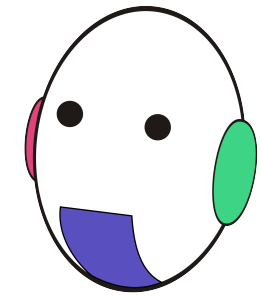


ROBOTIC PROJECT

Yagmur Cigdem Aktas, Dylan Bruzullier

December 2021

Contents

1 Introduction

1.1 Objectives

The goal of this project is create a line follower robot using ROS

The robot will be able to perform autonomous driving on a specific road design for an Auto-race challenge, the left side of the road is a yellow line and the right side is a white line, and this configuration will be used by the packages.

The main objective of this project is to understand how ROS and the functioning of the packages works and the different goals and technical constraints of robotics. We have also learn how manage and how to work in a basic robotics project.

1.2 Organization of Report

This report is simply organize the steps for the usage of line tracking package found in Turtlebot3 AutoRace project.

1.3 Hardware and Software Used for the Project

For installing Linux or ROS on your computer, please refer to the links at the end of the report, or the web.

Software:

ROS version : Melodic
Linux version : 18.04.6

Hardware:

TurtleBot3 Burger version

2 Camera calibration

Before doing the calibration, we have to launch roscore on the remote PC and launch the camera on the robot:

```
$ roscore
```

```
$ roslaunch turtlebot3_autorace_traffic_light_camera turtlebot3_
autorace_camera_pi.launch
```

2.1 Camera Imaging calibration

Before starting to use the camera and calibration, we need to set imaging parameters which determines the basic image view parameters like contrast, sharpness, brightness etc. Launching following commands on the Remote PC, we can select camera/image/compressed topic to see how different values effect the image taken by the Turtlebot3's camera. These parameters should be saved in **camera.yaml** located in **turtebot3autorace_lane_detection_camera/calibration /camera_calibration** folder

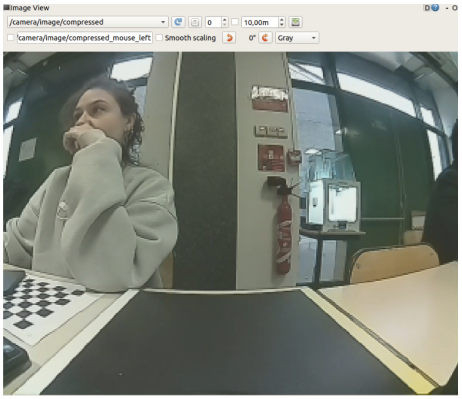
```
$ rqt_image_view
```

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

After examining different values for each parameters, we decided to work with the following parameters which gives us a clear difference between yellow and white lines:

```
camera:
  ISO: 894
  awb_mode: auto
  brightness: 54
  contrast: -13
  exposure_compensation: -5
  exposure_mode: auto
  hFlip: false
  saturation: 0
  sharpness: 87
  shutter_speed: 19853
  vFlip: false
  video_stabilisation: false
  zoom: 1.0
```

Which gives us the following image:

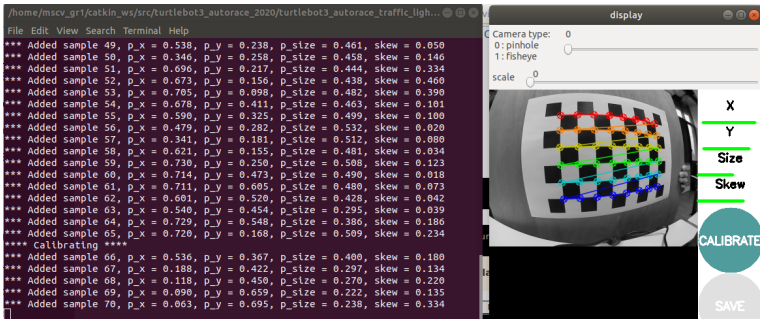


2.2 Intrinsic calibration

After finishing imaging calibration, we apply intrinsic camera calibration to determine intrinsic camera parameters of Turtlebot3 like focal lengths, principal points, distortion coefficients etc. To do that, we use the following commands on Remote PC

```
$ export AUTO_IN_CALIB=calibration
$ export GAZEBO_MODE=false
$ roslaunch turtlebot3_autorace_traffic_light_camera turtlebot3_a
```

To calibrate the robot's camera, we use a chessboard. We translate the chessboard vertically for the x axe and horizontally for the y axe. We move closer and farther for the size and for the skew we twist the chessboard in different directions. After obtaining enough amount of translation and orientation, we see all the variables turn green, that means we are able to click on the Calibrate button to obtain our intrinsic camera parameters which saved to **/tmp/calibrationdata/ost.yaml** file. Then we replace this file with the **/turtlebot3_autorace_lane_detection_camera/calibration /intrinsic_calibration/camerav2_320x240_30fps.yaml** file.



2.3 Extrinsic calibration

After applying intrinsic camera calibration, we obtain calibrated images in **/camera/image_input/compressed** topic. To obtain a better view for the lines, we need to apply extrinsic camera calibration which gives us a projected and compensated version of the image. This projected version will be the one used for lane detection in the next step.

Therefore, extrinsic calibration step contains 2 different configuration.

- Image Compensation
- Image Projection

Image Compensation is a step to scale the image brightness contrast of compressed image according to the **clip_hist_percent** parameter we define. When we set this parameter to 0, we basically take minimum and maximum values of the image and calculate alpha and beta parameters to use cv2.convertScaleAbs function with these minimum maximum values. In case of determining clip_hist_percent parameter to another value than 0, this parameter involves to the process of calculating the alpha and beta values where the image is re-scaled by multiplying with alpha and by adding the beta. Then the absolute value is calculated and the result intensities are casted to an unsigned char (8-bit). The main file for this process is **turtlebot3_autorace_camera/nodes/image_compensation.py**

and the compensated image is published in camera/image_output_compressed topic.

```
# histogram calculation
if clip_hist_percent == 0.0:
    min_gray, max_gray, _, _ = cv2.minMaxLoc(gray)
else:
    hist = cv2.calcHist([gray], [0], None, [hist_size], [0, hist_size])

    accumulator = np.cumsum(hist)

    max = accumulator[hist_size - 1]

    clip_hist_percent *= (max / 100.)
    clip_hist_percent /= 2.

    min_gray = 0
    while accumulator[min_gray] < clip_hist_percent:
        min_gray += 1

    max_gray = hist_size - 1
    while accumulator[max_gray] >= (max - clip_hist_percent):
        max_gray -= 1

input_range = max_gray - min_gray

alpha = (hist_size - 1) / input_range
beta = -min_gray * alpha

cv_image_compensated = cv2.convertScaleAbs(cv_image_compensated, -1, alpha, beta)

if self.pub_image_type == "compressed":
    # publishes compensated image in compressed type
    self.pub_image_compensated.publish(self.cvBridge.cv2_to_compressed_imgmsg(cv_image_compensated, "jpg"))
```

Image Projection is the step to project the image using Homography Matrix calculated by the parameters **top_x, top_y, bottom_x, bottom_y** we determine. As known, Homography matrix calculation requires at least 4 points and these 4 parameters determine the 4 initial points where the 4 destination points are determined stable in the code. Using these 4 pairs, the Homography matrix is calculated then applied to the initial image to obtain projected image.

When we examine the source code **turtlebot3_autorace_camera/nodes/image_projection.py**, we see how our parameters effect the ROI chosen for the homography matrix calculation. This ROI is drawn and published in **'/camera/image_calib/compressed'** topic since we are in calibration mode.

```
if self.is_calibration_mode == True:
    # copy original image to use for cablibration
    cv_image_calib = np.copy(cv_image_original)

    # draw lines to help setting homography variables
    cv_image_calib = cv2.line(cv_image_calib, (160 - top_x, 180 - top_y), (160 + top_x, 180 - top_y), (0, 0, 255), 1)
    cv_image_calib = cv2.line(cv_image_calib, (160 - bottom_x, 120 + bottom_y), (160 + bottom_x, 120 + bottom_y), (0, 0, 255), 1)
    cv_image_calib = cv2.line(cv_image_calib, (160 + bottom_x, 120 + bottom_y), (160 + top_x, 180 - top_y), (0, 0, 255), 1)
    cv_image_calib = cv2.line(cv_image_calib, (160 - bottom_x, 120 + bottom_y), (160 - top_x, 180 - top_y), (0, 0, 255), 1)

    if self.pub_image_type == "compressed":
        # publishes calibration image in compressed type
        self.pub_image_calib.publish(self.cvBridge.cv2_to_compressed_imgmsg(cv_image_calib, "jpg"))
```

The same ROI is used to calculate Homography Matrix with cv2.findHomography() function as well as the Homography Matrix obtained from this function is used with cv2.warpPerspective() function to get projected 1000x600 size output image. This output image is then published in **'/camera/image_output/compressed'** topic.

```
# adding Gaussian blur to the image of original
cv_image_original = cv2.GaussianBlur(cv_image_original, (5, 5), 0)

## homography transform process
# selecting 4 points from the original image
pts_src = np.array([[160 - top_x, 180 - top_y], [160 + top_x, 180 - top_y], [160 + bottom_x, 120 + bottom_y], [160 - bottom_x, 120 + bottom_y]])

# selecting 4 points from image that will be transformed
pts_dst = np.array([[200, 0], [800, 0], [800, 600], [200, 600]])

# finding homography matrix
h, status = cv2.findHomography(pts_src, pts_dst)

# homography process
cv_image_homography = cv2.warpPerspective(cv_image_original, h, (1000, 600))

# fill the empty space with black triangles on left and right side of bottom
triangle1 = np.array([[0, 599], [0, 340], [200, 599]], np.int32)
triangle2 = np.array([[999, 599], [999, 340], [799, 599]], np.int32)
black = (0, 0, 0)
white = (255, 255, 255)
cv_image_homography = cv2.fillPoly(cv_image_homography, [triangle1, triangle2], black)
```

When we examine **turtlebot3_autorace_camera/launch/extrinsic_camera_calibration.launch** file which is the main launch file to run for this section, we see the mapping between compensated and projected image topics. Finally, we obtain compensated over projected images in **'/camera/image_projected_compensated_compressed'** topic.

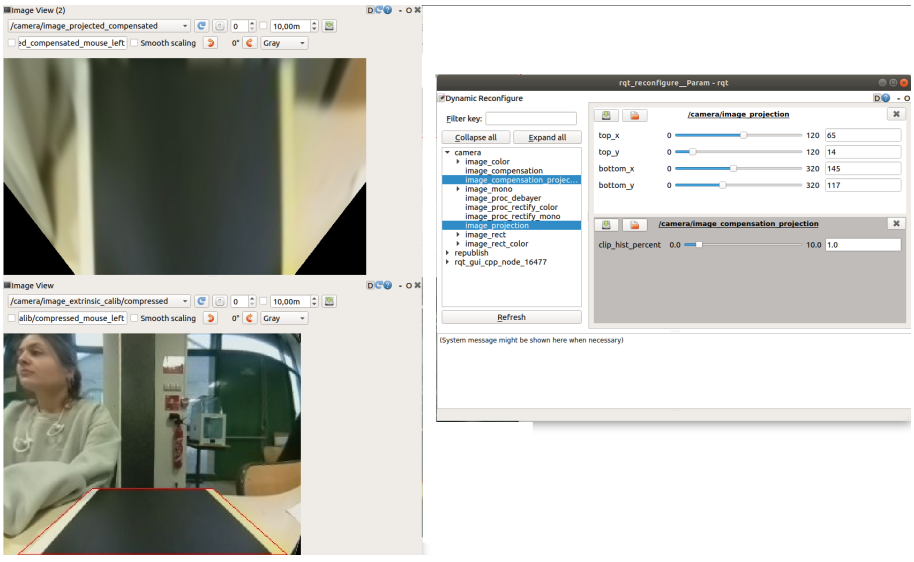
```
  <arg name="mode" default="action" doc="mode type [calibration, action]"/>

  <group ns="camera">
    <!-- image brightness & contrast compensation of ground-projected image -->
    <node pkg="turtlebot3_autorace_camera" type="image_compensation" name="image_compensation" output="screen">
      <rosparam command="load" file="$(find turtlebot3_autorace_camera)/calibration/extrinsic_calibration/compensation.yaml" />
      <param if="$(eval mode == 'calibration')" name="/is_extrinsic_camera_calibration_mode" value="True"/>
      <param if="$(eval mode == 'action')" name="/is_extrinsic_camera_calibration_mode" value="False"/>
      <remap from="/camera/image_input" to="/camera/image_rect_color" />
      <remap from="/camera/image_input/compressed" to="/camera/image_rect_color/compressed" />
      <remap from="/camera/image_output" to="/camera/image_compensated" />
      <remap from="/camera/image_output/compressed" to="/camera/image_compensated/compressed" />
    </node>

    <!-- image ground projection -->
    <node pkg="turtlebot3_autorace_camera" type="image_projection" name="image_projection" output="screen">
      <rosparam command="load" file="$(find turtlebot3_autorace_camera)/calibration/extrinsic_calibration/projection.yaml" />
      <param if="$(eval mode == 'calibration')" name="/is_extrinsic_camera_calibration_mode" value="True"/>
      <param if="$(eval mode == 'action')" name="/is_extrinsic_camera_calibration_mode" value="False"/>
      <remap from="/camera/image_input" to="/camera/image_rect_color" />
      <remap from="/camera/image_input/compressed" to="/camera/image_rect_color/compressed" />
      <remap from="/camera/image_output" to="/camera/image_projected" />
      <remap from="/camera/image_output/compressed" to="/camera/image_projected/compressed" />
      <remap from="/camera/image_calib" to="/camera/image_extrinsic_calib" />
      <remap from="/camera/image_calib/compressed" to="/camera/image_extrinsic_calib/compressed" />
    </node>

    <!-- image brightness & contrast compensation of ground-projected image -->
    <node pkg="turtlebot3_autorace_camera" type="image_compensation" name="image_compensation_projection" output="screen">
      <rosparam command="load" file="$(find turtlebot3_autorace_camera)/calibration/extrinsic_calibration/compensation.yaml" />
      <param if="$(eval mode == 'calibration')" name="/is_extrinsic_camera_calibration_mode" value="True"/>
      <param if="$(eval mode == 'action')" name="/is_extrinsic_camera_calibration_mode" value="False"/>
      <remap from="/camera/image_input" to="/camera/image_projected" />
      <remap from="/camera/image_input/compressed" to="/camera/image_projected/compressed" />
      <remap from="/camera/image_output" to="/camera/image_projected_compensated" />
      <remap from="/camera/image_output/compressed" to="/camera/image_projected_compensated/compressed" />
    </node>
  </group>
</launch>
```

Here is our final extrinsic parameters and result images:



To run extrinsic calibration we first launch intrinsic camera calibration file in action mode instead of calibration mode as before.

```
$ export AUTO_IN_CALIB=action
$ export GAZEBO_MODE=false
$ roslaunch turtlebot3_autorace_traffic_light_camera turtlebot3_a
```

```
$ export AUTO_EX_CALIB=calibration
$ roslaunch turtlebot3_autorace_traffic_light_camera turtlebot3_a
```

2.4 Lane detection calibration

After calibration our camera, we need to apply the last calibration called lane detection calibration where we determine the HSV values for yellow and white lanes. We start launching intrinsic and extrinsic calibration programs in action mode and use the following commands:

```
$ export AUTO_IN_CALIB=action
$ export GAZEBO_MODE=false
$ roslaunch turtlebot3_autorace_traffic_light_camera turtlebot3_a
```

```
$ export AUTO_EX_CALIB=action
$ roslaunch turtlebot3_autorace_traffic_light_camera turtlebot3_a
```

```
$ export AUTO_DT_CALIB=calibration
$ roslaunch turtlebot3_autorace_traffic_light_detect turtlebot3_a
```

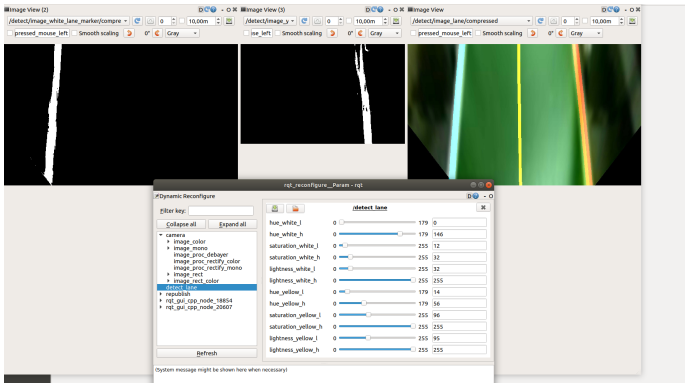
In this step, we have to determine low and high thresholds for white and yellow color as HSV values found in **'/turtlebot3_autorace_detect/param/lane/lane.yaml'** file.

Hue is determined by the dominant wavelength of the visible spectrum. Saturation pertains the amount of white light mixed with a hue where high-saturation colors, contain little or no white light. Lightness is the amount of white pixels mixed in with the color.

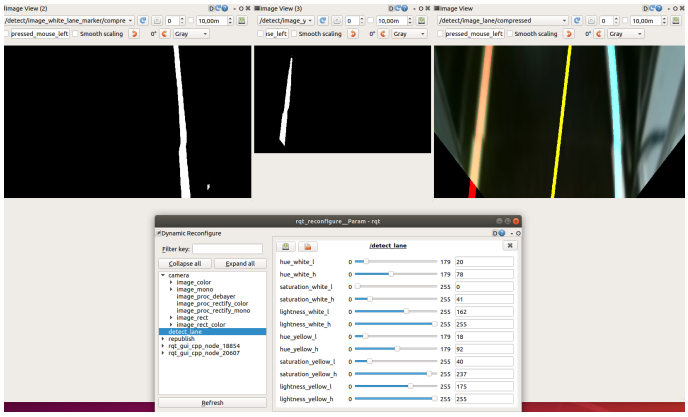
Accordingly, we can make an assumption that optimal white color should have 0 hue, 0 saturation and %100 lightness, while a yellow color may have 60 hue , %100 saturation and %50 brightness. Considering we have a tunnel and a light sign in our path, so that we are not working with these optimal values and the image coming from the Turtlebot3's camera is changing according to the environment light, we found different optimal parameters for each case.

Our parameters for these 3 cases and their results in rqt are as follows:

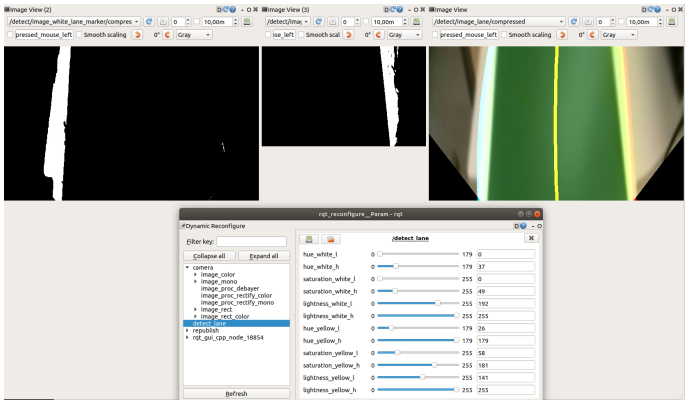
- In Dark:



- In Light Sign:



- In Normal Condition:



3 Challenges with Lane Detection

After different experiments we did with these different lane detection calibration parameters, we observed some common results:

1. If we calibrate the robot with dark light parameters, its able to go through the tunnel but not able to go in the normal or bright light. Or if we use the normal light calibration parameters, the robot can go in normal light but not in the tunnel or in the bright light etc.

2. Sometimes even the same parameters doesn't work for their calibration cases. Normal light calibration parameters may not work after 2-3 days due to the small change of light intensity in the room.

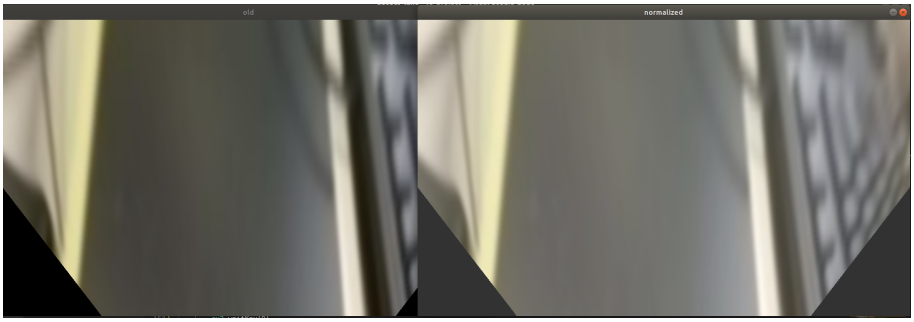
3. Most of the time, false orientation of the robot occurs due to the white line being detected incorrectly or not being detected while yellow lane can

be detected almost in any case. (Dark calibration parameters works in light, normal light calibration parameters works in tunnel etc) In most of the wrong detection cases, white lane is detected in the same place as the correctly detected yellow line. This situation is more dangerous than the first, as the line that the robot must follow is the middle line of these two lines and it is automatically calculated with a margin from detected line if only 1 line is detected such as yellow line which saves the situations that white line was not detected.

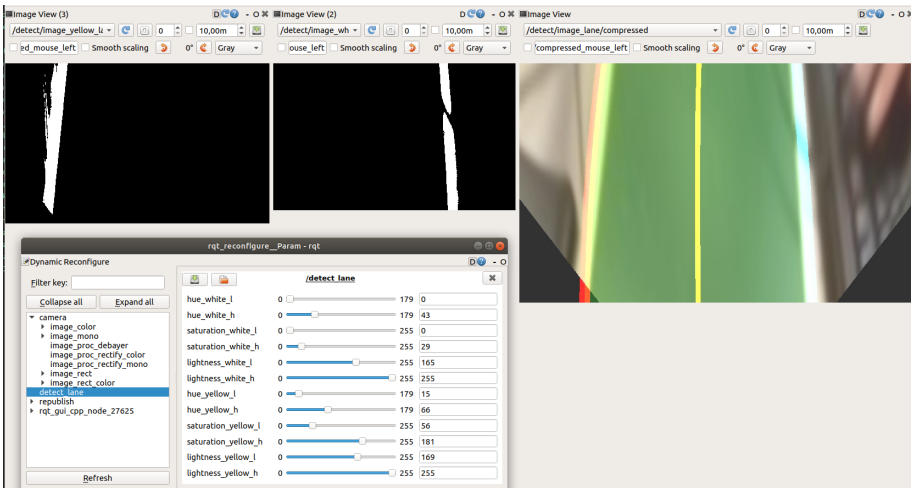
3.1 Experiments for Improving the Performance

For the first problem, we checked the image intensities between an in-tunnel taken image and under normal light taken image. As expected, there was a huge difference and we decided to improve the source code in such a way that if the incoming image has an intensity lower than the threshold we defined, lane detection parameters are changed to the dark mode and otherwise normal light mode is used.

For the second problem, we applied normalization to the incoming image before lane detection calibration. We obtained one type of calibration parameters according to the normalized image and during the action mode, we applied normalization all the images coming from robot's camera before detecting lanes.



We obtained following parameters after calibration with normalized images



For the third problem, we changed the code in such a way that if any yellow lane is detected, the white lane is not involved to the calculation of center point to track. If there is no yellow lane detected, but white lane is detected on the left side of the image that means it's actually yellow lane so we added this control by looking the detected lane placement to determine the line to track correctly from only 1 lane.

4 Conclusion

Changing between dark mode and light mode didn't work with a good performance. Although it solved the tunnel problem somehow it came out to be a new challenge for noise such as changing the mode to the dark mode while the robot should be in light mode due to the light reflection.

Normalization solved the dark - light problem with a very good performance and white lane sensitivity too for most of the cases. The only problem remained with the part where Turtlebot3 tries to take a big corner seeing only white lane and the white lane is detected as yellow lane.

The last approach didn't work for this special case we mentioned in the previous paragraph and this challenge stayed unsolved.