Lane detection using Turtlebot3 Burger

Project for TO52
Weitao YAN
A2020

Course: TO52
Supervising teacher: Zhi YAN, Sihao DENG
Course teacher: Yassine RUICHEK
Dec, 2020

Abstract
The autonomous driving technology is a growing hotspot for engineers and scientists around the world to work on. In this practical project, I did the configuration and parameter tuning of the Turtlebot3 Burger based on ROS in order to detect lanes, which is an essential part of a autonomous driving robot. The project mainly covers the following things: camera intrinsic calibration, camera extrinsic calibration, parameter tuning for lane detection and autonomous driving based on lane detection.

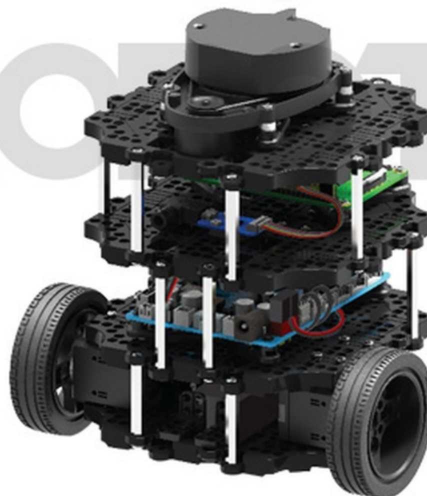


Table of contents

1. Introduction....3
1.1 Hardware....3
1.2 Software....4
2. Technologies....5
2.1 Brief presentation of ROS[1]....5
2.1.1 Project architecture....5
2.1.2 Package....5
2.1.3 Communication framework....6
2.1.4 Communication methods....6
2.1.5 Visualization....7
2.2 Why camera calibration....7
2.2.1 The 4 coordinate systems of a camera[2]....7
2.2.2 Pinhole camera model....8
2.2.3 Intrinsic and extrinsic parameters....9
2.2.4 Camera distortion[7][8][9]....11
2.3 Camera Calibration: Zhang's method....12
2.4 HSL color space....14
3. Project realization....15
3.1 Camera calibration....15
3.1.1 Basic parameters....15
3.1.2 Intrinsic calibration....15
3.1.3 Extrinsic calibration....17
3.2 HSL parameter tuning for lane detection....17
3.3 Autonomous driving with lane detection....19
3.4 Difficulties encountered....19
4. Conclusion....22
4.1. The limitation of the pinhole camera model....22
4.2. What I learned from the project....23
4.3. Appreciations....23
5. References....23

1.Introduction

A self-driving car is a car that can be driven with few or no people. The self-driving car can sense its environment and navigate without human operation. Although fully autonomous vehicles have not yet been fully commercialized, autonomous vehicles have recently received great attention, largely due to the rapid development of artificial intelligence (AI) technology.

Autonomous driving is a huge and complex project, involving many technologies. It intensively uses technologies such as computers, modern sensing, information fusion, communications, artificial intelligence and automatic control, and is a typical high-tech complex. The key technologies of autonomous driving can be divided into four parts: environment perception, behavior decision-making, path planning and motion control. Let's briefly introduce the composition of autonomous driving technology from both hardware and software aspects.

1.1 Hardware

- Radar and Lidar
  The advantage of lidar is that it has a high range accuracy resolution and can have a stable detection of physical dimensions. But the shortcomings are also obvious. The most serious is that in engineering, although there is a solid-state lidar, the life expectancy can not catch up with that of a vehicle. And the cost has not yet met the requirements of the public market. In addition, although accurate distance perception is his advantage, it is also his disadvantage in rain, snow, etc. For example, it cannot correctly judge the attributes of objects.

Figure 1 LDS-01 Lidar in Turtlebot3 Burger

- Camera
  Unlike lidar, the camera collects pixel information, which is similar to what the human eye sees. Unlike humans, the human eye is a natural super intelligent processor, which can easily recognize lanes, vehicles and pedestrians. While for vehicles, pixel information is just nothing but numbers. The massive amount of

data must go through complex processes such as abstraction and reconstruction using deep learning technology.
Since the perception of cameras is almost the same as that of humans, deep learning algorithms are also used most in cameras. At present, the camera technology of mass-produced cars is also very mature and low in cost, and the main cost comes from software. Therefore, if the visual software does well, it can be the king of autonomous driving. If the software does not do well, the camera is a decoration. Although camera has the same problems as people, it can't handle darkness and glare well, but its resolution advantage and hidden information can solve any problem theoretically. After all, we people also rely on vision.

Figure 2 Raspberry Pi Camera v2.1

In this project, I rely mainly on this camera to carry out all the lane detection configurations.

1.2 Software

The software contains four layers: perception, fusion, decision-making, and control.
First, the system collects information from the sensor, but after receiving the information from the sensor, it will be discovered that not all information is useful. Since the state of the sensor is not 100% effective, it is extremely irresponsible for the subsequent decision to determine whether there is an obstacle in front of it only based on the signal of a certain frame (maybe the sensor has misdetected). Therefore, it is necessary to pre-process the information to ensure that the obstacles in front of the vehicle always exist, rather than passing by in a flash.
Decision-making refers to how to plan the action correctly after obtaining the fused data. The plan includes longitudinal control and lateral control: longitudinal control means speed control, which is when to accelerate and when to brake; transverse control means behavior control, which is when to change lanes, when to overtake, etc.

In this project, it is mainly the decision-making part that I concern. The decision, such as turning left and turning right should be made according to lane information extracted by the camera with delicate calibration.

2.Technologies

2.1 Brief presentation of ROS[1]

ROS, namely Robot Operating System, sounds very much like an operating system. However, it is not. ROS is actually a middleware that connects the real system and the robot programs. It has multiple operating system-like functionalities, such as hardware abstraction, control of bottom layer devices, messaging between processes and package management.
In ROS, a node is actually a process. ROS uses a distributed framework that can run multiple processes simultaneously and provides the management and communication between nodes.

2.1.1 Project architecture

- Catkin

Catkin is a ROS-customized compile and building system extended from CMake.

- Catkin workspace

Catkin workspace is the folder that manages all the packages. It is compiled using catkin. The ‘src’ folder contains source code of packages. The ‘build’ folder contains cache of Catkin/CMake and middle files for compiling. The ‘dev’ folder contains target files, say, head files, link libraries and executable files.
All the work and programming things we do are situated in ‘src’ folder.
In ‘src’ folder, we can find many packages. A package is the basic compile unit in Catkin.

2.1.2 Package

A package contains one or multiple executable files(nodes). A package always contains a ‘CMakeLists.txt’ file and a ‘package.xml’ file. The ‘CMakeLists.txt’ file defines Catkin compile rules. The ‘package.xml’ file defines the attributes of the package.
We can program scripts, say, shell or Python, in a package. They are stored in the ‘scripts’ folder. Sometimes we also program in C++ whose head files are in ‘include’ folder and source files are in ‘src’ folder.
Some custom communication formats are also defined in the package, such as

message(msg), service(srv) and action(action).
A package also contains launch file(in 'launch' folder) and yaml file(in 'config' folder). Launch file allows us to executes multiples nodes at the same time.

2.1.3 Communication framework

- master – the node manager
  Each node registers at master before it starts. The master manages the communication between nodes. This is why we need to enter '*roscore*' every time we start ROS because it starts master.
- node – a process in ROS
  Node is a live instance of the executable file of a package.
  Using '*rosrun [pkg_name] [node_name]*' to start a node.
  Using '*rosnode list*' to list information of all current running nodes.
  Using '*roslaunch [pkg_name] [file_name.launch]*' to start master and multiple nodes at the same time. The launch file defines the rule for nodes to start.

2.1.4 Communication methods

ROS mainly has Topic, Service, Parameter Service and Actionlib as its communication methods.

- Topic
  Using predefined topic to communicate between nodes. It uses publish-subscribe method. Topic acts as a channel that the nodes can publish and subscribe. Topic is an asynchronous method. Message is the data type defined for topic. It is stored in '.msg' file.
- Service
  It uses synchronous request-reply method. It only acts when it is requested. Service is very useful in occasionally called tasks. Multiple clients requests one server. Srv is the data type defined for service. It is stored in '*.srv' file.
- Parameter Server
  It maintains a dictionary that stores parameters.
- Action
  A server-like communication method with status feedback. It is usually used in time-consuming and preemptive tasks. Action is the data type defined for Action. It is stored in '.action' file.

2.1.5 Visualization

rqt is the visualization tool for ROS. It is based on Qt. It is very helpful for understanding the communication framework of a ROS project.
Use '*rqt_graph* ' to show the current communication framework. For example, what nodes and topic are running, the message flow, etc.
Use '*rqt_plot* ' to plot data. For example, the speed of robot or IMU data.
Use '*rqt_console* ' to look up logs.

In a node graph('*rqt_graph* '), a circle represents a node. The arrow between nodes specifies the topic. When switching to 'Node/Topics(all)' mode, the rectangles represents topics.

In '*rqt_plot* ', we can monitor current robot value by specifying the topic name.

2.2 Why camera calibration

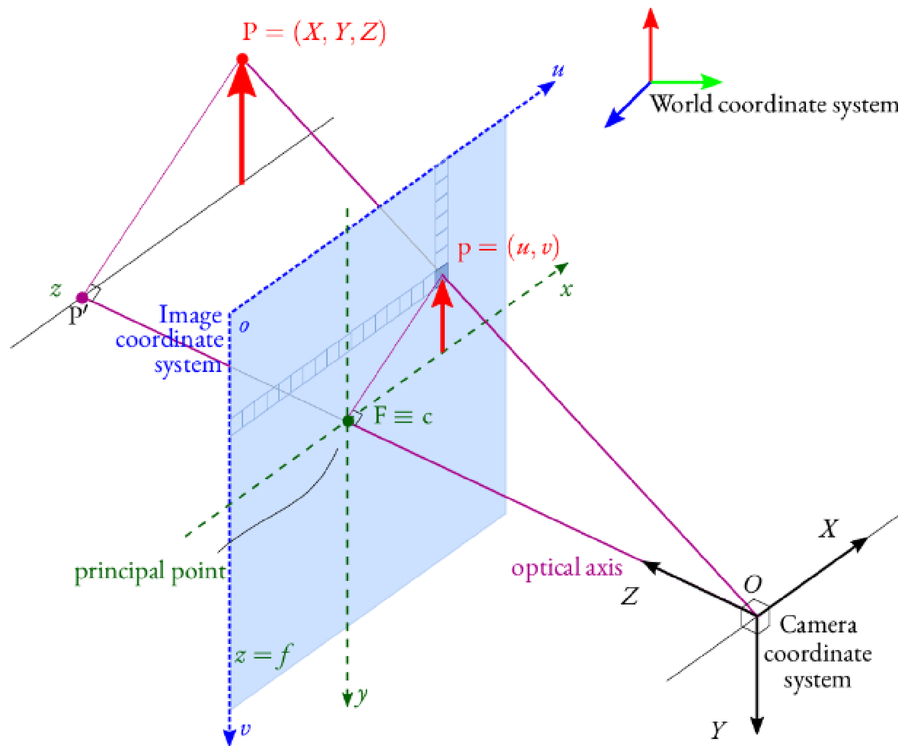
The camera model is the key to all future calibration algorithms. Only a thorough understanding of that can we have a better understanding of future calibration algorithms.

2.2.1 The 4 coordinate systems of a camera[2]

First, we must understand the relationship between the four plane coordinate systems in the camera model.
In the camera model, a certain point in the three-dimensional world and its corresponding pixel are obtained through the conversion of the coordinate system. Four coordinate systems are involved in this process, namely the world coordinate system, the camera coordinate system, the image coordinate system, and the pixel coordinate system. The conversion process of the four coordinate systems will be described in detail below.

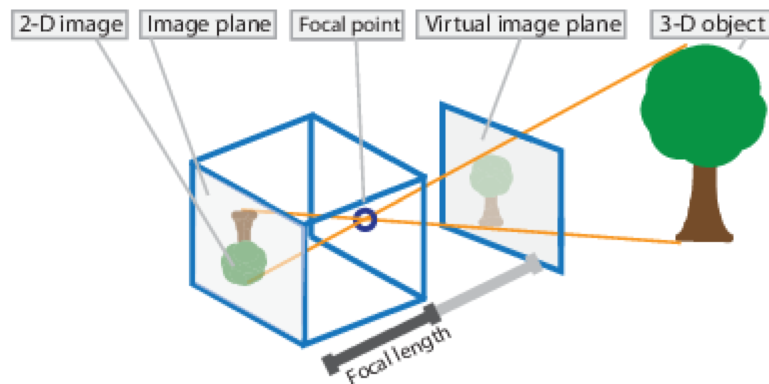
- World is the world coordinate system, you can arbitrarily specify the xw axis and yw axis, which is the coordinate system where the point **P** in the figure below is located.
- Camera is the camera coordinate system, the origin is located in the pin hole **O**, the z axis coincides with the optical axis, and the x axis and y axis are parallel to the projection plane, which is the coordinate system **XYZ** in the figure below.
- Image is the image coordinate system, the origin is at the intersection of the optical axis and the projection plane, and the x axis and y axis are parallel to the projection plane, which is the coordinate system **xyz** in the figure below.
- Pixel is the pixel coordinate system. Seen from the pin hole to the projection surface, the upper left corner of the projection surface is the origin, and the **uv** axis

coincides with the projection surface. The coordinate system and the image coordinate system are in the same plane, but the origin is different.



2.2.2 Pinhole camera model

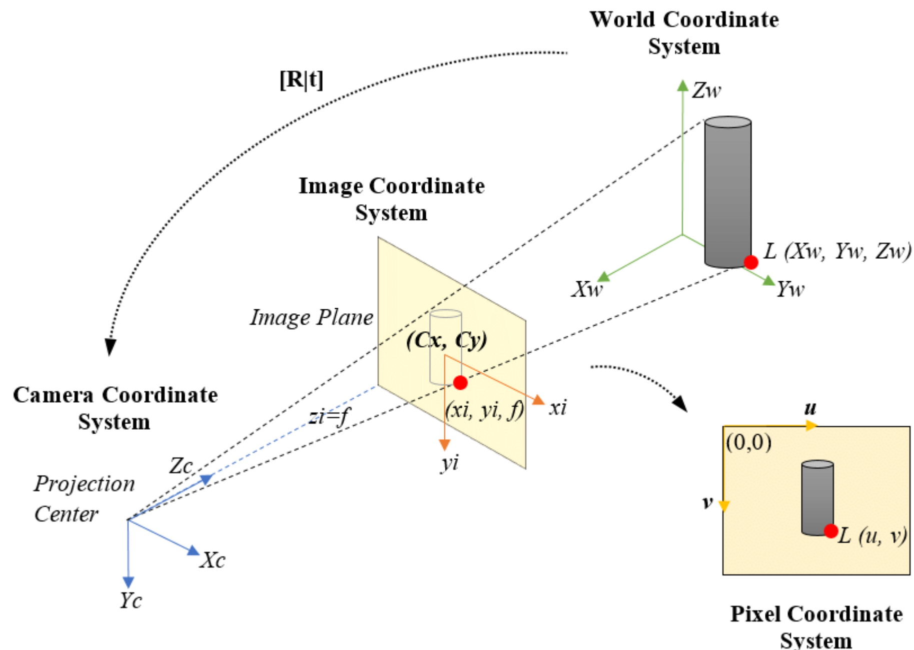
[1]The camera imaging process is actually the process of mapping the three-dimensional points in the real three-dimensional space to the imaging plane (two-dimensional space). You can simply use the pin hole imaging model to describe the process to understand the process of space transformation from the three-dimensional space to the two-dimensional image during the imaging process.



The camera can be abstracted into the simplest form: a small hole and an imaging plane. The small hole is located between the imaging plane and the real three-dimensional scene. Any light from the real world can only reach the imaging plane through the small

[1] https://www.mathworks.com/help/vision/ug/camera-calibration.html

hole. Therefore, there is a correspondence between the imaging plane and the real three-dimensional scene seen through the small hole, that is, there is a certain transformation relationship between the two-dimensional image points in the image and the three-dimensional points of the real three-dimensional world. Once this transformation relationship is found, the two-dimensional point information in the image can be used [2]to restore the three-dimensional information of the scene.[3]



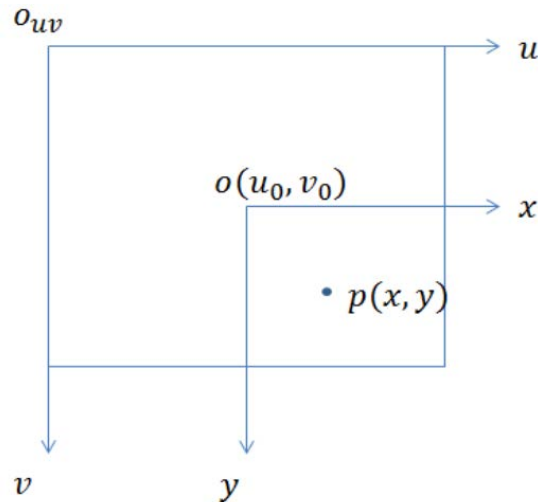
Since cameras and objects can be placed at any position in the environment, it is necessary to select a reference coordinate system in the environment to describe the position of the camera, and use it to describe the position of any object in the environment. This coordinate system is called the world coordinate system. The world coordinate system is an imaginary coordinate system, used as a general reference, and can be freely defined as required. Usually, the world coordinate system is defined to coincide with the camera coordinate system of the camera.[4]

2.2.3 Intrinsic and extrinsic parameters

- Intrinsic parameters[5][6]

  Consider the transform from **image** coordinate system to **pixel** coordinate system. Since the image coordinate system and the pixel coordinate system are on the same plane, the difference between the two lies in the position and unit of the coordinate origin. The origin of the pixel coordinate system is at the upper left corner of the image coordinate system, and the unit of the pixel coordinate system is pixels.

[2] https://www.researchgate.net/figure/Pinhole-Camera-Model-ideal-projection-of-a-3D-object-on-a-2D-image_fig1_326518096



So we can get the following equations:

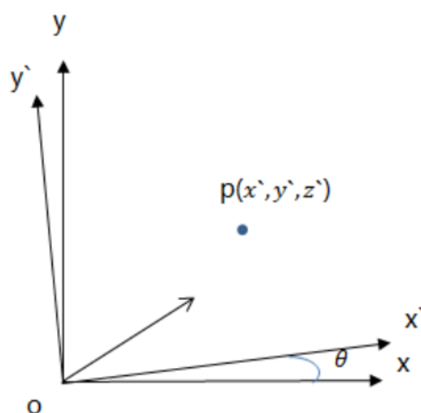
$$u = \frac{x}{dx} + u_0, v = \frac{y}{dy} + v_0 \qquad \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{dx} & 0 & u_0 \\ 0 & \frac{1}{dy} & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

where dx, dy represent the width and the height of a pixel in the pixel coordinate system respectively and u0, v0 represent the shift of horizontal and vertical coordinates of the origin from image coordinate system to pixel coordinate system. The four parameters above are the intrinsic parameters.

- Extrinsic parameters[5][6]

Consider the transform from **world** coordinate system to **camera** coordinate system, a process that includes rotation and translation.

For rotations, we have:



$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x` \\ y` \\ z` \end{bmatrix} = R_1 \begin{bmatrix} x` \\ y` \\ z` \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\varphi & sin\varphi \\ 0 & -sin\varphi & cos\varphi \end{bmatrix} \begin{bmatrix} x` \\ y` \\ 1 \end{bmatrix} = R_2 \begin{bmatrix} x` \\ y` \\ z` \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} cos\varphi & 0 & -sin\varphi \\ 0 & 1 & 0 \\ sin\varphi & 0 & cos\varphi \end{bmatrix} \begin{bmatrix} x` \\ y` \\ 1 \end{bmatrix} = R_3 \begin{bmatrix} x` \\ y` \\ z` \end{bmatrix}$$

Rotation along z axis

R1, R2, R3 represent the rotation matrix for the rotations along **z, x, y** axis with the angle **θ, φ, ω** respectively. Then we get R=R1*R2*R3 which is the overall rotation matrix. Add it to a translation vector T:

$$T = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \qquad \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = R \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} + T$$

Finally, we get the transform equation and the extrinsic parameters:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ 1 \end{pmatrix} = \begin{pmatrix} R & T \\ 0^3 & 1 \end{pmatrix} \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \qquad \begin{pmatrix} R & T \\ 0^3 & 1 \end{pmatrix}$$

2.2.4 Camera distortion[7][8][9]

Distortion can generally be divided into: radial distortion, tangential distortion.
It is an offset to rectilinear projection. Generally speaking, straight-line projection means that a straight line in the scene is projected onto the picture and remains as a straight line. Distortion simply means that a straight line projected onto the picture cannot be maintained as a straight line.
There are other types of distortion, but they are ignorable compared with radial distortion and tangential distortion.

- Radial distortion

The radial distortion comes from the lens shape. The light passing through the edge of the lens is prone to radial distortion. The farther the light is from the center of the lens, the greater the distortion.
The image below shows distortion: From left to right, normal distortion, barrel distortion, pincushion distortion.

Anti-distortion model:

$$\begin{cases} x_d = x(1 + k_1 r^2 + k_2 r^4 + + k_3 r^6) \\ y_d = y(1 + k_1 r^2 + k_2 r^4 + + k_3 r^6) \end{cases} \qquad r = \sqrt{x^2 + y^2}$$

x, y are the normalized image coordinates, which means the coordinate origin has been moved to the principal point, and the pixel coordinates are divided by the focal length. k1, k2, k3 are the radial distortion coefficients

- Tangential distortion

  Tangential distortion comes from the entire camera assembly process. When the camera sensor and the lens are not parallel, because there is an angle, when the light passes through the lens to the image sensor, the imaging position changes. Anti-distortion model:

$$\begin{cases} x_d = x + [2 p_1 xy + p_2 (r^2 + 2x^2)] \\ y_d = x + [2 p_2 xy + p_1 (r^2 + 2y^2)] \end{cases} \qquad r = \sqrt{x^2 + y^2}$$

  x, y are the normalized image coordinates, which means the coordinate origin has been moved to the principal point, and the pixel coordinates are divided by the focal length. p1 and p2 are the tangential distortion coefficients.

2.3 Camera Calibration: Zhang's method

In the pinhole camera model, as long as these 9 parameters (dx, dy, u0, v0, k1, k2, k3, p1, p2) are determined, the pinhole camera model can be uniquely determined. This process is called "camera calibration". The first four are called internal parameters, and the last five are called distortion parameters. The distortion parameters are to supplement the internal parameters. So once the camera structure is fixed, including the lens structure and the focus distance, we can use these 9 parameters to approximate the camera.[10]

For a camera with better quality, the tangential distortion is small and thus p1,p2 can be ignored, and the radial distortion coefficient k3 can also be ignored, and only the two parameters k1 and k2 are calculated. Zhang Zhengyou's method[11] defaults to p1 and p2 to 0 in the calibration.

- Zhang's calibration model

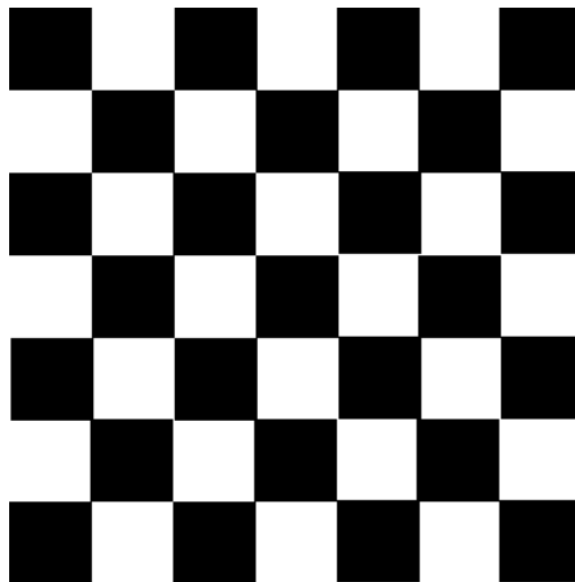
  Zhang Zhengyou calibration only considers radial distortion, not tangential distortion, which means only **k1** and **k2** are included. The model can be summed as follows:

$$\begin{bmatrix} (u-u_0)(x^2+y^2) & (u-u_0)(x^2+y^2)^2 \\ (v-v_0)(x^2+y^2) & (v-v_0)(x^2+y^2) \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} \bar{u}-u \\ \bar{v}-v \end{bmatrix}$$

  where (u,v) and (u¯,v¯) represent the ideal and real pixel coordinate respectively while (x,y) and (x¯,y¯) represent the ideal and real image coordinate respectively.

- Zhang's calibration procedures

  1. Print a checkerboard and paste it on a flat surface as a calibration object.
  2. By adjusting the orientation of the calibration object or the camera, take photos of the calibration object in different directions.
  3. Extract the checkerboard corner points from the photo.
  4. Estimate the five internal parameters and six external parameters under the condition of ideal no distortion.
  5. The least square method is used to estimate the distortion coefficient under the actual radial distortion.
  6. Maximum likelihood method, optimize estimation, improve estimation accuracy.



- Why checkerboard

  The first major role of the calibration board is to determine the correspondence

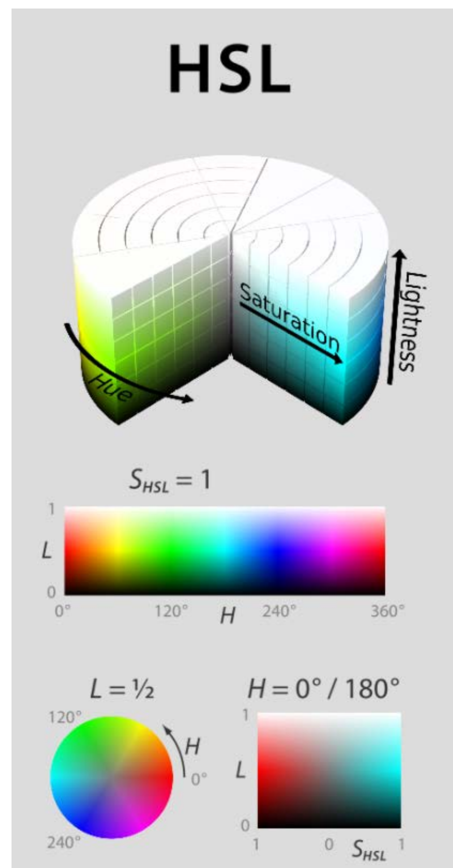
between the object point and the image point. The principle used here is mainly "perspective invariance." For example, if you look at a person up close and look at him far away, although the size of his nose has changed, the perspective of yourself has also changed, but the topology must be the same, you can't think of his nose as a mouth.[10]

In the calibration board, the topological structure is printed, and the checkerboard grid and dot grid are widely used. The reason why these two kinds of grids have become mainstream is not only because their topological structure is clear and uniform, but more importantly, the algorithm for detecting their topological structure is simpler and more effective. The checkerboard detects the corner points, as long as the gradient is calculated in the horizontal and vertical directions of the captured checkerboard image.

2.4 HSL color space

In Turtlebot3 burger, the parameter tuning for lane detection use HSL color space.

[3]HSL (hue, saturation, lightness) is an alternative representation of the RGB model designed to more closely match the human vision perception.



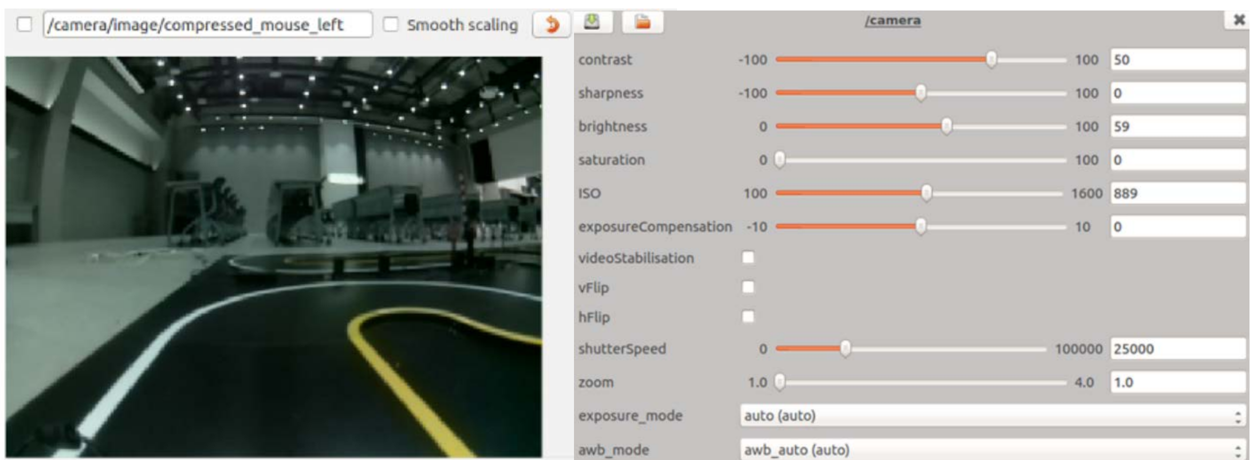
3 https://fr.m.wikibooks.org/wiki/Fichier:Hsl-hsv_models_b.svg

3.Project realization

3.1 Camera calibration

3.1.1 Basic parameters

In this step, we adjust the basic parameters of the camera to make it capture clear images. For example, the contrast, the sharpness, the saturation, the ISO and so on.



And save the parameters in the ‘camera.yaml’ file:

```
*camera.yaml (~/catkin_
Open  Sa
camera:
  ISO: 889
  awb_mode: tungsten
  brightness: 59
  contrast: 50
  exposureCompensation: 0
  exposure_mode: auto
  hFlip: false
  saturation: 0
  sharpness: 0
  shutterSpeed: 25000
  vFlip: false
  videoStabilisation: false
  zoom: 1.0
:8  Ln 8, Col 15  IN
```

so that the next time the camera is triggered, it will use the tuned parameters from the file.

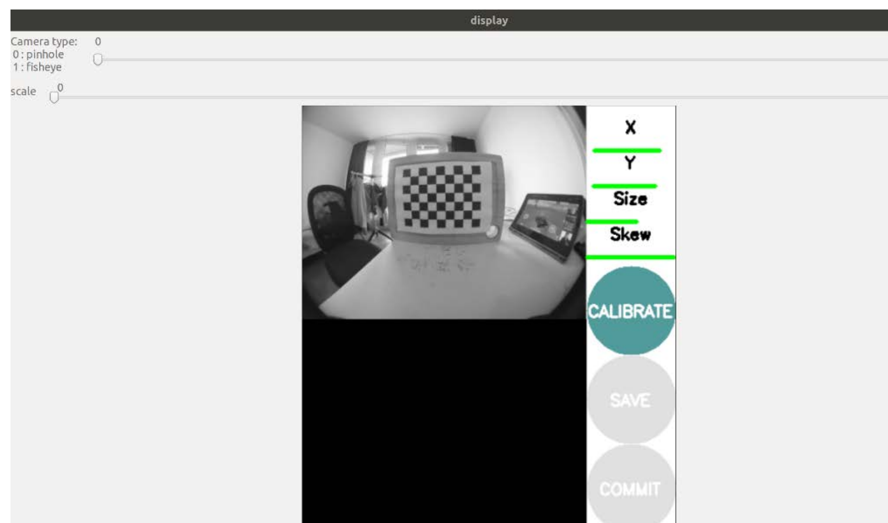
3.1.2 Intrinsic calibration

In this step, I printed a checkerboard on A4 size paper and stick it on a big board. Then I launch the intrinsic camera calibration and click the ‘calibration’ button. Next, I have to adjust the board’s position and angle multiple times until the 3 bars in the program

interface turn green, which means the program have received enough information to calibrate the intrinsic parameters.

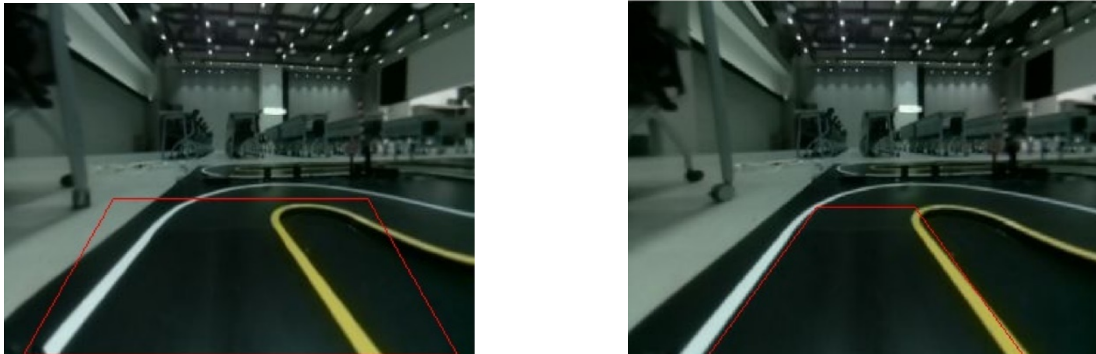
```
**** Calibrating ****
D = [-0.10297010972547899, 0.005936629995302991, 0.0009566214037599999, 0.0011848101876583712, 0.0]
K = [91.00851122498345, 0.0, 156.92547222211482, 0.0, 92.7385415742627, 101.88048058853697, 0.0, 0.0, 1.0]
R = [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P = [67.66864776611328, 0.0, 159.8876020847183, 0.0, 0.0, 67.56092834472656, 88.70237419403202, 0.0, 0.0, 0.0, 1.0, 0.0]
None
# oST version 5.0 parameters


[image]

width
320

height
240

[narrow_stereo]

camera matrix
91.008511 0.000000 156.925472
0.000000 92.738542 101.880481
0.000000 0.000000 1.000000

distortion
-0.102970 0.005937 0.000957 0.001185 0.000000

rectification
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000

projection
67.668648 0.000000 159.887602 0.000000
0.000000 67.560928 88.702374 0.000000
```

```
/home/tonyyan/catkin_ws/src/turtlebot3_autorace_2020/turtlebot3_autorace_traffic_light/turtlebot3_autorace_traffic_lig...
File  Edit  View  Search  Terminal  Help
*** Added sample 88, p_x = 0.240, p_y = 0.606, p_size = 0.265, skew = 0.205
*** Added sample 89, p_x = 0.310, p_y = 0.545, p_size = 0.254, skew = 0.043
*** Added sample 90, p_x = 0.160, p_y = 0.368, p_size = 0.274, skew = 0.462
*** Added sample 91, p_x = 0.206, p_y = 0.370, p_size = 0.286, skew = 0.167
*** Added sample 92, p_x = 0.086, p_y = 0.544, p_size = 0.338, skew = 0.068
*** Added sample 93, p_x = 0.151, p_y = 0.597, p_size = 0.295, skew = 0.011
*** Added sample 94, p_x = 0.665, p_y = 0.267, p_size = 0.247, skew = 0.319
*** Added sample 95, p_x = 0.796, p_y = 0.311, p_size = 0.275, skew = 0.289
*** Added sample 96, p_x = 0.821, p_y = 0.358, p_size = 0.281, skew = 0.109
*** Added sample 97, p_x = 0.759, p_y = 0.263, p_size = 0.242, skew = 0.135
*** Added sample 98, p_x = 0.694, p_y = 0.165, p_size = 0.249, skew = 0.048
*** Added sample 99, p_x = 0.577, p_y = 0.203, p_size = 0.262, skew = 0.247
*** Added sample 100, p_x = 0.502, p_y = 0.436, p_size = 0.263, skew = 0.989
*** Added sample 101, p_x = 0.218, p_y = 0.708, p_size = 0.338, skew = 0.200
*** Added sample 102, p_x = 0.319, p_y = 0.252, p_size = 0.254, skew = 0.208
*** Added sample 103, p_x = 0.429, p_y = 0.414, p_size = 0.295, skew = 0.723
*** Added sample 104, p_x = 0.328, p_y = 0.425, p_size = 0.423, skew = 0.534
*** Added sample 105, p_x = 0.325, p_y = 0.518, p_size = 0.240, skew = 0.265
*** Added sample 106, p_x = 0.276, p_y = 0.636, p_size = 0.318, skew = 0.027
*** Added sample 107, p_x = 0.188, p_y = 0.573, p_size = 0.344, skew = 0.282
*** Added sample 108, p_x = 0.423, p_y = 0.655, p_size = 0.256, skew = 0.635
*** Added sample 109, p_x = 0.410, p_y = 0.652, p_size = 0.252, skew = 0.391
*** Added sample 110, p_x = 0.397, p_y = 0.647, p_size = 0.249, skew = 0.203
*** Added sample 111, p_x = 0.548, p_y = 0.280, p_size = 0.303, skew = 0.113
*** Added sample 112, p_x = 0.603, p_y = 0.116, p_size = 0.333, skew = 0.416
*** Added sample 113, p_x = 0.557, p_y = 0.076, p_size = 0.377, skew = 0.301
```



3.1.3 Extrinsic calibration

The extrinsic camera calibration is quite straightforward. In order to detect the lanes as far as possible, we should adjust the projection view of the camera by adjusting the coordinates of the projection rectangle:

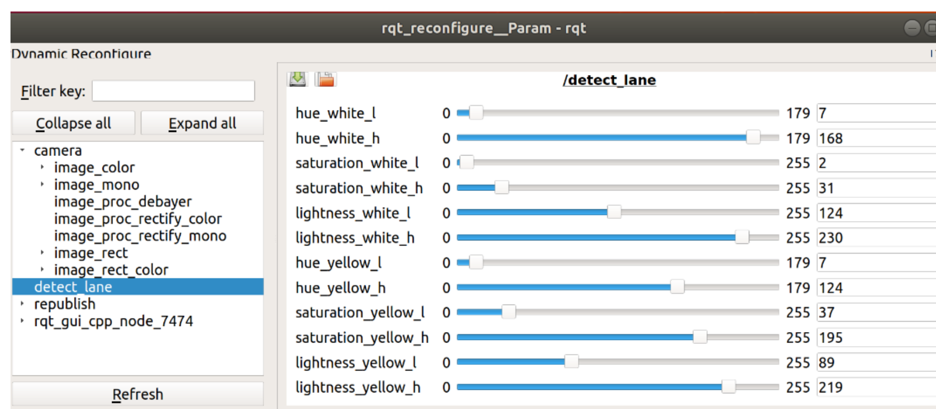


I also save the adjusted parameters into the local ‘projection.yaml’ file.

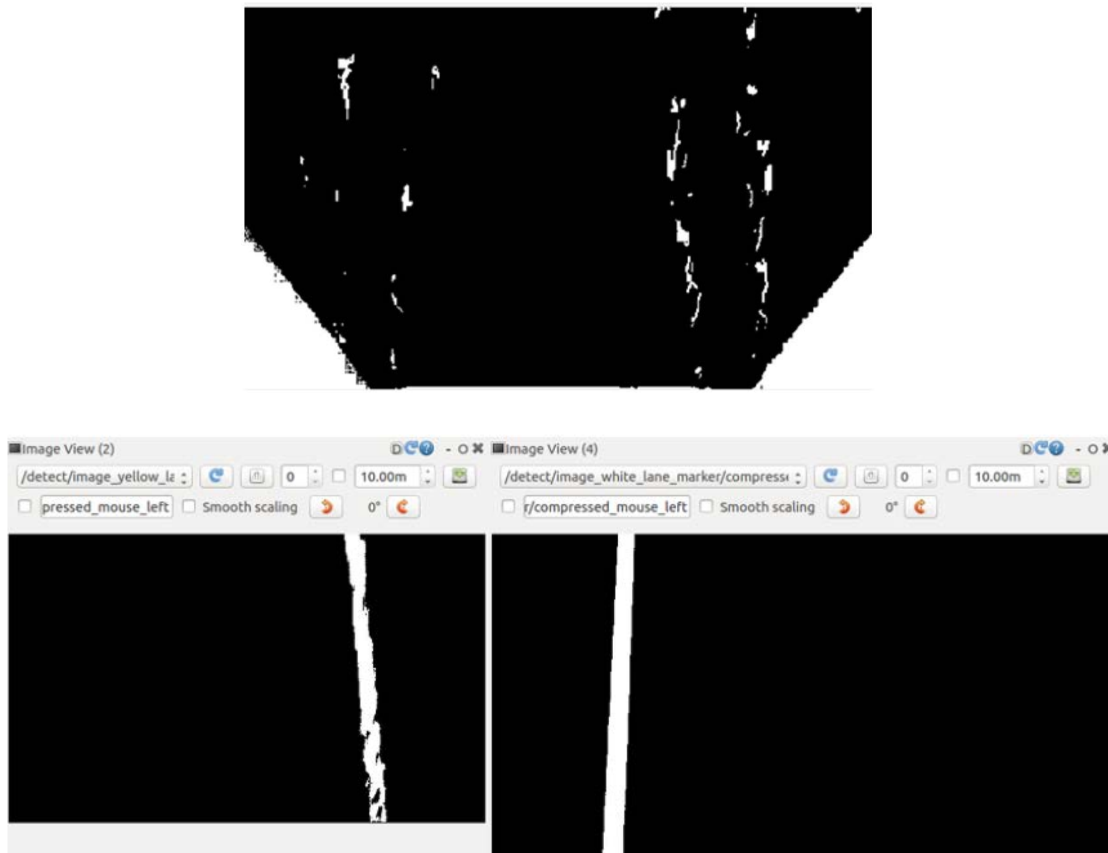
```yaml
camera:
  extrinsic_camera_calibration:
    top_x: 24
    top_y: 54
    bottom_x: 105
    bottom_y: 118
```

3.2 HSL parameter tuning for lane detection

By default, the Turtlebot3 burger detects lane by identifying the left yellow lane and the right white lane on a gray or black ground. In order to the yellow and white color, we have to tune the HSL parameters for the 2 colors:



After several adjustments, the blurry lanes become clearly identified:



Finally, I save the adjusted parameters into the local ‘lane.yaml’ file:

```
lane.yaml
home > tonyyan > catkin_ws > src > turtlebot3_autorace_2020
You, a few seconds ago | 2 authors (You and others)
1  detect:
2    lane:
3      white:
4        hue_l: 7
5        hue_h: 168
6        saturation_l: 2
7        saturation_h: 31
8        lightness_l: 124
9        lightness_h: 230
10     yellow:
11       hue_l: 7
12       hue_h: 124
13       saturation_l: 37
14       saturation_h: 195
15       lightness_l: 89
16       lightness_h: 219        You, a few
17
```

so that when the lane detection module is launched in action model, the lanes will be detected using the values above.

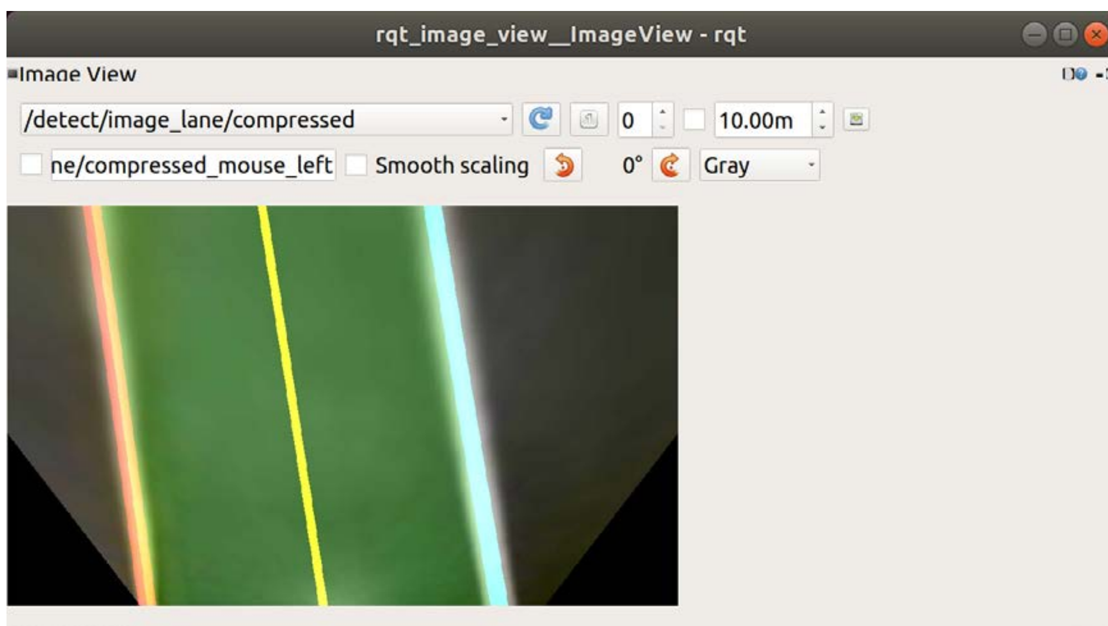
3.3 Autonomous driving with lane detection

After having configured the parameters correctly above, follow several steps below to run the Turtlebot3 burger with lane detection:

- Launch roscore on Remote PC.
- Trigger the camera on SBC.
- Run an intrinsic camera calibration launch file on Remote PC.
- Run an extrinsic camera calibration launch file on Remote PC.
- Run a lane detection launch file on Remote PC.
- Run a lane control launch file on Remote PC.
- Bring up the Turtlebot3 burger.

Then the lanes should be detected and the Turtlebot3 burger should start to run following detected lanes.

The detected lanes can be shown by using the 'rqt' command:



3.4 Difficulties encountered

Although the intrinsic camera calibration should have been the most time-consuming task of the entire project, I encountered a problem that is trickier. After following the instructions above, the Turtlebot3 burger didn't move after I typed the last command. The following steps describes the means I took to solve the problem:

- I first checked the teleoperation function[4] using remote PC keyboard. But it didn't work. No response from the 2 wheels. Also, the topic monitor could not detect the

[4] https://emanual.robotis.com/docs/en/platform/turtlebot3/basic_operation/#teleoperation

2 wheels:

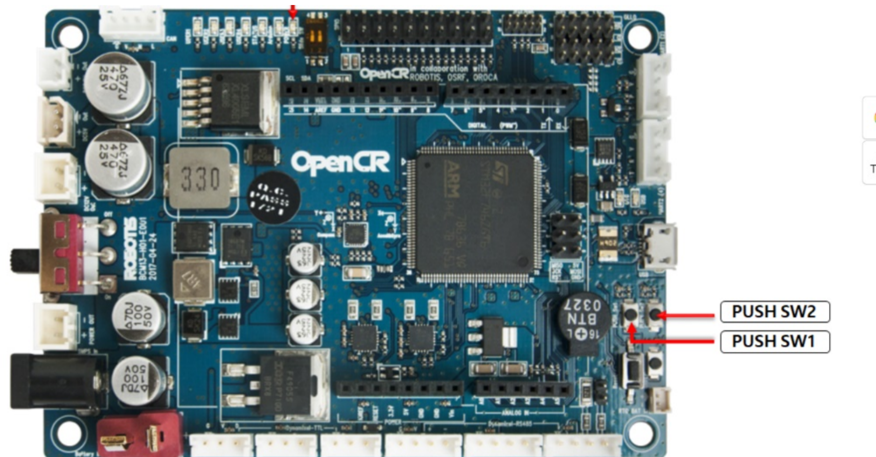
Default - rqt

File Plugins Running Perspectives Help

Topic Monitor

Topic	Type	Bandwidth	Hz	Value
seq	uint32			5583
▸ stamp	time			
▾ status	diagnostic_msgs/DiagnosticStatus[]			
▾ [0]	diagnostic_msgs/DiagnosticStatus			
hardware_id	string			'MPU9250'
level	byte			0
message	string			'Good Condition'
name	string			'IMU Sensor'
values	diagnostic_msgs/KeyValue[]			[]
▾ [1]	diagnostic_msgs/DiagnosticStatus			
hardware_id	string			''
level	byte			0
message	string			''
name	string			''
values	diagnostic_msgs/KeyValue[]			[]
▾ [2]	diagnostic_msgs/DiagnosticStatus			
hardware_id	string			'HLS-LFCD-LDS'
level	byte			0
message	string			'Good Condition'
name	string			'Lidar Sensor'
values	diagnostic_msgs/KeyValue[]			[]
▾ [3]	diagnostic_msgs/DiagnosticStatus			
hardware_id	string			''
level	byte			0
message	string			''
name	string			''
values	diagnostic_msgs/KeyValue[]			[]
▾ [4]	diagnostic_msgs/DiagnosticStatus			
hardware_id	string			''
level	byte			0
message	string			''

➢ Then I checked the wheels' moving function using 2 push buttons on the OpenCR board.[5] Unfortunately, after following the instructions, the wheels didn't move at all.



1. After assembling TurtleBot3, connect the power to OpenCR and turn on the power switch of OpenCR. The red Power LED will be turned on.
2. Place the robot on the flat ground in a wide open area. For the test, safety radius of 1 meter (40 inches) is recommended.
3. Press and hold PUSH SW 1 for a few seconds to command the robot to move 30 centimeters (about 12 inches) forward.
4. Press and hold PUSH SW 2 for a few seconds to command the robot to rotate 180 degrees in place.

➢ Having discussed with my supervising teacher, I reinstalled the firmware of OpenCR. But that doesn't solve the problem.

➢ By chance, I found a latest official video[6] that gives me hint on how to test the wheels whom are officially called 'DYNAMIXEL'. But the wheels still didn't

---

[5] https://emanual.robotis.com/docs/en/platform/turtlebot3/opencr_setup/#opencr-setup

[6] https://www.youtube.com/watch?v=0_M0Da9SHDw&list=WL&index=1&t=131s&ab_channel=ROBOTISOpenSourceTeam

move by uploading the source file ‘position_mode’.

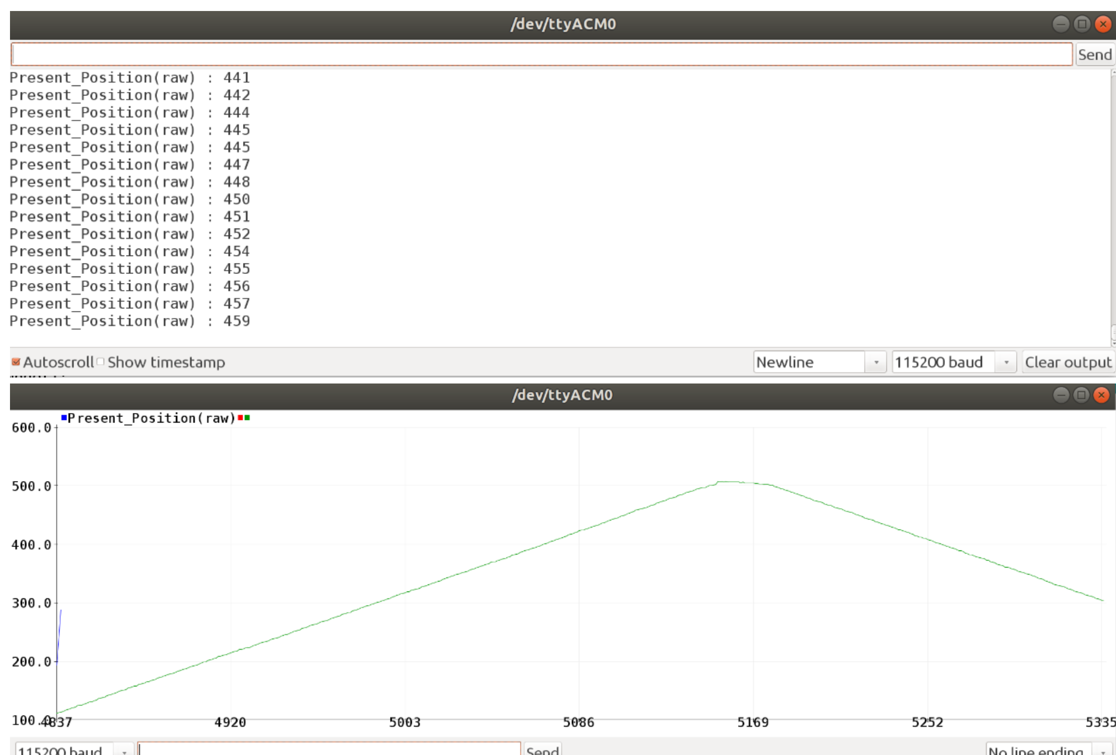
- I had to test if the 2 DYNAMIXELs can be found in OpenCR. Otherwise, this might be a hardware problem. Luckily, by uploading the ‘find_dynamixel’ file from OpenCR package in Arduino IDE, I successfully detected the 2 DYNAMIXELs:

You can see very clearly that they are detected under the baud rate 1000000.

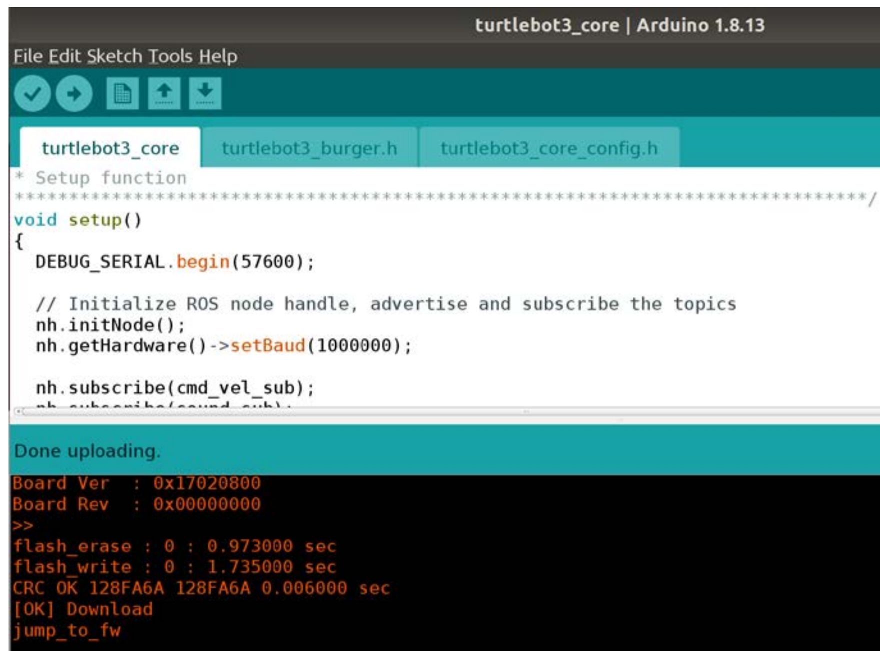
- So, I changed the baud rate from the default 57600 bps to 1000000 bps in the source file ‘position_mode’.

```
// Set Port baudrate to 57600bps. This has to match with DYNAMIXEL baudrate.
dxl.begin(1000000);
```

I uploaded the modified source file ‘position_mode’ to OpenCR. One wheel can move:



- In order to apply the 1000000 baud rate to both wheels, I changed the default baud rate in source file ‘turtlebot3_core’ and uploaded it to the OpenCR:



- Now the robot's 2 wheels are well configured. It can run with 2 wheels using the configured lane detection module.

4.Conclusion

4.1.The limitation of the pinhole camera model

Although the pinhole imaging model fully considers the influence of the camera's internal parameters on imaging, it does not consider another important part of the imaging system, the lens. Commonly used lenses are ordinary lenses, wide-angle lenses, fish-eye lenses, etc. In the field of driverless and visual slam, fish-eye lenses and wide-angle lenses are used a lot, mainly because the angle of view is large, and more information can be observed.[7]

Any lens has different degrees of distortion, and different types of lenses use different distortion models. According to the physical characteristics of lens manufacturing and imaging, ordinary lenses mainly consider radial distortion and tangential distortion, and the distortion models can be approximated by polynomials. For large wide-angle and fisheye lenses, the physical model of ordinary lenses is no longer applicable.

4.2.What I learned from the project

By realizing this project, the most things I learned is about the camera, such as the camera's pinhole model, its intrinsic and extrinsic parameters, why we should calibrate a camera, how to calibrate it and so on. I practiced the Zhang's calibration method with my own hands and had a intuitive experience.
Moreover, I also learned a lot about the basic concepts of ROS system, for instance, its workspace structure, its communication methods, its modules and launch files and so on.
Finally, by debugging the 'DYNAMIXEL' wheels, I had a contact with the lower-level hardware configurations of OpenCR and Arduino IDE.
In conclusion, through this project, I learned a lot of the underlying knowledge related to machine vision and autonomous driving. Such as camera, ROS system, and low-level hardware debugging. Not only did I increase my knowledge in related fields, it also enhanced my ability to analyze and solve problems on my own.

4.3.Appreciations

2020 is a difficult year because of the COVID-19 pandemic. During the Autumn semester, we can hardly go to school due to the lockdown. Here I would like especially thank Mr. YAN, my supervising teacher for the project, who spared time to communicate with me to help solve the problems. Also, I would like thank Mr. DENG, who helped me a lot in the final testing step.

5.References

[1] https://www.bilibili.com/video/BV1mJ411R7Ni
[2] https://www.guyuehome.com/7832
[3] https://www.cnblogs.com/wangguchangqing/p/8126333.html
[4] https://www.zhihu.com/question/45982671
[5] https://blog.csdn.net/lyq_12/article/details/83625339
[6] https://zhuanlan.zhihu.com/p/125006810
[7] https://zhuanlan.zhihu.com/p/87334006
[8] https://blog.csdn.net/lql0716/article/details/71973318
[9] https://blog.csdn.net/a083614/article/details/78579163
[10] https://zhuanlan.zhihu.com/p/30813733
[11] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr98-71.pdf