User-Manual AAIP

February 2020

Dependencies: ROS python 2.7, OpenCV version 4.1.2 and pytorch 1.4, ROS Melodic

Details on running the simulation or real car project can be found in *README_Project.md* Results:

- **Simulation** - please visit:
  `https://www.youtube.com/watch?v=wZoVcA4jXVs&feature=youtu.be`
- **Real car** - please visit:
  `https://www.youtube.com/watch?v=HdXiDNB-Irk`

1 Perception module

In this section our perception module is described. This module uses camera frame, detect & recognise traffic and traffic lanes.

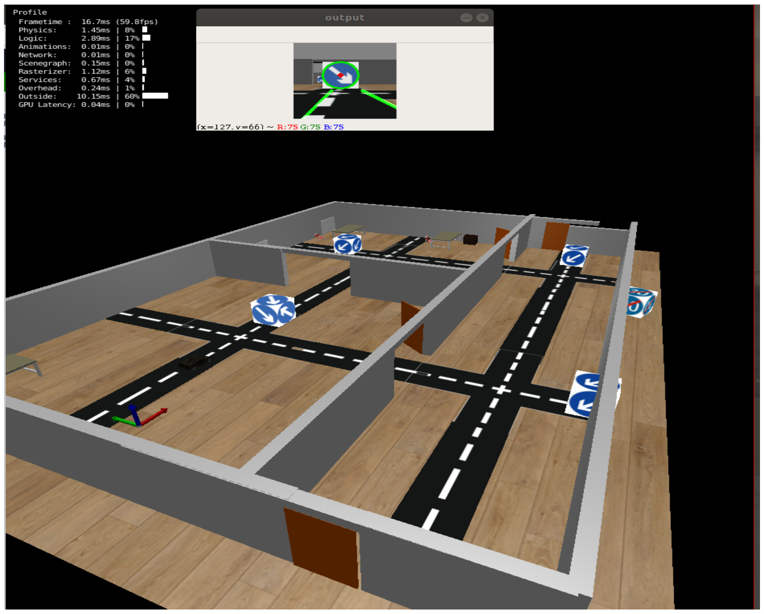
To test it run the simulation fourwd car (*morse run fourwd*) and then run '*roslaunch camera_values camera_perception.launch*'.
Now you should be able to see the simulation environment like in Figure 1.
To test perception module on real car run
'*roslaunch camera_values camera_perception.launch is_simulation:=false*'.

This module is using opencv hough transform to detect traffic sign. To change lane detection parameters see *main_Torch.py* line 639 for real car and line 643 for simulation.

- **Simulation** - $cv2.HoughLinesP(segment, rho = 3, theta = np.pi/180, threshold = 50, lines = np.array([]), minLineLength = 5, maxLineGap = 20)$
- **Real car** - $cv2.HoughLinesP(segment, rho = 1, theta = np.pi/360, threshold = 10, lines = np.array([]), minLineLength = 70, maxLineGap = 20)$

For traffic sign detection we used opencv circle detection. To change circle detection parameters see *main_Torch.py* line 137 for real car and line 146 for simulation.

- **Simulation** - $cv2.HoughCircles(img, cv2.HOUGH_GRADIENT, 1, rows/8, param1 = 100, param2 = 30, minRadius = 1, maxRadius = 50)$


```
Profile
  Frametime :   16.7ms (59.8fps)
  Physics:      1.45ms | 8%
  Logic:        2.89ms | 17%
  Animations:   0.01ms | 0%
  Network:      0.01ms | 0%
  Scenegraph:   0.15ms | 0%
  Rasterizer:   1.12ms | 6%
  Services:     0.67ms | 4%
  Overhead:     0.24ms | 1%
  Outside:     10.15ms | 60%
  GPU Latency: 0.04ms | 0%
```

```
(x=127,y=66) ~ R:75 G:75 B:75
```

Figure 1: Perception module

- **Real car** - $cv2.HoughCircles(img, cv2.HOUGH_GRADIENT, 1, 500, param1 = 100, param2 = 50, minRadius = 20, maxRadius = 200)$

Traffic sign recognition is done with convolutional neural network. Details on how to retrain the network can be seen in *camera_values/src/CNN_Torch.py*

2 Path planning

In this section we will present path planning technique, that we used. For path planning we used spline function. We used path planning in 2 modes: online and offline.

2.1 Path planning online

Online path planning is based on camera perception presented above. Camera perception node computes desired local via points based on traffic lane and traffic sign. We combined local via points with spline function to get the local path. The result should look like in Figure 2. Green line from Figure 2 is the local path provided by camera perception.

2.2 Path planning offline

Offline planning is based on some predefined global via points, it computes the path using spline and publish it to */gobalpath* topic. Result of offline path planning is provided in Figure 3.

To change the offline path planning see *camera_values/src/publishPath.py*

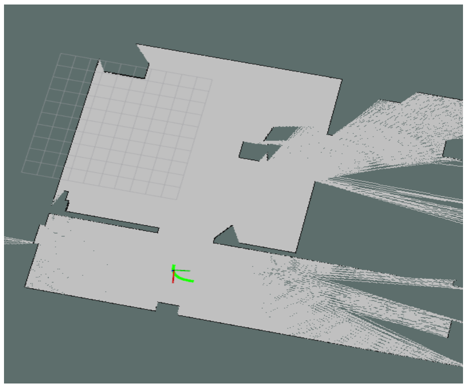


Figure 2: Local planner

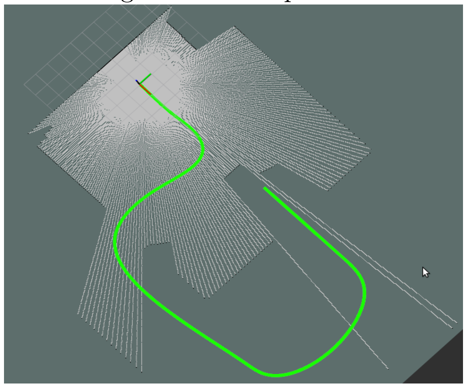


Figure 3: Global planner with known via points

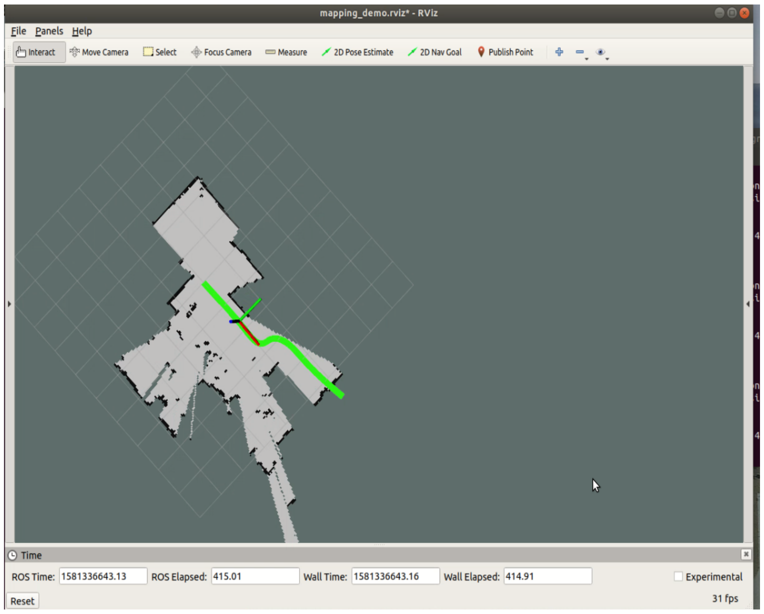
3 Path tracking

Path tracking package requires as input desired via points. It computes the path using this via points and spline function. To follow the path we used low level speed controller and low level steering angle controller. We compute desired Twist messages to publish to cmd_vel topic. PD controller is used to compute desired linear velocity, stanley method is used to compute desired angular velocity. In our project *'rosrun camera_values followPath.py* ' runs the global path tracking.

4 Mapping & Localization

- **Simulation** - For mapping and localization we used hector_slam package. Hector slam is using Extended Kalman Filter for localization, and for that we feeded it with wheel odometry, IMU and lidar data. To run it in the simulation execute: *roslaunch bear_car_launch myfile.launch*

- **Real car** - On the real car we used the same package as before, but the localization did not work. We found that on the real car the wheel odometry data was too noisy. So we decided to remove them, and used Extended Kalman Filter only with lidar. To run slam with lidar data execute: *'roslaunch odometry_agent hectorslam_standalone.launch* '. After running localization, we can run offline path tracking:
  'rosrun move_controller main_car.py ' and the car will follow the path like in Figure 4.



ROS Time: 1581336643.13 ROS Elapsed: 415.01 Wall Time: 1581336643.16 Wall Elapsed: 414.91

Figure 4: Localization on real car

4.1 Path planning A*

A* global planner subscribes to the occupancy grid map from the SLAM which provided the input as which cells are occupied and which are free. It also subscribe to odometry to get the current location of the robot. The A* planner is made as a service. The goal to generate a path from the robot location is given as an input while calling the service. The path is published under the topic '/astar_path'. Figure 5 represents the path generated from A* algorithm.

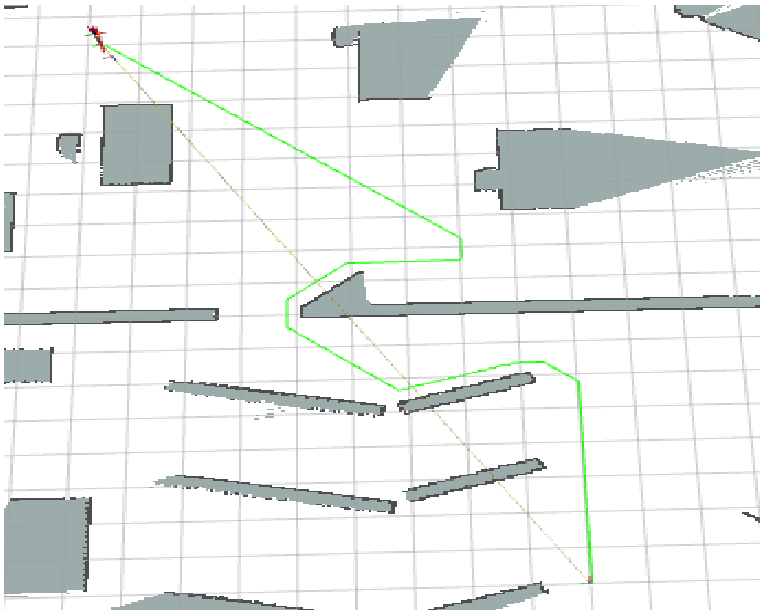


Figure 5: A* global path generated in the simulation