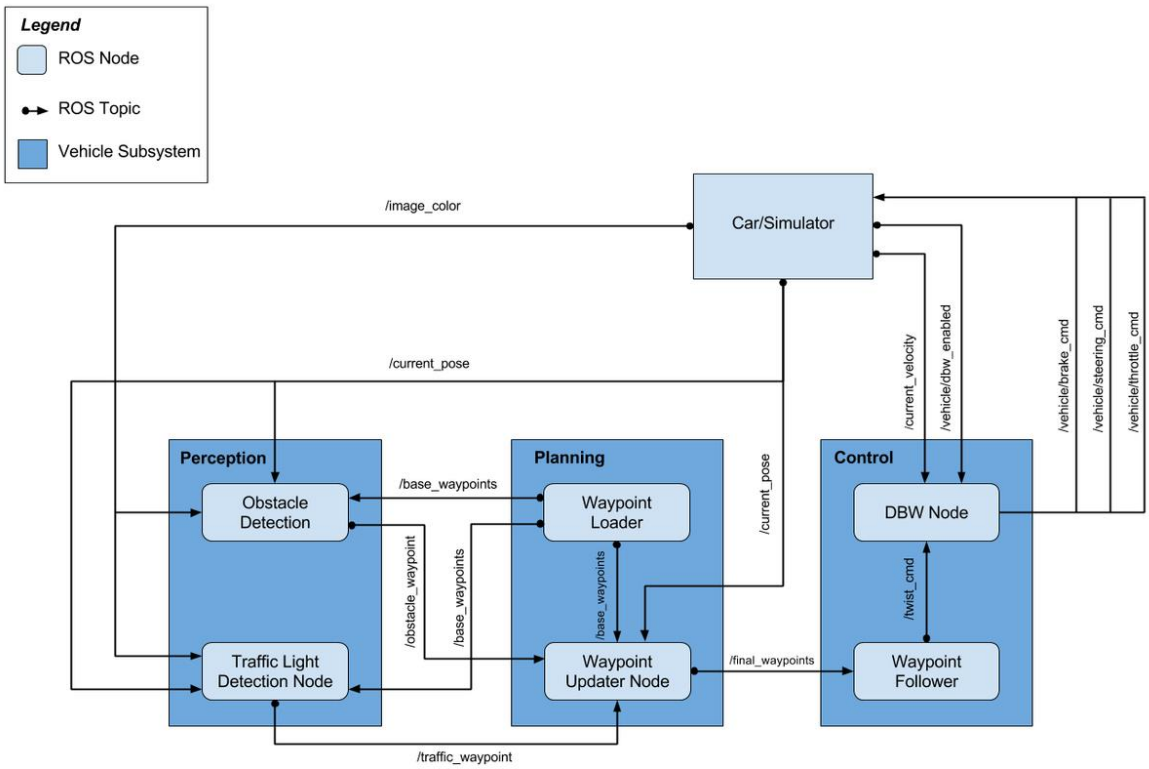
Programming a Real Self-Driving car

For this project, you'll be writing ROS nodes to implement core functionality of the autonomous vehicle system, including traffic light detection, control, and waypoint following! You will test your code using a simulator, and when you are ready, your group can submit the project to be run on Carla.



Team members

- Asad Zia - asadzia@gmail.com - UTC +5 - Lahore, Pakistan
- Ferran Garriga - zegnus@gmail.com - UTC+0 - London, UK
- Leon Li - asimay_y@126.com - UTC+8 - Guangzhou, China
- Shahzad Raza - raza.shahzad@gmail.com - UTC+4 - Dubai, United Arab Emirates
- Mike Allen - mikeleonardallen@gmail.com - UTC-7 - Los Angeles, US

External Resources

- Self-Driving Car Engineer: Final Project Teams - Team 3 Start: February 1

- How to format this document here
- ROS cheatsheet

Project Plan Tasks

	Basic Waypoint Updater (further information)
Depends on	Nothing
Description	Complete a partial waypoint updater which subscribes to /base_waypoints and /current_pose and publishes to /final_waypoints
Contributors	1
Responsible	Mike Allen
Status	Complete

	DBW (further information)
Depends on	Basic Waypoint updater
Description	PID and Twist controller. Get the car rolling. Potentially two person, one doing PID other dowing twist controller. After completion of this, the car will start rolling, ignoring Traffic Lights.
Contributors	1 or 2
Responsible	Leon Li and Ferran Garriga as support
Status	Completed

	Traffic Light (further information)
Depends on	Nothing
Description	Detection and classification. I will recommend two persons work on it individually. This will be a competition. We will take the best performing node on simulator. Then other person (whose code we did not merge) can take the winning code and train it on real life camera data from Carla. This way both get credit. You should have access to good GPU.
Contributors	2 or more
Responsible	Shahzad Raza, Mike Allen
Status	Pending (Shahzad)

	Full Waypoint Updater (further information)
Depends on	Basic Waypoint updater, DBW and Traffic Light
Description	Change the waypoint target velocities before publishing to */final_waypoints.* Car should now stop at red traffic lights and move when they are green. One person
Contributors	Anyone free at the time
Responsible	
Status	Not planned

	Integration and testing
Depends on	
Description	Review pull requests, test and merge. This will be ongoing throughout the project
Contributors	1
Responsible	Asaz Zia
Status	

	Bug Fixing and performance improvements
Depends on	
Description	As code is merged and issues are found, anyone can fix a bug and implement a performance improvement.
Contributors	Any
Responsible	
Status	

Set-up instructions and troubleshooting

Repository Information

Fork the GitHub repositoryand follow the readme instructions and create pull request against the main project
We will be doing Pull Requests

Environment Setup

1. Install docker for your mac, windows, ubuntu or others
2. Fork the Github repository and clone to your desktop
3. cd (path_to_project_repo)/CarND-Capstone
4. docker build . -t capstone (here for how to handle images)
5. docker run -p 4567:4567 -v $PWD:/capstone -v <log path on localhost>:/root/.ros/ --rm --name sys-int -it capstone (This will run a container called sys-int. Click here if you have permissions problems)
6. rosdep update
7. Navigate to /capstone/ros/src and run rm CMakeLists.txt (if existing)
8. catkin_init_workspace
9. Navigate to /capstone/ros and run catkin_make
10. source devel/setup.bash

The ROS code will run with:
roslaunch launch/styx.launch

Then in your machine, you can start the car simulator and it will connect to the ros process in docker

Development cycle

1. Make a change in the source code
2. run catkin_make && source devel/setup.sh && roslaunch launch/styx.launch
3. run the simulator
4. If you want to do another change, cancel styx with control+c and go to **1**

Logging and Debugging

1. With styx running, open a new terminal tab

2. docker ps will give you list of running docker processes
3. run docker exec -it <container id> bash to enter same docker process that has styx running.
4. run source devel/setup.sh
5. For logging run tail -f /root/.ros/log/latest/<log>
6. For info run rostopic info /<topic> , etc

Docker Containers

Docker containers do not ship with any text editors included by default. Additionally, any changes made to the containers need to be committed to an image file to be retrievable when the container is spun up again. The items below incorporate some basic housekeeping features.

1. apt-get update
2. apt-get install vim
3. Append the following to .bashrc
    a. This to set some nice colors. Note replace di=1;35 with di=0;36
    b. export EDITOR='vim' to set vim as the default rosed editor
    c. source /opt/ros/kinetic/setup.bash to have the environment sourced in every new terminal session
4. In the root directory, execute:
    a. vi .vimrc
    b. Add lines syntax on and colorscheme murphy

Commit all the changes to a new docker image called capstone:modified using:
docker commit -m <commit_message> <container_name> capstone:modified

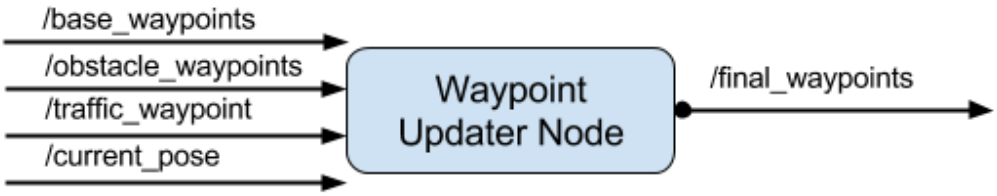
To launch a new session to the running container, execute:
docker exec -it <container_name> /bin/bash

Project Tasks

Will need to modify for the project will be contained entirely within the (path_to_project_repo)/ros/src/ directory

Basic Waypoint Updater

/ros/src/waypoint_detector This package contains the waypoint updater node: waypoint_updater.py. The purpose of this node is to update the target velocity property of each waypoint based on traffic light and obstacle detection data. This node will subscribe to the /base_waypoints, /current_pose, /obstacle_waypoint, and /traffic_waypoint topics, and publish a list of waypoints ahead of the car with target velocities to the /final_waypoints topic.



Steps

The steps that we have followed in order to implement this module are the following:

1. Implement a loop in order to publish at 50hz our results
2. Calculate the closest waypoint from the car's current location among the full list of waypoints injected to the code by the subscription
3. Calculate the next waypoint taking the direction's car is facing into account from the closest waypoint and map's coordinates
4. We will return a sublist of injected waypoints starting from the next calculated waypoint

Implementation description

We have implemented the solution in the file waypoint_updater.py, the major blocks implemented are the following:

Main code loop

We have implemented a main loop that cycles at 50hz and continuously publishes the calculated waypoints while we have some data available.

```
def loop(self):
    rate = rospy.Rate(0.5)
    while not rospy.is_shutdown():
        if self.current_pose is None or self.base_waypoints is None:
            continue


        self.publish()
        rate.sleep()
```

Publish

From the udacity project planning project, we will get the next_waypoint that's closets to the direction of the car while base_waypoints is feeded into the code through a subscription rospy.Subscriber('/base_waypoints', Lane, self.waypoints_cb)

The last step would be to publish the calculated waypoints to our publisher
self.final_waypoints_pub = rospy.Publisher('final_waypoints', Lane, queue_size=1)

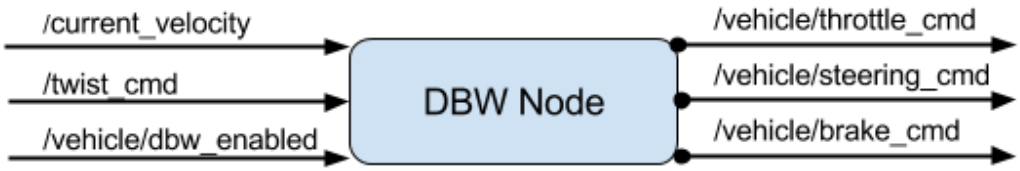
```
def publish(self):
    """publish Lane message to /final_waypoints topic"""


    next_waypoint = self.next_waypoint()
    waypoints = self.base_waypoints.waypoints
    # shift waypoint indexes to start on next_waypoint so it's easy to grab
LOOKAHEAD_WPS
    waypoints = waypoints[next_waypoint:] + waypoints[:next_waypoint]
    waypoints = waypoints[:LOOKAHEAD_WPS]


    lane = Lane()
    lane.waypoints = waypoints
    self.final_waypoints_pub.publish(lane)
```

DBW drive-by-wire

/ros/src/twist_controller Carla is equipped with a drive-by-wire (dbw) system, meaning the throttle, brake, and steering have electronic control. This package contains the files that are responsible for control of the vehicle: the node dbw_node.py and the file twist_controller.py, along with a pid and lowpass filter that you can use in your implementation. The dbw_node subscribes to the /current_velocity topic along with the /twist_cmd topic to receive target linear and angular velocities. Additionally, this node will subscribe to /vehicle/dbw_enabled, which indicates if the car is under dbw or driver control. This node will publish throttle, brake, and steering commands to the /vehicle/throttle_cmd, /vehicle/brake_cmd, and /vehicle/steering_cmd topics.



Steps

The steps that we have followed in order to implement this module are the following:

1. Transform the python code to c++ in order to use helper functions for MPC implementation such as lpopt. **status: in progress, compiles but does not run**
2. Implement dbw_node.cpp
3. Implement twist_controller.cpp.

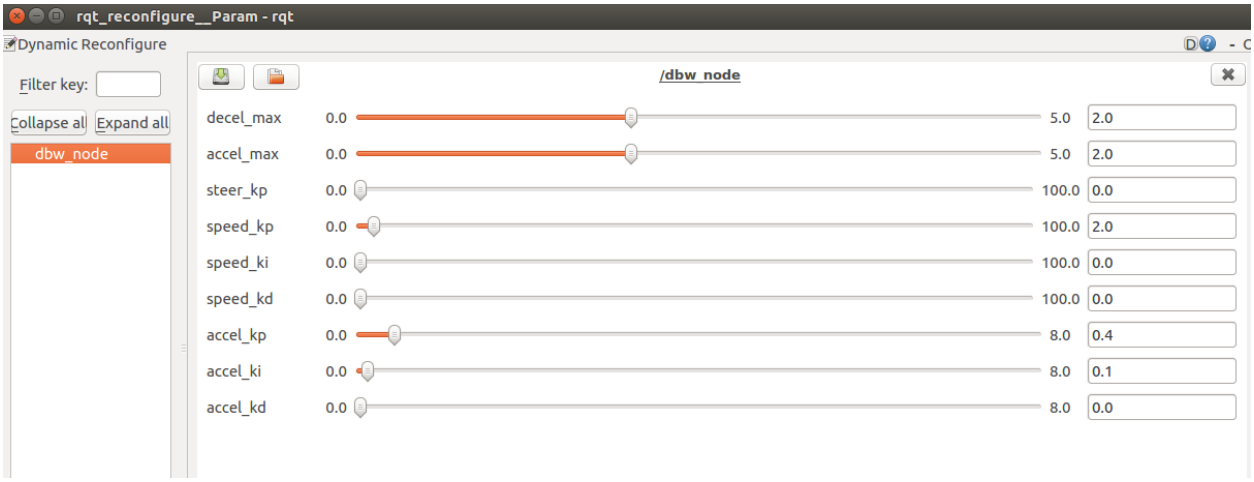
Implementation description

We implement PID control with dynamic_reconfigure plugin first. and then will try MPC control. For PID control, use velocity from /twist_cmd and /current_velocity as velocity CTE, use this value to handle speed_pid controller. we can use dynamic_reconfigure to adjust P,I,D's value to get a good find tuned parameters.

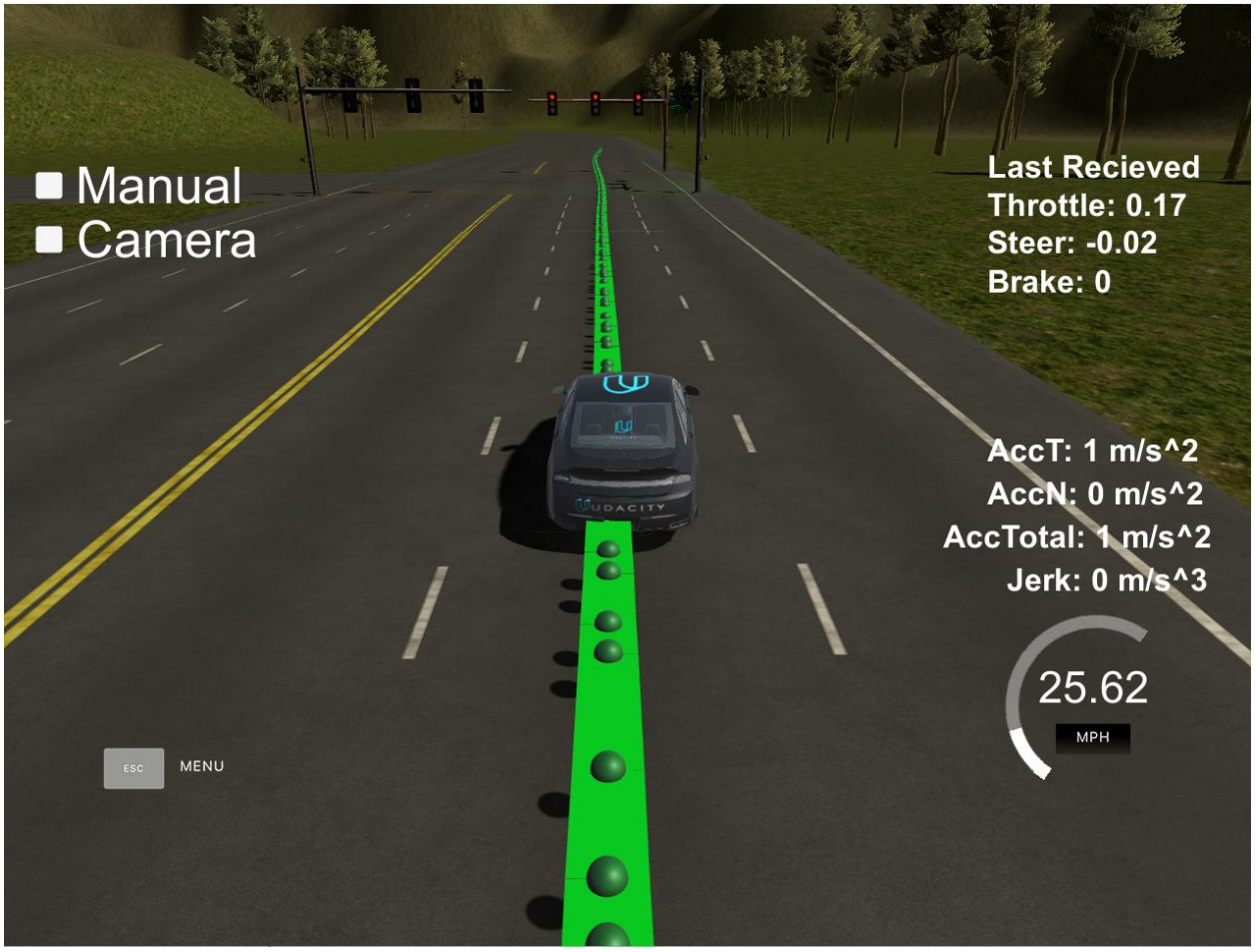
usage:

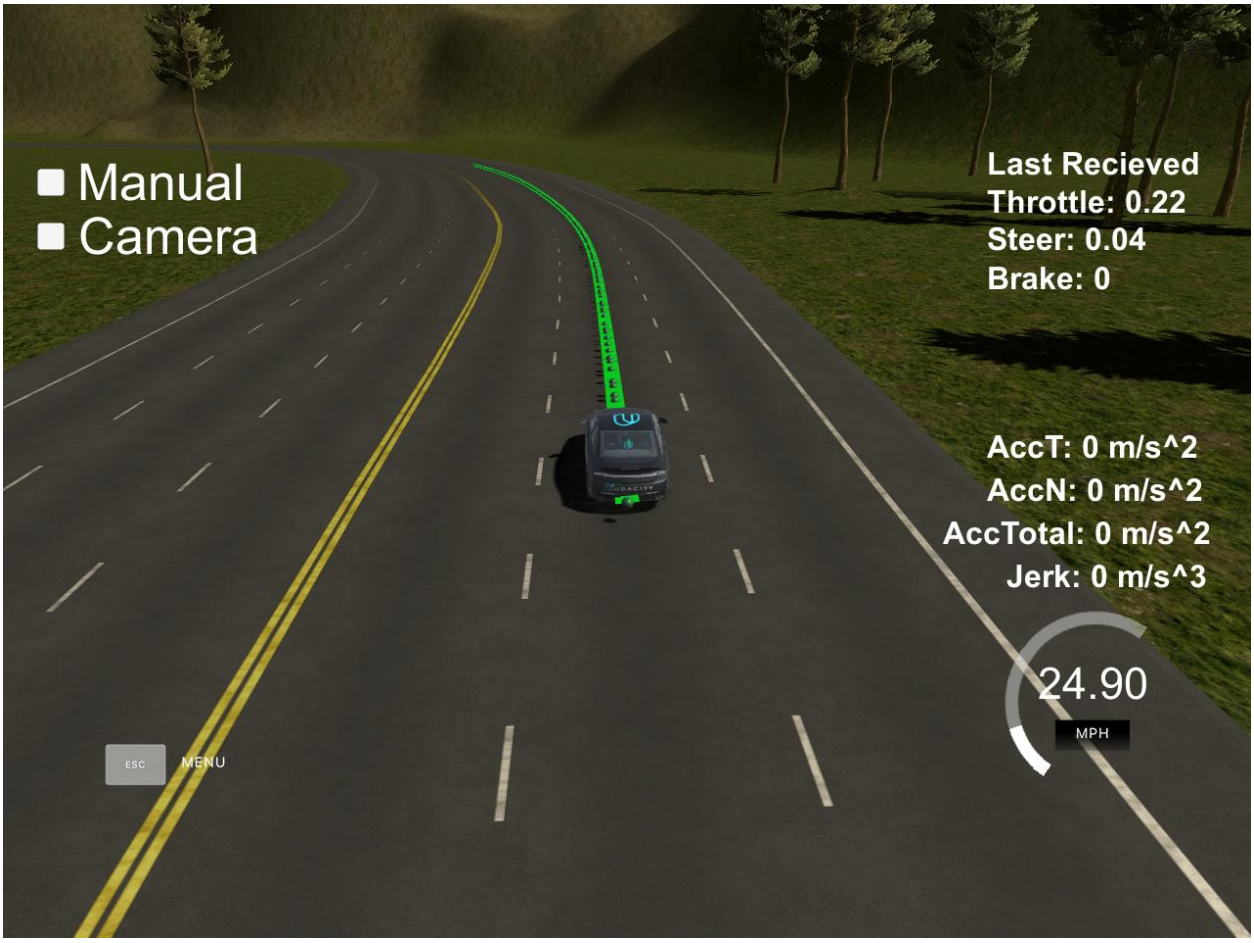
```
$ rosrun rqt_reconfigure rqt_reconfigure
```

to dynamic config the parameters of P,I,D. Following is screen snapshot.



also, we used acceleration pid controller to control the throttle and brake.


Manual
Camera
Last Recieved
Throttle: 0.17
Steer: -0.02
Brake: 0
AccT: 1 m/s^2
AccN: 0 m/s^2
AccTotal: 1 m/s^2
Jerk: 0 m/s^3
25.62
MPH
ESC
MENU



Above is the PID testing screen snapshot, we need to test and tune a good PID value to make car run smoothly. Finally, we tuned the values as below:

```
steer_kp = 3.0;
------
speed_kp = 2.0;
speed_ki = 1.0;
speed_kd = 0;
------
accel_kp = 0.4;
accel_ki = 0.2;
accel_kd = 0.2;
```

We also use lowpassfilter to filter the fuel variable, we assume it is 50% full of gas in simulator environment, but in real environment, we get the value from /fuel_level_report topic.

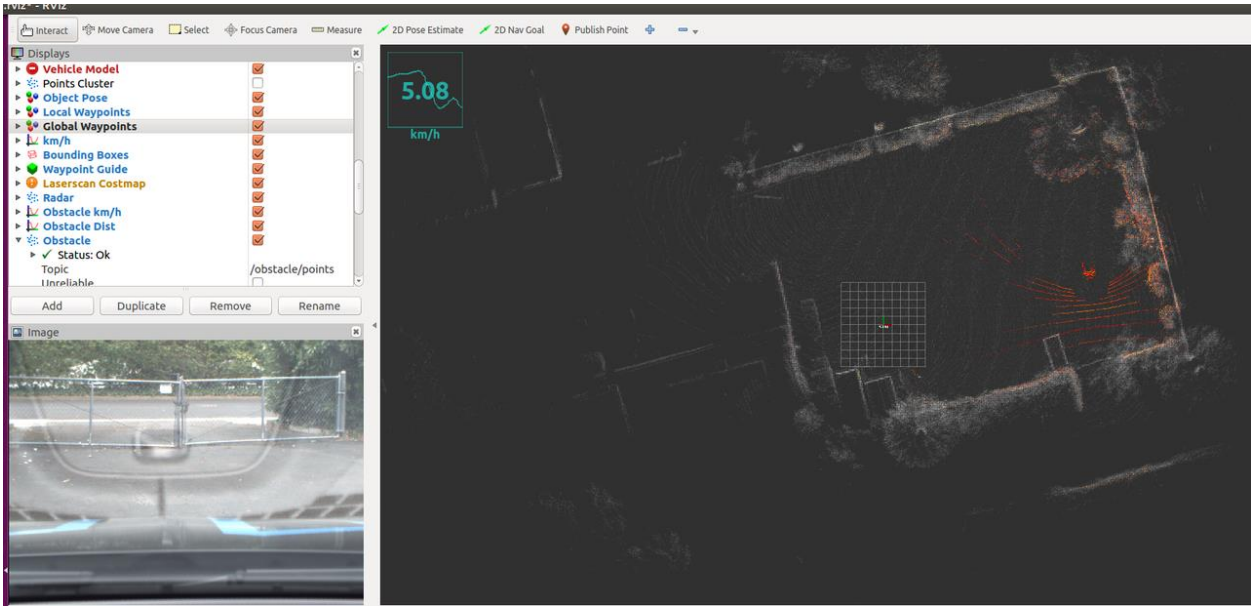
The DBW node subscribe the /current_velocity topic and /twist_cmd topic, we use the velocity from these two topic to do CTE estimation.

We get velocity CTE from:

```
double vel_cte = twist_cmd_.twist.linear.x - cur_velocity_.twist.linear.x;
```

this drive our PID module to move.
there are also some condition check for twist_cmd velocity message, if too small, we reset PID error, in order to avoid accumulated error.

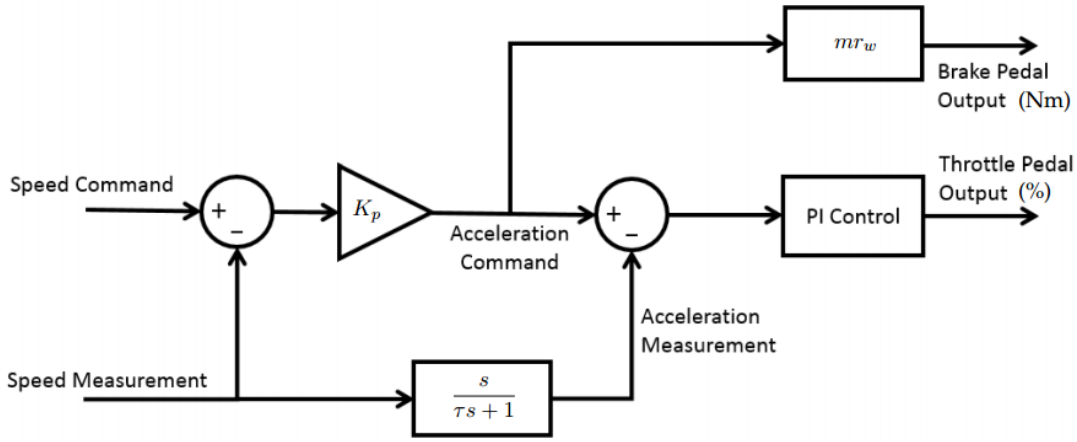
Then we use PID module to drive the vehicle's throttle, brake. we also downloaded the udacity's bagfile to do testing. snapshot as below:



Under this module, the code template is based on Python, but we consider to use MPC, which needs Ipopt lib and which is better supported under c++. So we decide to change the python code template of this module into c++ template. Following is some implementations of c++.

After implementation the PID code, we found that the udacity docker container doesn't support library like Ipopt which is needed for MPC. so we still focus on PID tuning and implementation.

We found the materials about MKZ's control theory, so we did the code as what it said.
Following is the principle of control of MKZ:

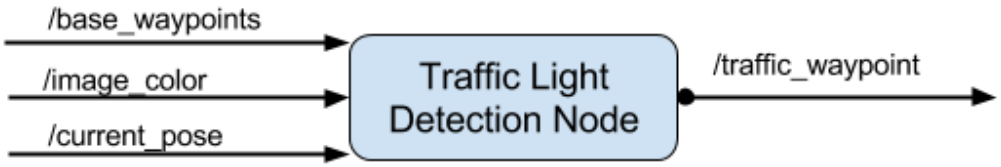


The CTE come from speed command and speed measurement, according to related /twist_cmd and /current_velocity, we get cte from here, and then we use uses proportional control with gain Kp, this produced acceleration command, it is used to multiply m and r, to produce torque, which is T=a*m*r, and combined losspass filter from speed measurement, we produced acceleration pid input, use PI controller to produce throttle pedal output.
During the whole process, we actually used all P,I,D method in it. The detailed info can be checked in code.

Traffic Light

/ros/src/tl_detector This package contains the traffic light detection node: tl_detector.py. This node takes in data from the /image_color, /current_pose, and /base_waypoints topics and publishes the locations to stop for red traffic lights to the /traffic_waypoint topic.

The /current_pose topic provides the vehicle's current position, and /base_waypoints provides a complete list of waypoints the car will be following.
You will build both a traffic light detection node and a traffic light classification node. Traffic light detection should take place within tl_detector.py, whereas traffic light classification should take place within ../tl_detector/light_classification_model/tl_classfier.py.



Steps
[please describe the steps needed for completing this module]

Implementation description
[please describe the major implementation or most interesting code blocks needed for completing this module]

Full Way-point Updater
[project follow-up]

Steps
[please describe the steps needed for completing this module]

Implementation description
[please describe the major implementation or most interesting code blocks needed for completing this module]