**Self-Driving Car Engineer Nanodegree Programm**

**Project – Capstone**

**Authors**:

Derrick Choo

Chidhanandh Krishnaraj

Libin Jia

Michael Zill

Siqi Ying

Version 2.0

This document provides brief description of how the Capstone project was completed with different section explained in detail and what steps were followed.

Abstract:

In Automobile Industry, the expectation towards the driver assistance and driver safety arouse the need of autonomously driving vehicles which could have a near-zero accident rate, keeping the safety of the driver and environment and maintaining the standards of the Automobile industry. This project is divided into different section to attain the main goal which is to make the Car drive by itself in the simulator and in the real world with real time environment (Carla) considering obstacles and traffic lights.

Goal:

To make the Ego car drive by itself in the traffic situations and following the right trajectory to reach the goal point in the Simulator and in the real world (Carla).

TODO:

We will be writing ROS nodes to implement core functionality of the autonomous vehicle system, including traffic light detection, control, and waypoint following! Test our code using a simulator, and when everything is working, the project can be submitted to be run on Carla.

The project is distributed into different phases as bellow, which are ROS nodes:

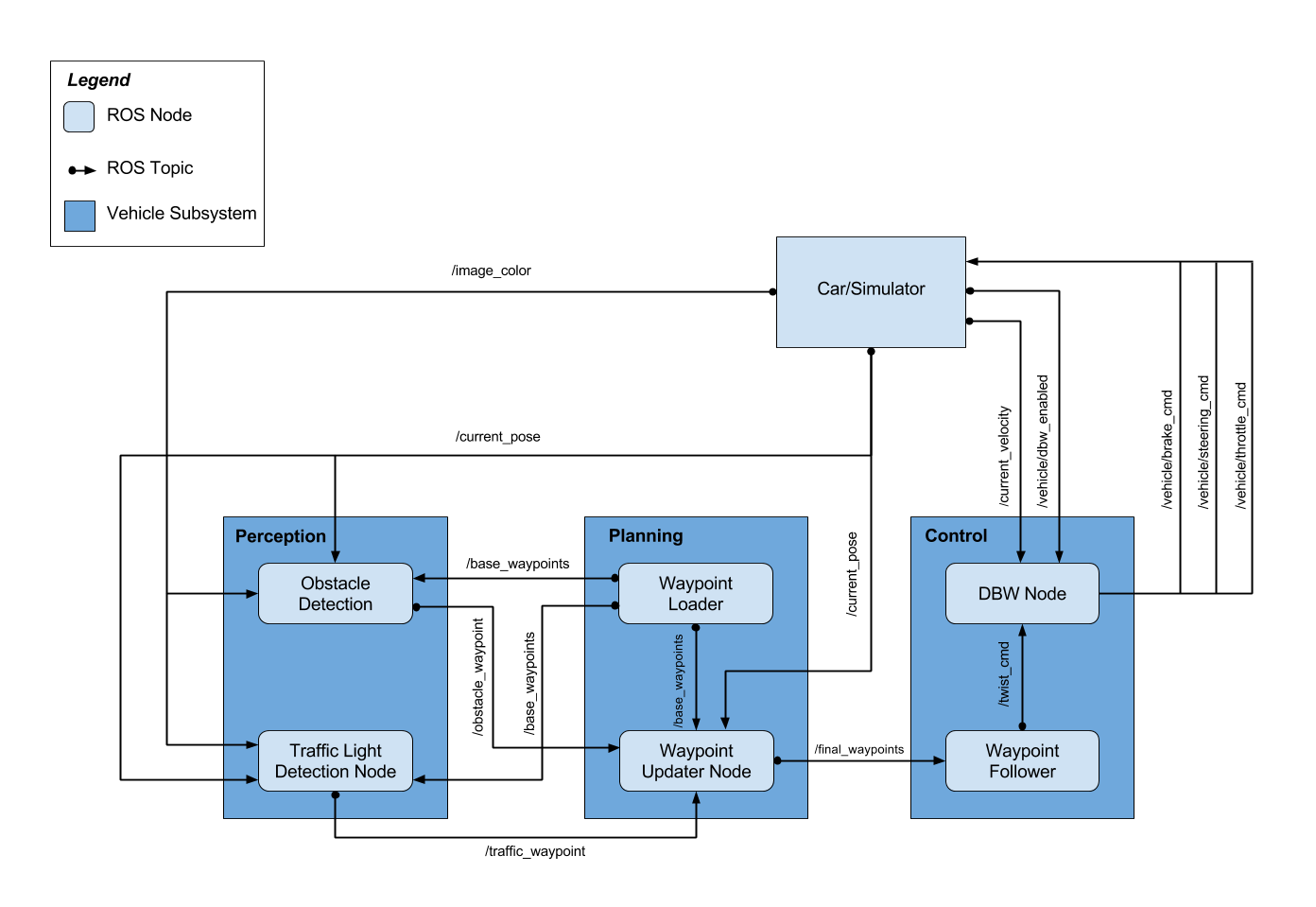
Phase 1: Waypoint updater (Partial)

Phase 2: DBW

Phase 3: Traffic Light Detection

Phase 4: Waypoint updater (full)

The following is a system architecture diagram showing the ROS nodes and topics used in the project.



Phase 1: Waypoint Updater (Partial)

Description:

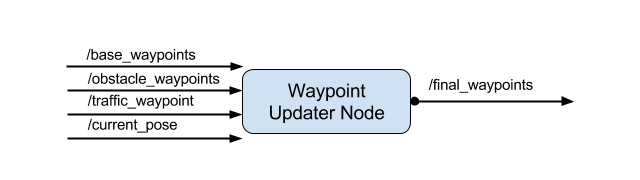
The waypoint updater node will publish the final waypoints which provides the trajectory for the ego car to move around.

Inputs:

/base\_waypoints: Published by Waypoint\_loader, which is the static waypoints which are the list of all the waypoints from the track. Waypoints as provided by a static .csv file.

/obstacle\_waypoints: Published by the Obstacle detection module.

/traffic\_waypoint: Published by Traffic Light Detection Node which published the waypoints to the traffic red light.

/current\_pose: Current position published by the Car or the simulator.

Output:

/final\_waypoints

The final waypoints is published which provides the fixed number of waypoints ahead of the vehicle.

The total number of waypoints ahead of the vehicle that should be included in the /final\_waypoints list is provided by the LOOKAHEAD\_WPS (200 in this case) variable in *waypoint\_updater.py*.

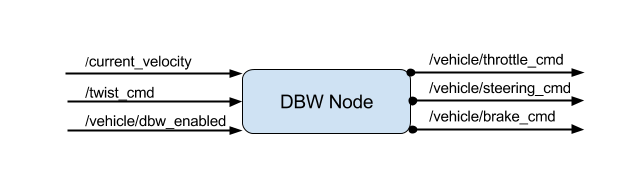
Phase 2: DBW

Description:

Drive by wire (DBW) system will control the vehicle through controlling throttle, braking, and steering. The DBW node logic accepts linear and angular velocity by subscribing to twist\_cmd and publish the throttle, brake, and steering commands. The DBW node can be disabled and the driver can control it.

Inputs and outputs

This diagram illustrates the inputs and outputs for DBW node:



The inputs are:

/current\_velocity: published by simulator and used by the DBW node to determine the linear velocity and provide it to controller.

/twist\_cmd: Waypoint\_follower node publishes it and subscribed by DBW node to publish throttle, steering and brake commands.

/vehicledbw\_enable: pusblished by simulator. DBW will determine whether or not to publish throttle, steering, and brake information to respective topics.

The outputs from DBW node are throttle, steering, and brake commands published to throttle\_cmd, steering\_cmd, and brake\_cmd respectively.

Implementation

The dbw\_node.py logic calls the Controller and Control objects based on linear\_vel, angular\_vel, current\_vel, and dbw\_enabled to produce throttle, brake, and steering commands. If DBW node is enabled, throttle, braking and steering computed through the Controller will be published to /vehicle/throttle\_cmd, /vehicle/braking\_cmd, and /vehicle/steering\_cmd respectively.

The Controller logics within the twist\_controller.py employs the PID.py to give a control on throttle command. The steering commands are calculated through yaw\_controller.py. Both throttle and steering commands are smoothed by a low pass filter from lowpass.py.

Phase 3: Traffic Light Detection

The Perception subsystem here senses the surrounding world for traffic lights (in this project, obstacles are not detected), and publishes useful messages to other subsystems. The traffic light detection node is a core element of the solution as it informs about the presence and state of traffic lights based on the images it receives from the camera. This node subscribes to the data from the /image\_color, /current\_pose, and /base\_waypoints topics, and publishes the stop line to the nearest red traffic light to the topic /traffic\_waypoint. The input messages are the car's current position (from /current\_pose topic), camera images (from /image\_color), and a complete list of waypoints (from /base\_waypoints), while the output of the detection and classification node is the state of the traffic light, and the index of the closest stop line (-1 if not exists)(publishes to /traffic\_waypoint). The module consists of two parts:

1. Traffic light classifier. The classifier uses a TensorFlow based CNN (ssd\_inception\_v2\_coco) for object detection. The model has been trained with a udacity provided data set. Given that the real-world scenario and the simulator are quite different, we created to different trained models – one for each scenario.

A large amount of effort went into labeling and augmenting the training data which is based on the provided ros-bag file for the real-world scenario and simulator images we saved from driving on the simulator track.   
We than experimented with different CNN models to find a good balance between inference speed and accuracy.

|  |  |  |
| --- | --- | --- |
| **Model** | **Inference speed [ms]** | **Accuracy** |
| ssd\_mobilenet\_v2\_coco | 19.9 | TBD |
| ssd\_inception\_v2\_coco | 27.8 | TBD |
| faster\_rcnn\_inception\_v2\_coco | 79.7 | TBD |
| faster\_rcnn\_resnet101\_coco | 201 | TBD |

It turned out that ssd\_inception\_v2\_coco provides a good balance between speed and accuracy. The following images show some samples of test data which we used to verify the trained model.

Simulator test samples:  
  
A screenshot of a traffic light

Description automatically generated  
  
Real-world parking lot test samples:

A picture containing ground, tree, outdoor, window

Description automatically generatedA picture containing window, tree, outdoor, ground

Description automatically generated

1. The traffic light detection uses the information provided by the traffic light classifier to perform a traffic light detection. The *get\_light\_state( )* function can determine the current color of the traffic light (ID of traffic light color, UNKNOWN=4, GREEN=2, YELLOW=1, RED=0). The traffic light state detection and classification was finished by the function get*\_classification(image)* in class *TLClassifier* which is coded in *tl\_classifier.py*. The *process\_traffic\_lights()* can finally find the closest visible traffic light (index of waypoint closes to the upcoming stop line for a traffic light), and determines its location and color in *tl\_detector.py*.

However, the current detected state is not regarded as the predicted traffic light state. The predicted state has to occur STATE\_COUNT\_THRESHOLD（here = 3) number of times till we start using it; otherwise the previous stable state is used. This is applied as a damper to avoid the sudden velocity change, and smooth the vehicle behaviour. The *image\_cb( )* identifies the upcoming red light at camera frequency, and publishesthe index of the waypoint closest to the red light's stop line to the topic */traffic\_waypoint*.

Phase 4: Waypoint Updater (Full)

Description:

The Waypoint Updater (full) is the extension of the phase 1 Waypoint updater, which publishes the final waypoints based on traffic light detections or obstacle detections. The final waypoints published by this node considers the calculation of the velocity which increases and decreases on the traffic signal situations. The velocity will be reduced when the traffic signal changes to RED and the velocity will increase when the traffic signal changes to GREEN.

The inputs and outputs are already described in the Phase 1: Waypoint\_updater (Partial) section.

Implementation:

The target velocity is set for the waypoints leading up to the red traffic lights to bring the vehicle to a smooth stop.

The velocity is calculated based on the following formula. And in the graph, we can see how the velocity is gradually decreasing instead of a linear reduction, taking into account the max deceleration and the stopping distance. The max deceleration is set to 0.5 m/s^2 and stopping distance is calculated based on the closest id of the red traffic light.

Velocity

Waypoints

This way the Waypoint\_updater publishes the final waypoints considering the target velocity of the car to the waypoint follower, which again is sent to the DBW node which controls the braking and accelerating the car.

Open Topic Discussion:

*“Topics that could be used to improve the performance of the vehicle”*

Conclusion