Lab Guide

Lane Keeping

Background

The objective of the lane keeping lab guide is to teach students how to extract lane information from camera feeds and implement lane keeping control using a pure pursuit controller. Prior to starting this lab guide, please review the following concept reviews:

- Concept Review – Camera Model
- Concept Review – Image Filters
- Concept Review – Geometric Lateral Control

Considerations for this lab guide:

Having the camera intrinsics of the virtual Intel RealSense camera is a pre-requisite for this lab guide. For details, refer to the Image Interpretation Skills Activity lab guide. In addition, you can refer to Concept Review – Camera Model for camera extrinsics and Concept Review – Geometric Lateral Control for the formulation of the pure pursuit controller.

This lab will guide you through the 2 pipelines as shown below:

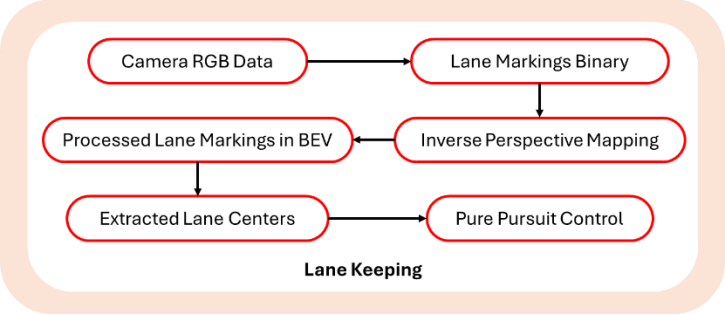

Figure 1: Lane Keeping

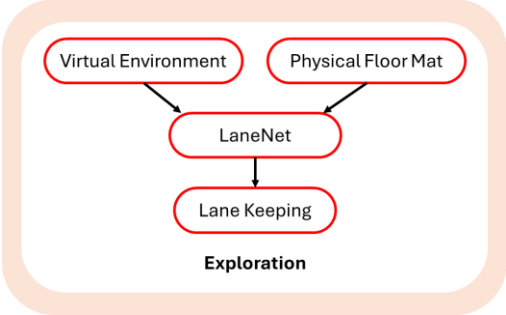

Figure 2: Exploration

- **Lane Keeping** is designed to guide students through the development of a basic lane keeping algorithm and observe its limitations in a simple virtual environment.
- **Exploration** allows students to experiment with a neural network (LaneNet) for lane extraction and observe performance in virtual and physical environments.

Skills Activity - Setup

The setup for the lab will require **a physical QCar2** to run the lane keeping algorithm and **a local machine** to host the simulated environment in Quanser Interactive Lab. This lab currently does not support QCar1 or virtual-only configuration. It is recommended that both the local machine and the QCar2 are connected to the same network via **Ethernet** to minimize latency. It is recommended that the QCar2 is remotely accessed via **Window's Remote Desktop Connection** to better interface with the visual outputs.

lane_keeping.py is the main file that needs to be edited and executed on the QCar2 for interacting with this skills activity. Experiments can be configured by modifying the following parameters, located near the top of **lane_keeping.py**

```
# ===== Timing Parameters
# - controllerUpdateRate: control update rate in Hz. 10 is sufficient for
#   lane keeping application.
controllerUpdateRate = 10
# ===== Speed Controller Parameters
# - vDesire: desired velocity in m/s
# - Kp: proportional gain for speed controller
# - Ki: integral gain for speed controller
# - Kff: feed forward gain for speed controller
vDesire=10
Kp=0.01
Ki=0.005
Kff=1/60
# ===== Bird's-Eye View (BEV) Parameters
# - bevShape: width and height of the BEV image
# - bevWorldDims: [min x, max x, min y, max y] of the world represented by the
BEV image
bevShape = [800,800]
bevWorldDims = [0,20,-10,10]
# ===== Pure Pursuit Controller Parameters
# - Kdd: gain for calculating look-ahead distance
# - ldMin: minimum look-ahead distance
# - ldMax: maximum look-ahead distance
# - maxSteer: maximum steering angle in either direction
Kdd = 0
ldMin = 10
ldMax = 20
maxSteer = 0
# ===== Keyboard Driver Parameters
# - maxKeyThrottle: maximum PWN command from keyboard driver
# - maxKeySteer: maximum steering angle from keyboard driver
maxKeyThrottle = 0.2
maxKeySteer = 0.1
```

Prior to running the main script on QCar2, on your local machine, open Quanser Interactive Labs, navigate to the Self-Driving Car Studio pane and select Open Road. Then, run **qlab_setup.py** to spawn a QCar2 agent and NPC vehicles as shown in Figure 3.

Figure 3: Virtual Environment Setup

Review the User Manual – Connectivity for establishing a network connection and transferring files to the QCar2. After transferring **lane_keeping.py** to the QCar2, navigate to its directory and execute the script:

```
python lane_keeping.py
```

Then, you will be asked to verify the IP address of your local machine or modify it. Once the physical QCar2 is connected to the virtual environment, a confirmation message will be displayed as shown in Figure 4.

```
nvidia@qcar2-master:~/Documents/06-lane_keeping$ python lane_keeping.py
This QCar is currently configured for remote ip 172.16.4.61, would you like
to change? (y/n)y
enter new remote ip:172.16.4.61
QCar configured successfully.
```

Figure 4: Running the Main Script.

You can drive the virtual QCar2 with '**WASD**' keys and terminate the script with '**ESC**' key. When this skills activity is fully implemented, you can enable lane keeping with the **Space Bar** and select lane with the **left** and **right arrow** keys. When **lane_keeping.py** is running, a Pygame window labeled 'Keyboard Window' will open, as shown in Figure 5. Make sure to select this window in order to send keyboard inputs.

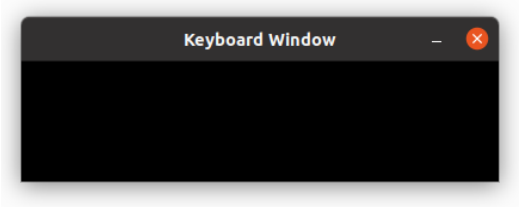

Figure 5: Pygame Window for Receiving Keyboard Inputs.

Skills Activity – Lane Keeping

In this section, students will familiarize themselves with some key concepts in self-driving such as lane extraction, inverse perspective mapping (IPM) and pure pursuit control. In addition, students will interact with industry standard image processing tools such as OpenCV and develop intuition on the strength of these tools. By the end of this skills activity, students will have a completed lane keeping algorithm and use it to navigate through traffic to reach the finish line by commanding the virtual QCar2 to switch between lanes.

Step 1 – Extract Lane Marking Information from RGB Camera Feed

The goal of this section is to utilize the line detection algorithm developed in the image interpretation skills activity to extract lane marking from the camera feed. Complete **SECTION A** of the in **lane_keeping.py** to generate a binary mask, ***laneMarking***, which should have the same width and height as the camera feed.

```
# ==============  SECTION A - Lane Marking  ====================
laneMarking = np.zeros((480,640),dtype=np.uint8)
# ==============       END OF SECTION A     ====================
```

Results:

Two cv2 windows labeled 'camera' and 'lane marking' will open. Adjust the line detection parameters until the lane markings are sufficiently highlighted as shown in Figure 6.

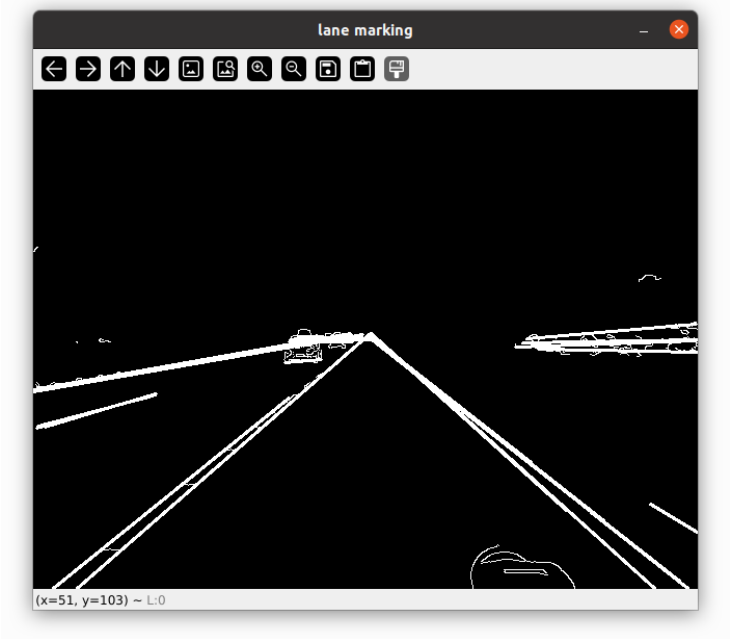

Figure 6. Result of Lane Marking Extraction using Line Detection.

Step 2 – Implement Inverse Perspective Mapping (IPM)

In this section, IPM will be implemented to convert the camera feeds and extracted lane marking to bird's-eye view (BEV). BEV is important in self-driving algorithms as most state-of-the-art methods fuse multi-sensory data and compute driving decisions in the BEV space.

Navigate to the **InversePerspectiveMapping()** class inside **qcar_functions.py**. This can be done by hover over **create_bird_eye_view()** method, right click and select 'Go to definition' or find **qcar_functions.py** in hal/content directory.

Under the **__init__()** method, complete **SECTION B.1**. you can copy over the camera intrinsics computed from image interpretation skills activity or recalibrate the camera to produce the camera intrinsics again.

```
        # ==============  SECTION B.1 - Camera Intrinsics  ====================
        self.camera_intrinsics = np.zeros((4,4))
        # ==============          END OF SECTION B.1        ====================
```

Then, navigate to the **get_extrinsics()** method, and complete **SECTION B.2**. In this subsection, you will need to compute the camera extrinsics, multiplied by which a vector can be converted from vehicle frame to camera frame.

```
        # ==============  SECTION B.2 -  Camera Extrinsics  ====================
        self.camera_extrinsics = np.zeros((4,4))
        # ==============          END OF SECTION B.2         ====================
```

The camera intrinsics and extrinsics can then be used together to convert a point in vehicle frame to a pixel in image frame. Navigate to **v2img()** method and complete **SECTION B.3**. When completed, **v2img()** should convert an input nx3 vector, *XYZ*, to an output nx2 vector, *uv*. *XYZ* represents n points (3 dimensional) in vehicle frame whereas *uv* is the corresponding n pixel locations (2 dimensional) in the image frame.

```
        # ==============  SECTION B.3 -  Vehicle to Image  ====================
        uv = np.zeros((XYZ.shape[0],2),dtype=np.uint8)
        return uv
        # ==============          END OF SECTION B.3        ====================
```

The next step to creating a BEV is to compute the homography matrix which maps the pixels in the camera feeds to pixels in BEV image. This can be done in the following steps:

i. Choose four arbitrary points in vehicle frame (four corners of BEV).
ii. Convert them to image coordinates in RGB image frame using **v2img()**.
iii. Convert the same four points to BEV coordinates based on desired BEV world dimensions.
iv. Compute the homography matrix using the two set of corners utilizing OpenCV's **getPerspectiveTransform()**.

Implement these steps in **get_homography()** method and complete **SECTION B.4**. The resulting homography matrix should be stored in the variable *self.M*.

```
        # ==============  SECTION B.4 -  Image to Vehicle  ====================
        self.M = np.eye(3)
        # ==============          END OF SECTION B.4        ====================
```

Finally, complete **SECTION B.5**. to create the BEV using the homography matrix via OpenCV **warpPerspective()**.

```
        # ==============  SECTION B.5 -  Create BEV  ====================
        dst = np.zeros(self.bevShape,dtype=np.uint8)
        return dst
        # ==============      END OF SECTION B.5     ====================
```

Results:

Uncomment the 'camera BEV' and 'lane marking BEV' windows in the 'Visualization' section located near the bottom of **lane_keeping.py**, and the cv2 windows will open. You can also comment out unnecessary visualization windows which may improve the performance of the remote connection.

To verify the correct implementation of IPM, the lane markings should be parallel in the BEV as shown in Figure 7.

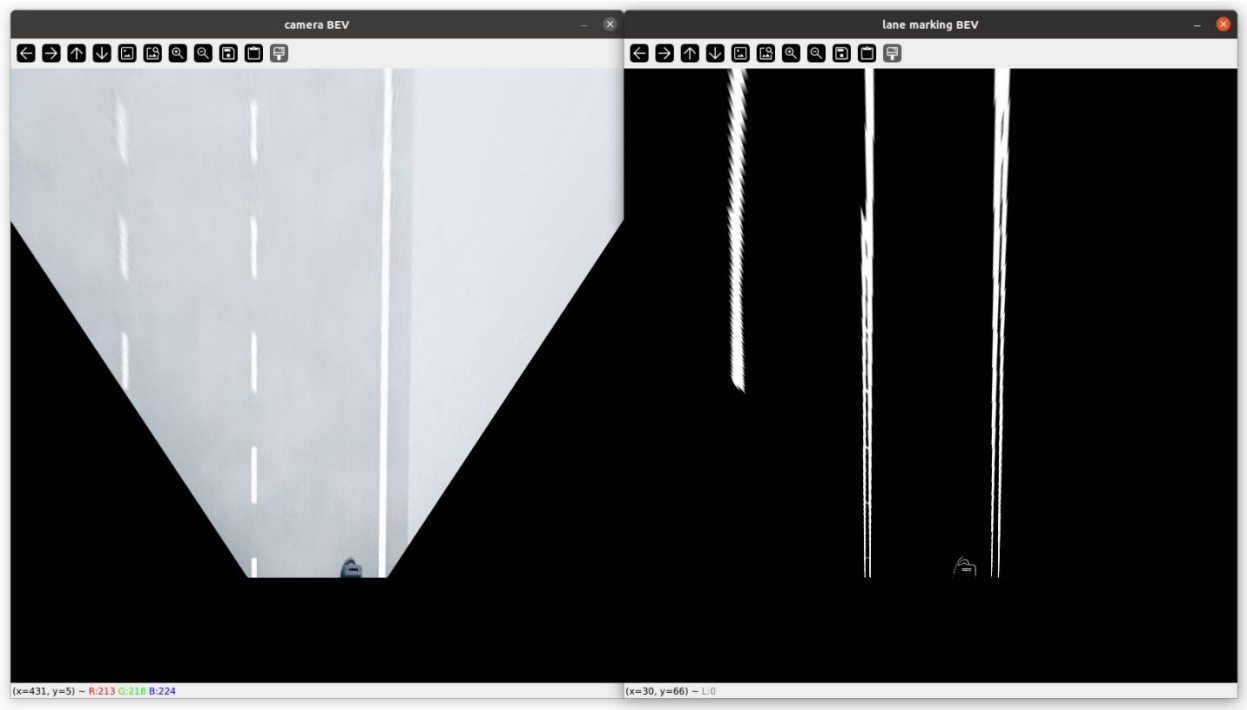

Figure 7. Example of camera BEV (left) and lane marking BEV (right).

Step 3 – Lane Marking BEV Cleanup

After transforming the lane marking to BEV, region that is further away (in vehicle frame) from the camera becomes blurry due to interpolation. The goal of this section is to clean up the Lane Marking BEV for further analysis.

Navigate to the **preprocess()** method in the **LaneKeeping()** class and complete **SECTION C.** to convert the lane marking BEV to binary (no grey pixels) and merge the two edges of the same lane marking.

```
        # ==============   SECTION C - Preprocess Lane Marking   ==============
        return np.zeros_like(lane_marking)
        # ==============              END OF SECTION C           ==============
```

Results:

Uncomment 'processed BEV' window and comment out appropriate lines in the 'Visualization' section located near the bottom of **lane_keeping.py**. Adjust the image processing parameters until the lane markings have no gray pixels and the edges of the same lane markings are merged as shown in Figure 8.

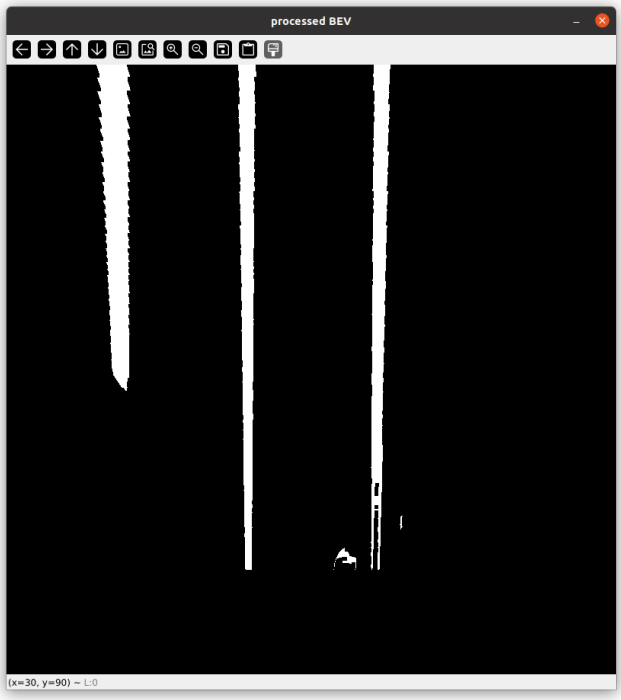

Figure 8. Processed BEV after Eliminating Gray Pixels and Merged Edges.

Step 4 – Lane Marking Isolation

In this section, individual lane markings will be isolated. Navigate to the **isolate_lane_markings()** method in **qcar_functions.py** and complete **SECTION D**. Consider using OpenCV for connected component analysis to extract individual lane marking. Note that the output list should contain a list of binary images, each of which contains only one lane marking. In addition, you should remove blobs that are too small.

```
        # ==============  SECTION D - Isolate Lane Marking ====================
        isolated_binary_blobs=[]
        return isolated_binary_blobs
        # ==============          END OF SECTION D        ====================
```

Results:

Uncomment the lines containing the for loop and 'blob' window and comment out appropriate lines in the 'Visualization' section located near the bottom of **lane_keeping.py**, and one cv2 window for each isolated lane marking will open. Adjust the isolation parameters until appropriate blobs are extracted as shown in Figure 9.

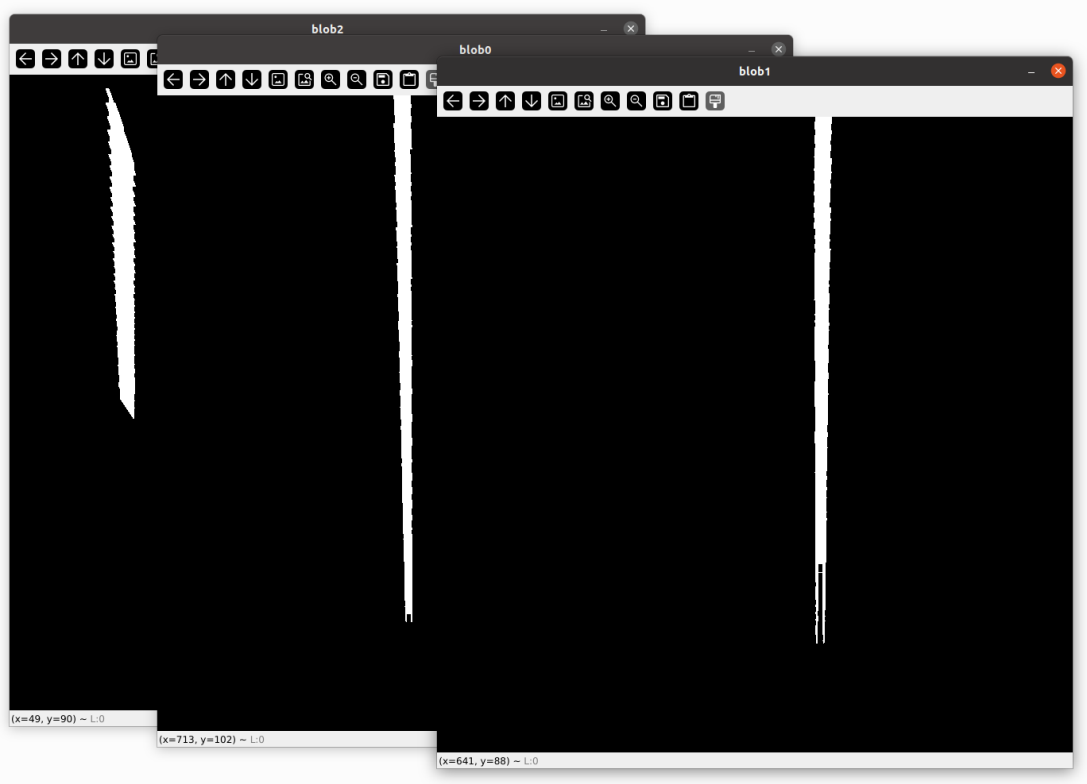

Figure 9. Three Isolated Lane Markings.

Step 5 – Find Lane Centers

With the individual lane markings isolated, the next step is to find the available lane centers as valid target points for the pure pursuit controller. Navigate to the **find_target()** method in **qcar_functions.py**, and complete **SECTION E.1** to derive the look-ahead distance (ld) based on measured velocity, *v*. Consider using the ld gain *self.Kdd*, as well as *self.ldMin* and *self.ldMax*.

```
        # ==========   SECTION E.1 - Calculate Look-ahead Distance ==============
        self.ld_pix = 1 #look-ahead distance in number of pixels
        # ==========              END OF SECTION E.1              ==============
```

Immediately after SECTION E.1, each isolated lane marking is used to instantiate one **LaneMarking** object which stores relevant information about the lane marking. Then, an intersection point is determined for each lane marking at the look-ahead distance away from the rear wheel. To achieve this, navigate to the **find_intersection()** method within the **LaneMarking()** class, and complete **SECTION E.2**. Note that the pixel coordinates of the rear wheel are stored in the variable *self.bev_rear_wheel_pos*.

```
        # ==============  SECTION E.2 -  Find Intersection  ====================
        self.intersection = None
        # ==============          END OF SECTION E.2        ====================
```

Return to the **find_target()** method. When all the **LaneMarking** objects are instantiated, they're stored in the variable ***self.isolated_lanes***, which is then sorted by the location of their intersection points (with index 0 corresponding to the rightmost lane marking).

Next, estimate the lane centers based on the location of each intersection. You will explore different ways to estimate the lane centers based on the number of the detected lane markings and their locations.

Case I – only one lane marking is detected. The lane center should be located a certain distance away from the intersection point, either to the left or right, with the connecting line perpendicular to the lane marking. To compute the estimated lane center, first determine which side it should lie on by calculating an offset vector, ***vec***. Complete **SECTION E.3**

```
        # ============ SECTION E.3 - Determine Side (Case I) ===============
        vec=np.array((75,0))
        # ============           END OF SECTION E.3          ===============
```

Then, compute the rotation matrix to orient the offset vector perpendicular to the lane marking. Navigate to the **find_rot_mat()** method and complete **SECTION E.4**.

```
        # =============  SECTION E.4 - Find Rotation Matrix ====================
        rot_mat = np.eye(2)
        return rot_mat
        # =============           END OF SECTION E.4        ====================
```

Finally, the target point (lane center) can be computed by adding the rotated offset vector to the intersection point.

Case II – more than one lane marking is detected and the distance between the neighbouring intersections is similar to the width of a lane (indicating adjacent lane markings). Complete **SECTION E.5** accordingly to estimate the lane center based on two neighboring intersection points.

```
        # ============  SECTION E.5 - Find Target (Case II)   ================
        target = np.array((0,0))
        # ============           END OF SECTION E.5          ================
```

Case III – the distance between the neighbouring intersections is significantly larger than the width of one lane (indicating that these lane markings contain two lanes.) Complete **SECTION E.6** to find two target points. Hint: Case III can be treated as two instances of Case I.

```
        # ============  SECTION E.6 - Find Target (Case III)  ================
        target1 = np.array((0,0))
        target2 = np.array((0,0))
        # ============           END OF SECTION E.6          ================
```

Results:

In the 'Visualization' section, uncomment **myLaneKeeping.show_debug()** and comment out appropriate lines, to open a cv2 window labeled 'debug'. In this window, the blue arc represents ld, while the red and green circles represent intersection points and estimated lane centers respectively, as shown in Figure 10. If SECTION E.1 is implemented properly, the blue arc will change dynamically with the speed of QCar2. Test the system by driving QCar2

to locations representing each case, and consider adjusting the pure pursuit parameters (located in the 'Tunable Experiment Configuration') to stabilize the blue arc for debugging.

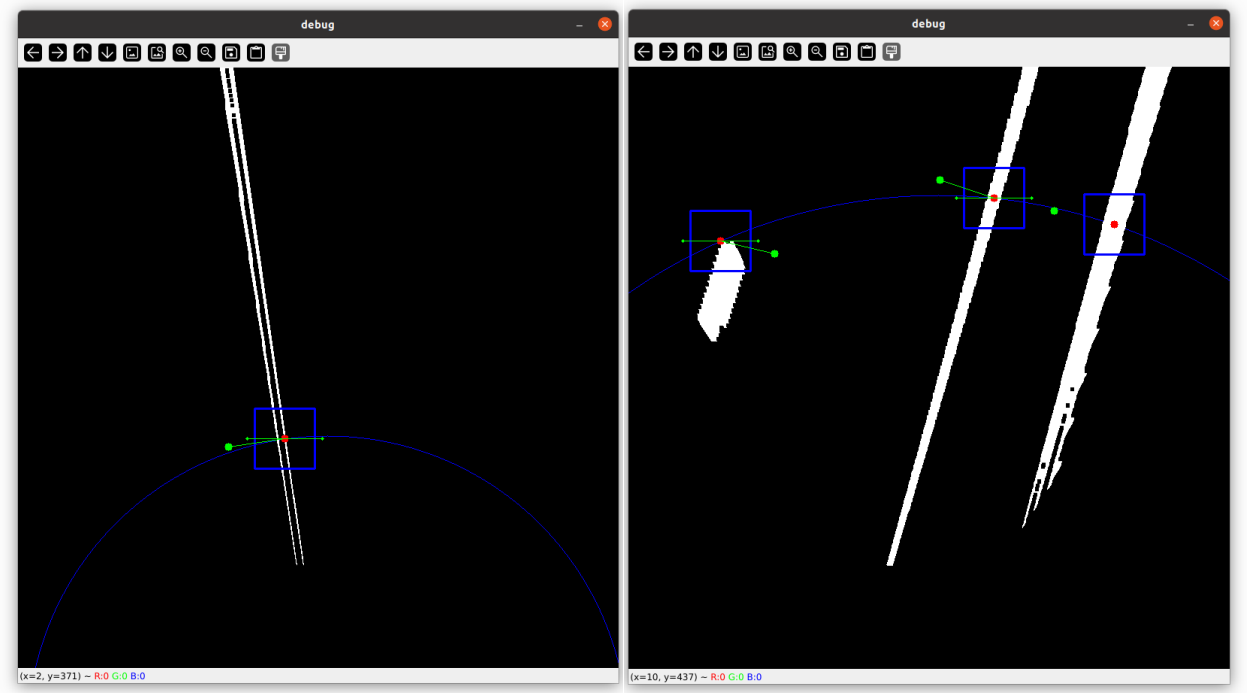

Figure 10. Estimated Lane Centers for Case I (left), and Case II & III (right).

Step 6 – Implement Pure Pursuit Controller

Now that the lane centers are detected, they can be used by a pure pursuit controller to facilitate lane keeping. In this section, you'll implement the pure pursuit controller. Navigate to the **PurePursuitControler()** class in **qcar_functions.py**, and complete **SECTION F** in the **target2steer()** method. This function should take a target point in BEV pixel coordinates and output the steering angle in radian. Consider using the variables *self.m_per_pix*, *self.bev_rear_wheel*, and *self.maxSteer* in your implementation.

```
        # ==============  SECTION F - Compute Steering Angle ====================
        steer = 0
        return steer
        # ==============          END OF SECTION F           ====================
```

Results:

After implementing the pure pursuit controller, press and hold the Space Bar to enable lane keeping and use the left and right arrow keys to select lanes. Complete the driving scenario by selecting different lane to avoid traffic. Uncomment 'camera BEV' window in the Visualization section to see the detected lane centers (green) and selected lane center (red) as shown in Figure 11. You can tune the BEV parameters and pure pursuit controller parameters in the 'Experiment Configuration' section to improve the lane keeping performance.

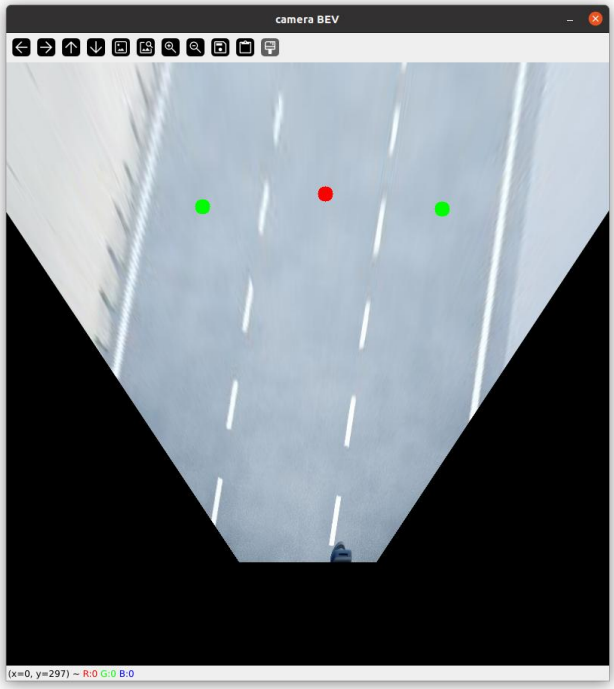

Figure 11. Annotated Camera BEV.

Considerations:

Under which scenarios do lane keeping fail? What are the potential sources of errors in lane detection in these cases? Can you suggest any methods to address these issues?

Skills Activity – Exploration

The objective of this section is to go beyond the conventional image processing techniques and the simple virtual environment. You will explore a popular lane detection neural network model (LaneNet) and observe its performance in both the simulated environment and the physical environment. This skills activity should help you develop intuition on the strength and limitations of LaneNet.

Step 1 – Explore LaneNet

To implement LaneNet for lane marking extraction, first, uncomment the three lines related to LaneNet in the 'Initial Setup' region.

```
        myLaneNet = LaneNet(rowUpperBound=240,
                            imageWidth=640,
                            imageHeight=480)
```

Then replace **SECTION A** with the following lines:

```
        ### using LaneNet to find lane markings
        rgbProcessed = myLaneNet.pre_process(img)
        myLaneNet.predict(rgbProcessed)
        laneMarking = myLaneNet.binaryPred
```

The first line pre-processed the input camera feed into a format acceptable for LaneNet. The second line make a prediction based on the pre-processed image. Finally, the lane markings prediction (a binary lane mask) is stored in the *laneMarking* variable. The **LaneNet()** class, part of Quanser's Python Intelligence Toolset (PIT), simplifies model integration and output post-processing. You can explore its additional functionalities by reviewing the class implementation.

Rest of the script remains unchanged. Ensure you run **qlab_setup.py** on your local machine and before executing **lane_keeping.py** on the QCar2.

Results:

If the LaneNet implemented correctly, the 'debug' and 'camera BEV' window should look similar to those in the previous activity, as shown in Figure 12. Complete the driving scenario and take note of the differences in performance compared to the previous activity.

Figure 12. Debug Window (left) and Annotated Camera BEV Window(right) with LaneNet.

Step 2 – Explore Physical Floor Mat

Modify lane_keeping.py to receive sensor data from and send commands to the physical QCar2. In the 'File Description and Import' region, modify the following line to import the QCar API that communicates with the hardware instead:

```
from pal.products.qcar import QCarRealSense,QCar
# from hal.content.qcar import QCarRealSense,QCar
```

Since the physical QCar2 is 1/10th the scale of the virtual QCar2, velocity-related variables need to be adjusted. Locate *vDesire* variable under 'Speed Controller Parameters' in the 'Tunable Experiment Configuration' region and reduce the to 1. Then, find the definition of *v*, near the top of the 'Main Loop' region and remove the x10 gain.

```
        v = qcar.motorTach  # motor tach output is in 1/10th scale
```

Place the QCar2 on the outer ring of the floor mat and run **lane_keeping.py** on the QCar2. Variables in the 'Tunable Experiment Configuration' can also be adjusted to improve performances. Take notes of the difference in lane keeping behavior between using the Open Road virtual environment and using the physical floor mat.

Results:

When the script is executed, the 'debug' and 'camera BEV' windows will open, as shown in Figure 13.

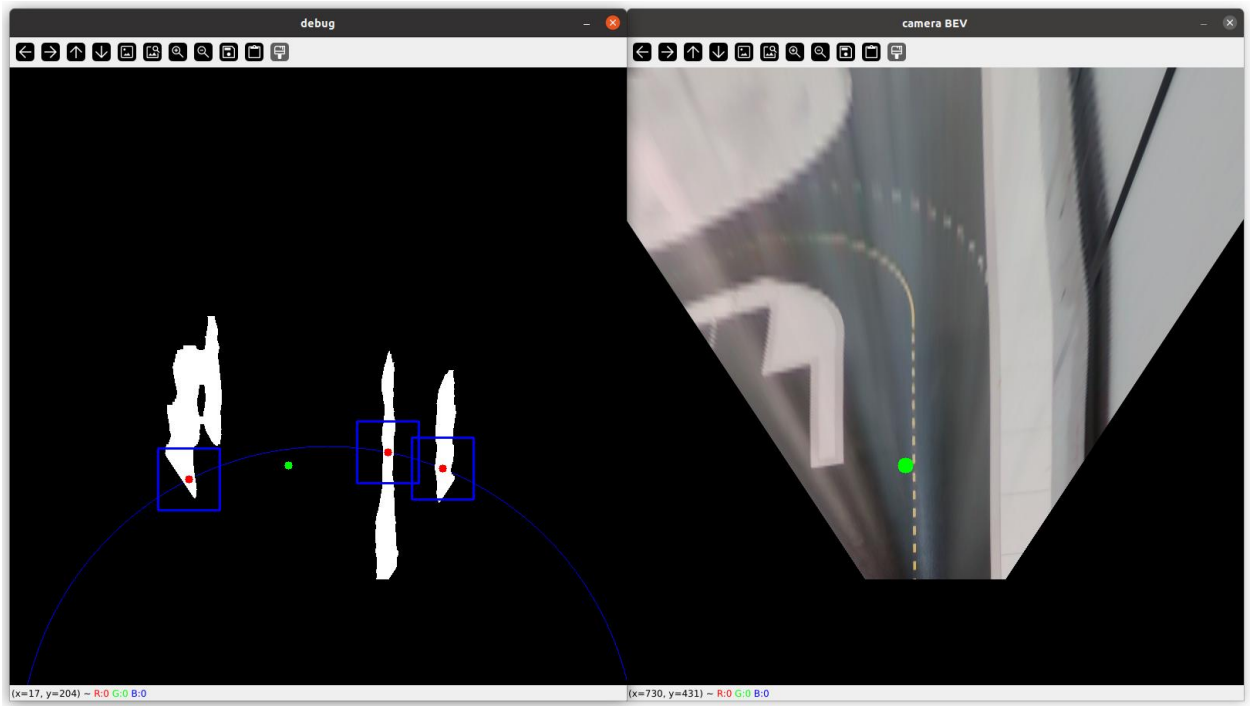

Figure 13. Debug window (left) and Camera BEV window (right) with LaneNet on Floor Mat.

Considerations:

What are the parameters you have changed to improve the lane keeping performance? What was the biggest challenge with the physical floor mat? Quanser's LaneNet model was trained on a dataset collected only in 'Open Road', what are the main differences between the two environments that might cause LaneNet to perform poorly on the physical floor mat?