(see cv.HoughModes). Currently, the only implemented method is HOUGH_GRADIENT

dp : inverse ratio of the accumulator resolution to the image resolution. For example, if dp = 1, the accumulator has the same resolution as the input image. If dp = 2, the accumulator has half as big width and height.
minDist : minimum distance between the centers of the detected circles. If the parameter is too small, multiple neighbor circles may be falsely detected in addition to a true one. If it is too large, some circles may be missed.
param1 : first method-specific parameter. In case of HOUGH_GRADIENT , it is the higher threshold of the two passed to the Canny edge detector (the lower one is twice smaller).
param2 : second method-specific parameter. In case of HOUGH_GRADIENT , it is the accumulator threshold for the circle centers at the detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first.
minRadius : minimum circle radius.
maxRadius : maximum circle radius.



```
int detected_r = 0, detected_y = 0, detected_g = 0;
cv::HoughCircles(r_threshold, r_circle, cv::HOUGH_GRADIENT,1, 20, 50, 3, 3, 10);
cv::HoughCircles(r_threshold1, r_circle1, cv::HOUGH_GRADIENT,1, 10, 50, 5, 2, 30);

// find green circles
cv::HoughCircles(g_threshold, g_circle, cv::HOUGH_GRADIENT,1, 10, 50, 5, 2, 30);
// find yellow circles
cv::HoughCircles(y_threshold, y_circle, cv::HOUGH_GRADIENT,1, 10, 50, 5, 2, 30);
```

Diagram 9: Hough Circle Function

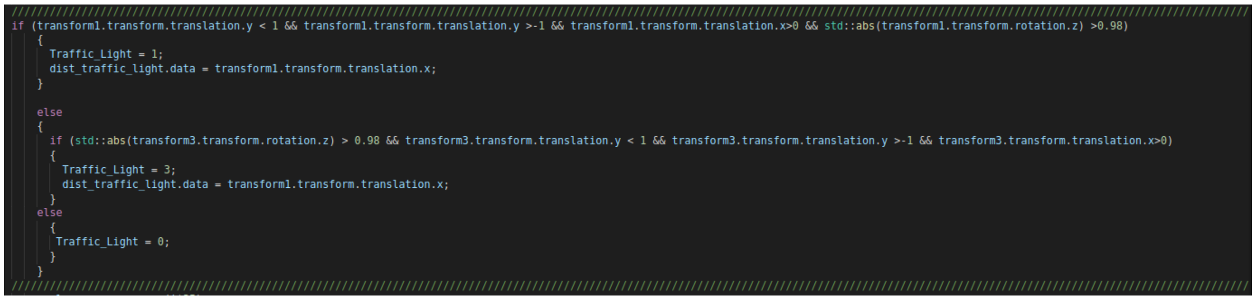
Combining these two codes together, the colored light circles are able to be detected by the camera.

Vehicle Control

This section will present the three steps necessary to control a car based on different traffic light conditions.

1) <u>Identification of the Traffic Light:</u>

Based on the figure-eight design of the route, a total of four traffic lights are located at its center. Due to their unique position, the system needs to ensure what traffic light is currently being approached by the car. The figure-eight design makes it fairly easy to identify, since every traffic light is located at an angle of 90 degrees towards each other. Using the TF transforms and comparing the relative x and y values allow the identification of each traffic light. The following figure shows the code implemented in order to determine the traffic lights' ID.



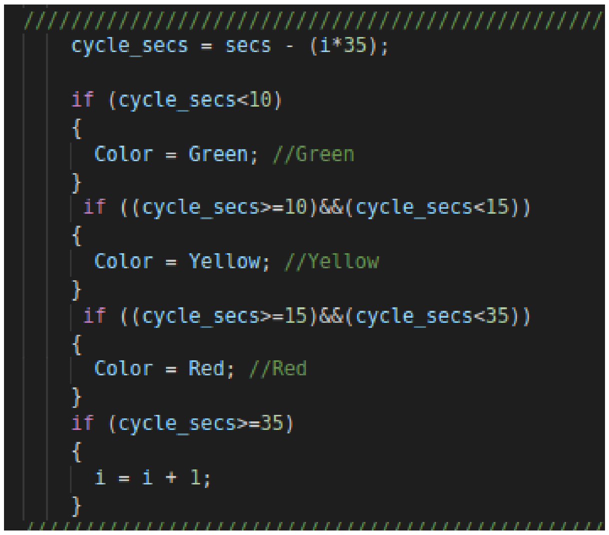
```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
if (transform1.transform.translation.y < 1 && transform1.transform.translation.y >-1 && transform1.transform.translation.x>0 && std::abs(transform1.transform.rotation.z) >0.98)
    {
      Traffic_Light = 1;
      dist_traffic_light.data = transform1.transform.translation.x;
    }

    else
    {
      if (std::abs(transform3.transform.rotation.z) > 0.98 && transform3.transform.translation.y < 1 && transform3.transform.translation.y >-1 && transform3.transform.translation.x>0)
      {
        Traffic_Light = 3;
        dist_traffic_light.data = transform1.transform.translation.x;
      }
    else
      {
       Traffic_Light = 0;
      }
    }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Diagram 10: Traffic Light Identification

In the figure above, it can be seen that only traffic lights one and three are being detected. The car's route is designed in such a way that only the above mentioned lights would be passed. If the car is currently not near any traffic light structure, the variable called "Traffic_Light" is being set to "0". This declaration is important for the state specific control function explained at a later point of this report.

2) <u>State Detection based on Timestamps</u>

Due to problems with the integration of the color detection algorithm, an alternative approach has been implemented. Knowing the green, yellow and red light conditions are cycling through at a constant pace, allowed a state estimation based on the simulation time of the system. In our case the green light is active for 10 seconds, the yellow light for 5 seconds and the red light for 25 seconds which then repeats the cycle starting with the activation of the green light for 10 seconds. The figure shown below demonstrates the algorithm to catch the state of the lights based on the above mentioned behavior.



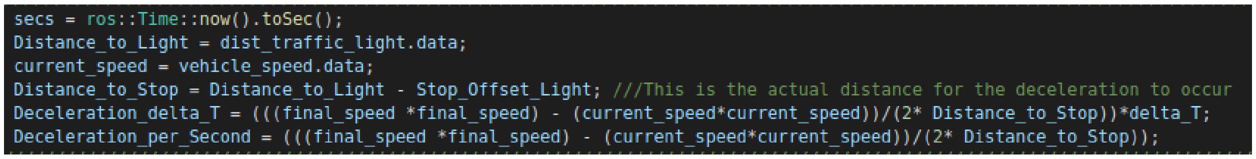
```
//////////////////////////////////////////////////
    cycle_secs = secs - (i*35);

    if (cycle_secs<10)
    {
      Color = Green; //Green
    }
     if ((cycle_secs>=10)&&(cycle_secs<15))
    {
      Color = Yellow; //Yellow
    }
     if ((cycle_secs>=15)&&(cycle_secs<35))
    {
      Color = Red; //Red
    }
    if (cycle_secs>=35)
    {
      i = i + 1;
    }
```

Diagram 11: State detection through cycle time

Every cycle is represented by the variable "i" which allows an easy but effective loop to achieve the desired behavior. To simplify this, our simulation synchronized the state (color) of every traffic light, which would never happen in a real life scenario.

3) <u>State specific controls</u>

Since the state and the ID of each traffic light is now known, the main control algorithms for each state could be developed. As mentioned in the introduction section, the task was to control the car based on the color of the traffic lights. In order for the car to slow down the "linear velocity ulc command" was used to input to the vehicle. In order for the deceleration to work, multiple variables needed to be defined beforehand.



```
secs = ros::Time::now().toSec();
Distance_to_Light = dist_traffic_light.data;
current_speed = vehicle_speed.data;
Distance_to_Stop = Distance_to_Light - Stop_Offset_Light; ///This is the actual distance for the deceleration to occur
Deceleration_delta_T = (((final_speed *final_speed) - (current_speed*current_speed))/(2* Distance_to_Stop))*delta_T;
Deceleration_per_Second = (((final_speed *final_speed) - (current_speed*current_speed))/(2* Distance_to_Stop));
```

Diagram 12: Variables

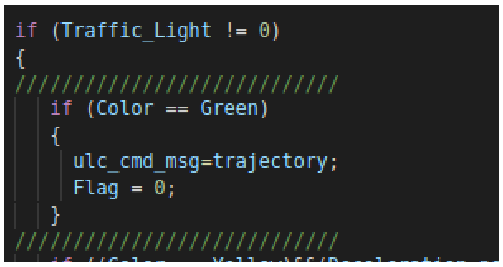
The distance between the car and the desired stopping point ("stop line") is calculated by taking the distance towards the light and subtracting the offset of the line towards the traffic light structure. This value was stored as the name "Distance_to_Stop" and is constantly updated at a rate of 50Hz (call back frequency). Along to this variable the current speed of the car is calculated. Knowing both variables now allowed the computation of the deceleration necessary to slow down the car in a controlled manner. The deceleration value is based on the basic kinematic equation: $A = (v_f^2 - v_i^2) \div (2 \times D_S)$.

Where v_f is the final desired velocity of the car ($0\frac{m}{s}$ in our case), v_i is the instantaneous velocity of the car and D_s is the distance within the car needs to be slowed down to a complete stop. Due to the fact that each value is updated at a rate of 50Hz, a variable called Deceleration_delta_T has been initialized.

As mentioned earlier, the variable called “Traffic_Light” is being set to “0” when the car is traversing through space and is currently not approaching any traffic light. The controls are only applied to the car, in case the value of the variable is not zero.

<u>Green Condition</u>

In the case the traffic light is in a “green” state:



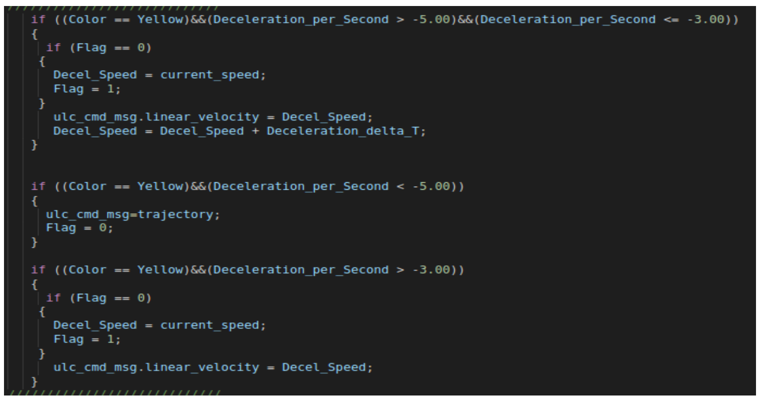
```
if (Traffic_Light != 0)
{
////////////////////////////
    if (Color == Green)
    {
      ulc_cmd_msg=trajectory;
      Flag = 0;
    }
////////////////////////////
```

Diagram 13: Green Condition

At the green light condition, the car is cruising at the pre encoded speed of the route, while no controls are being applied.

<u>Yellow Condition</u>

When the traffic light turns to a yellow light, multiple variables need to be considered in order to apply the desired input to the car. The following figure illustrates three cases that apply different control inputs to the car.



```
    if ((Color == Yellow)&&(Deceleration_per_Second > -5.00)&&(Deceleration_per_Second <= -3.00))
    {
      if (Flag == 0)
     {
       Decel_Speed = current_speed;
       Flag = 1;
     }
       ulc_cmd_msg.linear_velocity = Decel_Speed;
       Decel_Speed = Decel_Speed + Deceleration_delta_T;
    }


    if ((Color == Yellow)&&(Deceleration_per_Second < -5.00))
    {
      ulc_cmd_msg=trajectory;
      Flag = 0;
    }

    if ((Color == Yellow)&&(Deceleration_per_Second > -3.00))
    {
      if (Flag == 0)
     {
       Decel_Speed = current_speed;
       Flag = 1;
     }
       ulc_cmd_msg.linear_velocity = Decel_Speed;
    }
////////////////////////////
```

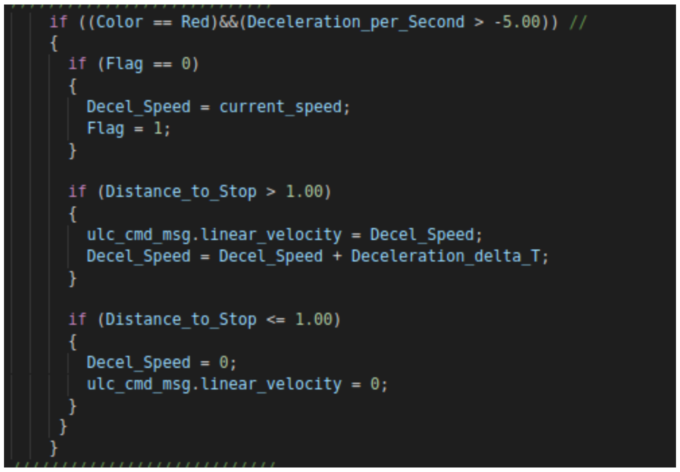
Diagram 13: Yellow Light Detection

If the deceleration value necessary to stop the car is between $-5\frac{m}{s^2}$ and $-3\frac{m}{s^2}$ (valid range) the linear velocity command is input to the vehicle. Every 0.02s the velocity input of the car is subtracted by the “Deceleration_delta_T” variable. This case is very similar to the traffic light being red, which is explained at a later point of this report.
If the deceleration is lower than the maximum valid deceleration ($-5\frac{m}{s^2}$), the traffic light is being ignored since there would not be enough time to slow down the car.
Last but not least, if the deceleration necessary to slow down the car is higher than a user defined value of $-3\frac{m}{s^2}$there is no reason to start slowing down the car yet. However, in this case the car stays at a constant speed.

Red Condition

In the case where the traffic light turns to red, the controls behave similar to the yellow conditions.

If the deceleration is larger than the maximum valid deceleration ($-5\frac{m}{s^2}$), the car is slowed down until it comes to a complete stop or the traffic light turns green. An additional condition has been implemented that inputs a zero velocity to the car, when it is one meter away from the “stop line”. Due to the internal “velocity ramp” behavior of the system, this control puts the vehicle fairly close to the desired position. Using this strategy avoids the deceleration value to blow up to infinity.



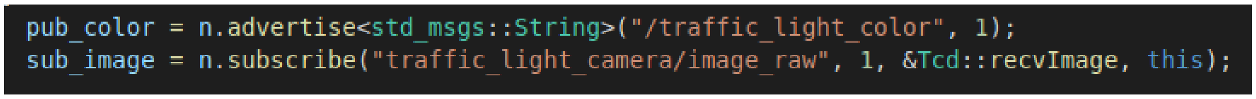
```
if ((Color == Red)&&(Deceleration_per_Second > -5.00)) //
{
  if (Flag == 0)
  {
    Decel_Speed = current_speed;
    Flag = 1;
  }

  if (Distance_to_Stop > 1.00)
  {
    ulc_cmd_msg.linear_velocity = Decel_Speed;
    Decel_Speed = Decel_Speed + Deceleration_delta_T;
  }

  if (Distance_to_Stop <= 1.00)
  {
    Decel_Speed = 0;
    ulc_cmd_msg.linear_velocity = 0;
  }
 }
}
```

Diagram 14: Red Light Condition

System Design

The vehicle control node and the color detection node are two sperate modules each with their own significant functions.

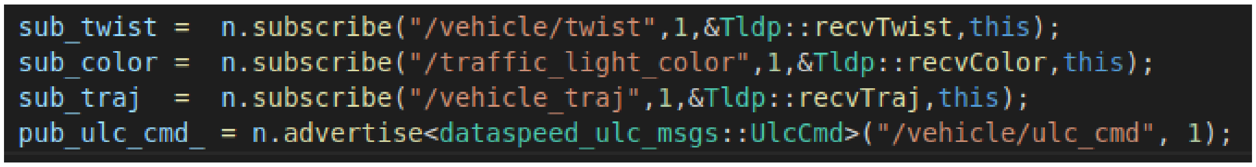
The color detection node is subscribed to “traffic_light_camera/image_raw” topic where it gets the captured image from. Then it performs the image processing and color detection and ultimately advertises the color detected on “traffic_light_color” topic.



```
pub_color = n.advertise<std_msgs::String>("/traffic_light_color", 1);
sub_image = n.subscribe("traffic_light_camera/image_raw", 1, &Tcd::recvImage, this);
```

Diagram 15: Color Detection Node topics

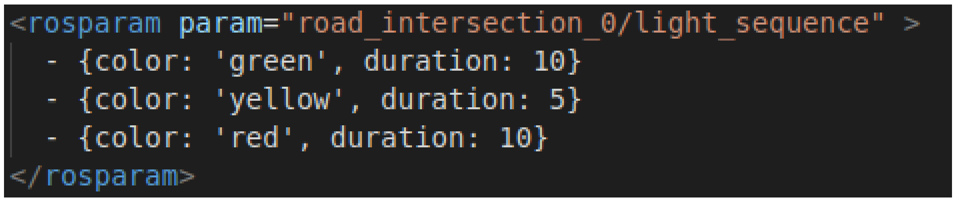
The vehicle control node subscribes to “traffic_light_color” for color information, "/vehicle/twist" topic for speed data and "/vehicle_traj" topic for route map. Then after processing the vehicle control based on color detection, it advertises the vehicle velocity command on the "/vehicle/ulc_cmd" topic.



```
sub_twist =  n.subscribe("/vehicle/twist",1,&Tldp::recvTwist,this);
sub_color =  n.subscribe("/traffic_light_color",1,&Tldp::recvColor,this);
sub_traj  =  n.subscribe("/vehicle_traj",1,&Tldp::recvTraj,this);
pub_ulc_cmd_ = n.advertise<dataspeed_ulc_msgs::UlcCmd>("/vehicle/ulc_cmd", 1);
```

Diagram 16: Vehicle Control node topics

But for the real implementation as mentioned in vehicle control section, the timer module was used. As mentioned in diagram 12, the timing for each of the lights has been encoded in the launch file.



```
<rosparam param="road_intersection_0/light_sequence" >
  - {color: 'green', duration: 10}
  - {color: 'yellow', duration: 5}
  - {color: 'red', duration: 10}
</rosparam>
```

Diagram 17: Traffic Light timing sequence

So, by taking into consideration a timing-based approach and using simulation timing, the vehicle control was achieved. The Diagram 18 below will illustrate the system with timer-based control.

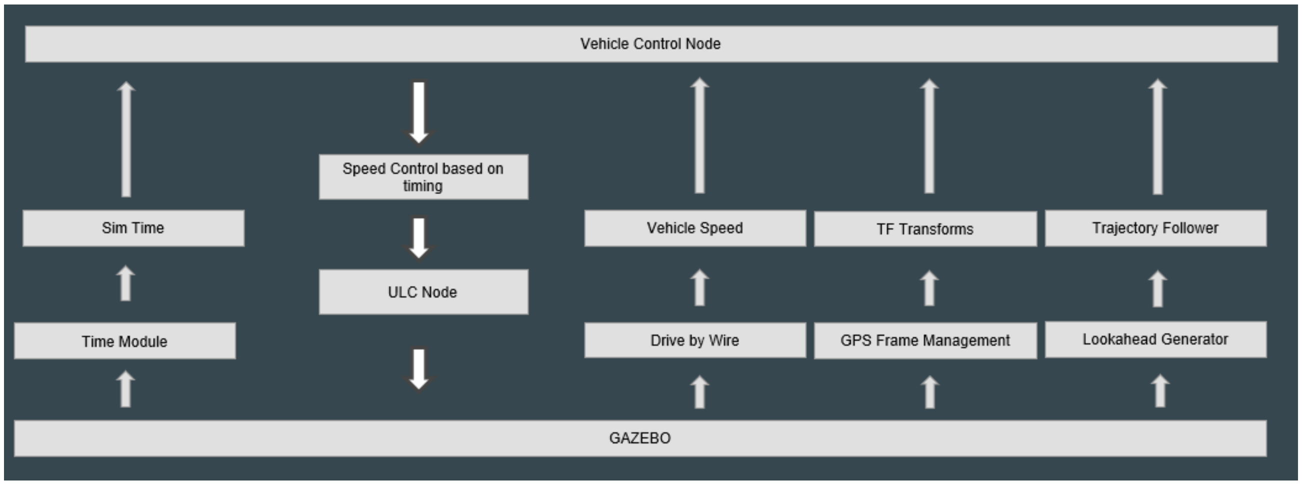


Diagram 18: Timer based system design and implementation

Conclusion

The color detection and timing-based vehicle control have been achieved successfully in this project. The following GITHUB link has the working code, videos and images that shows the working.

https://github.com/ou-git-classrooms/fall-2020-final-project-group_5

Although there is a lot of scope for this project, we will be working towards better implementation of color detection-based vehicle control. Also, this project could be extended to real world traffic scenario which can include blinking red/yellow signs, sign board detection etc.

Responsibilities

Isha Kulkarni, B.M.E.
Introduction and Overview

Karthik Challa, E.C.E.
System Design coding, testing and documentation of “System Design”

Zhiming Qian, E.C.E.
Color Detection coding, testing and documentation of “Color Detection”

Paul Huch, M.E.
Vehicle Control coding, testing and documentation of “Vehicle Control”

References

https://docs.opencv.org/3.4/d3/de5/tutorial_js_houghcircles.html
http://wiki.ros.org/
https://opencv.org/