Lateral Control: Lane Keeping

In the previous lab, you have implemented a lateral controller for vision-based lane keeping in simulation. This week you will implement a lateral controller on the BARC. When working with real hardware, there is additional uncertainty that you must factor into your control design. For example, the real camera will have noise and visual warping from the lens. Even with a good calibration, the image may not be perfectly converted from a fisheye to rectangular. You may also deal with model imperfection and disturbances to your input.

Task 1 Kinematic Bicycle Model - No Deliverables

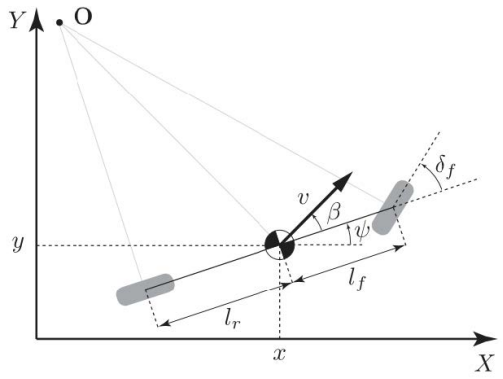


Figure 1: Simplified Kinematic Bicycle Model

Recall the simplified kinematic bicycle model shown in Figure 1:

$$\begin{aligned} \dot{x} &= v\cos(\psi+\beta) \\ \dot{y} &= v\sin(\psi+\beta) \\ \dot{\psi} &= \frac{v}{l_r} sin(\beta) \\ \delta_f &= \tan^{-1}\left(\frac{l_r+l_f}{l_r}\tan(\beta)\right) \end{aligned} \tag{1}$$

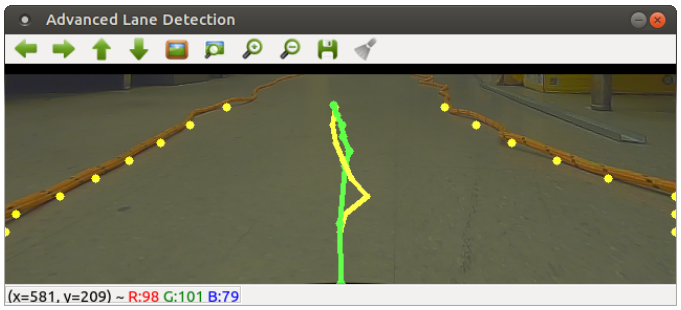
with states $z = \begin{bmatrix} x, & y, & \psi \end{bmatrix}^T$ and inputs $u = \begin{bmatrix} v, & \delta_f \end{bmatrix}^T$ where

$$\begin{aligned} x &= \text{inertial x coordinate, mass center location} \\ y &= \text{inertial y coordinate, mass center location} \\ v &= \text{longitudinal speed of the vehicle} \\ \psi &= \text{global heading angle} \\ \delta_f &= \text{steering angle of the front wheels} \end{aligned}$$

l_r = distance from the CoM to the rear axle
l_f = distance from the CoM to the front axle
β = sideslip angle

Task 2 Converting Image Pixels to Inertial Frame Coordinates

You will be using live camera images to determine the orientation of the track with respect to the camera mounted on the BARC. To do this, you first need to correlate image pixels to the corresponding real world position. We want to obtain positions in the inertial frame, which is reinitialized from the body frame at step k=0 for all time t.



1. Set the BARC on the floor and run `roslaunch labs camera_only.launch`
   This turns on the camera and displays the live feed images. Note that the original image has a coordinate frame that does not align with the inertial frame. The image starts with (0,0) on the top left corner but we want to change the (x_{pixel}, y_{pixel}) to $(x_{newpixel}, y_{newpixel})$ which is centered about the inertial origin (see Fig. 2).

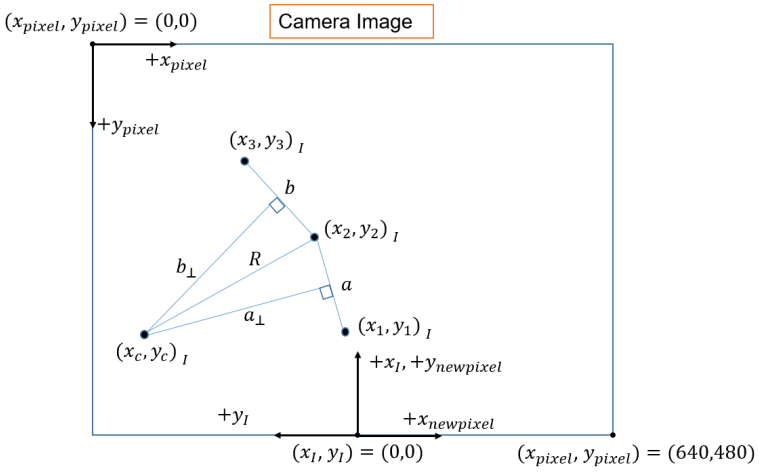


Figure 2: Image vs. inertial reference frames

2. In this section you will manually record the pixel coordinates corresponding to points in the real world (the coordinates will appear at the bottom of the window if you hover your mouse over the object). You can do it in two different ways: a) moving a small object multiple times and recording its position. b) using a grid that you draw on a large sheet of paper. Depending on if you record the positions with respect to the CoM of the car or the front bumper, you may need to add an offset in your code. Note there is a different version of **Pixel_To_Distance_Conversion.m** in each folder.

   a) Manual object approach: Start by placing an object in front of the barc at $(x_I = +1ft, y_I = 0)$, record the pixel coordinates. For example, for $(x_I = 1ft, y_I = 0ft)$, this might correspond to $(x_{newpixel} = 0, y_{newpixel} = 30)$. Repeat this for the other values in the table. Since this only records positions in the right half plane in front of the barc, it assumes symmetry of the camera about the $x_{inertial}$ centerline. In the "Manual Object Approach" folder, open **Pixel_To_Distance_Conversion.m** and replace the *xPixel_yPixel_centerline* and *xPixel_yPixel_lateral* values with your recorded pixels. You can see how it runs with the example values left in the file. Note that we are using **feet** in this approach!

(x_I, y_I) in feet	$(x_{newpixel}, y_{newpixel})$ in pixel coordinate
(1, -0.0)	
(1, -0.25)	
(1, -0.50)	
(1, -0.75)	
(2, -0.0)	
(2, -0.25)	
(2,- 0.5)	
(2, -1.0)	
(3, -0.0)	
(3, -0.5)	
(3, -1.0)	
(3, -1.5)	
(3, -2.0)	
(4, -0.0)	
(4, -0.5)	
(4, -1.0)	
(4, -1.5)	
(4, -2.0)	
(4, -2.5)	

Table 1: Recording pixel locations

   b) Grid approach: Draw a grid on large sheet of paper (see Figure 30. Make sure to boldly circle the vertices so that the camera will be able to see it. Record the values in the PixelCoordinatesSheet.xlsx in the "Grid Approach" folder. In that example, we use a 5x7 vertex grid with space intervals of 0.1m as pictured. Note that we are using **meters** in this approach!

3. Run the **Pixel_To_Distance_Conversion.m** to obtain 'f1Matrix','f2Matrix','bMatrix','xInertial_to_yPixel_Matrix', 'yPixel_to_xInertial_Matrix' arrays that will be plugged into *image_processing_parameters.yaml* on the BARC located in the *barc/workspace/src/labs/launch/ACC2018/module3/* folder. There are other important parameters here that are loaded by the lane keeping script!

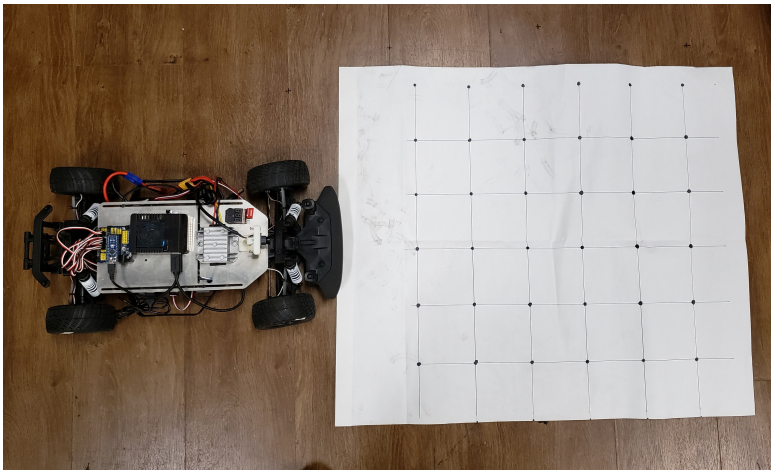


Figure 3: Grid Approach

4. Open the `image_processing_lane_keeping.py` file in the lab7 folder and carefully read through the code. This code is uses the camera to take images of a red/white road and performs some processing to extract lanes. There is an option to detect edges in greyscale rather than color by setting colorFilter to False. A reference trajectory is published to the "reference_trajectory" topic. A current example of how it is utilized is in the "mpc_reference_tracking.jl" file. For visualization purposes, the image processing node subscribes to the "optimal_state_trajectory" which will plot the optimal trajectory as calculated by your controller on the live image.

5. Modify `image_processing_lane_keeping.py` to include your controller. It is recommended that you add your own function to the "image_processing_node" class which should be called at the end of the "publish_states" function (line 394). **The low level controller is listening to the "uOpt" topic which uses a Input message made up of "vel" and "delta".** You can use the "self.uOpt_pub" publisher.

We mainly the python OpenCV library to perform edge detection on the track and extract a reference trajectory. Most of the image processing code will be provided for you. For your reference, we have linked two tutorials about the theory for edge detection in **colored**(linked) and **grayscale**(linked) images.

Feel free to modify and improve the image processing portion of the `image_processing_lane_keeping.py` code as you desire, for example by altering the filters or adding some smart planning for when the car has steered out of the lanes.

Task 3 Controller design (LQR - Optional)

You are allowed to design any controller you want such as the one in the previous simulation lab. For those of you interested in designing a reference-tracking LQR controller for lane keeping, see this section. You can use the **controlPy** toolbox for implementation in Python. Recall that in reference tracking LQR, the controller works to minimize the gap between the true state z and the desired state z_d, so that $\lim_{t\to\infty} z(t) - z_{ref}(t) = 0$. In this case the cost becomes

$$J = \int_{t=0}^{\infty} \big(e(t)^T Q e(t) + \delta u(t)^T R \delta u(t)\big),$$

where $e(t) = z(t) - z_{ref}$, $\delta u(t) = u(t) - u_{ref}(t)$, and (z_{ref}, u_{ref}) is an equilibrium point of the system.

1. First we define our reference equilibrium point (z_{ref}, u_{ref}). Recall from lecture that we can make the following approximations:

$$z_{ref} = \begin{bmatrix} x_{ref} \\ y_{ref} \\ \psi_{ref} \end{bmatrix} = \begin{bmatrix} x_{ref} \\ y_{ref} \\ \frac{v_{ref}}{R}\Delta t \end{bmatrix} \qquad u_{ref} = \begin{bmatrix} v_{ref} \\ \beta_{ref} \end{bmatrix} = \begin{bmatrix} v_{ref} \\ \sin^{-1}(\frac{l_r}{R}) \end{bmatrix}$$

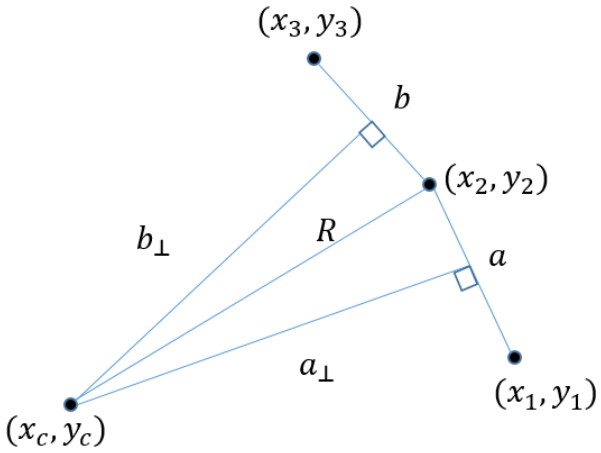


Figure 4: Finding reference trajectory

v_{ref} is our desired constant reference velocity. x_{ref} and y_{ref} are found by processing the camera image of the track. To find the radius of the track R, we need 3 points along the trajectory, as shown in Fig. 4. (x_1, y_1) represents the BARC's center of mass, $(x_2, y_2) = (x_{ref}, y_{ref})$ is the first point along the trajectory that the BARC will travel to, and (x_3, y_3) is the next point along the trajectory (used purely for calculating the radius). We can use simple geometry to solve for R:

$$\begin{aligned}
&m_a = \frac{y_2 - y_1}{x_2 - x_1}, m_b = \frac{y_3 - y_2}{x_3 - x_2} \\
&y_a = m_a(x - x_1) + y_1 \\
&y_b = m_b(x - x_2) + y_2 \\
&y_{a_\perp} = -\frac{1}{m_a}\left(x_{a_\perp} - \frac{x_1 + x_2}{2} + \frac{y_1 + y_2}{2}\right) \\
&y_{b_\perp} = -\frac{1}{m_b}\left(x_{b_\perp} - \frac{x_2 + x_3}{2} + \frac{y_2 + y_3}{2}\right) \\
&x_c = \frac{m_a m_b(y_1 - y_3) + m_b(x_1 + x_2) - m_a(x_2 + x_3)}{2(m_b - m_a)} \\
&y_c = y_{a_\perp} \; at \; x_c \\
&R = \sqrt{(x_2 - x_c)^2 + (y_2 - y_c)^2}
\end{aligned} \tag{2}$$

Confirm that this is described in the `image_processing_lane_keeping.py` file.

2. Let $e = z - z_{ref}$ and $\delta_u = u - u_{ref}$. In order to use a LQR controller, we need to compute the Jacobian Linearization of the nonlinear kinematic bicycle model about the equilibrium point (z_{ref}, u_{ref}).

$$\dot{e}(t) = Ae(t) + B\delta_u(t) \tag{3}$$

The dynamics equations can be rewritten as:

$$\begin{aligned} f_1(z,u) &= u_1 \cos(z_3 + u_2) \\ f_2(z,u) &= u_1 \sin(z_3 + u_2) \\ f_3(z,u) &= \frac{u_1}{l_r} sin(u_2) \end{aligned} \tag{4}$$

Solve for the linearization matrices A and B:

$$A = \begin{bmatrix} \frac{\partial f_1}{\partial z_1} & \frac{\partial f_1}{\partial z_2} & \frac{\partial f_1}{\partial z_3} \\ \frac{\partial f_2}{\partial z_1} & \frac{\partial f_2}{\partial z_2} & \frac{\partial f_2}{\partial z_3} \\ \frac{\partial f_3}{\partial z_1} & \frac{\partial f_3}{\partial z_2} & \frac{\partial f_3}{\partial z_3} \end{bmatrix} \Bigg|_{\substack{z=z_{ref} \\ u=u_{ref}}} \tag{5}$$

$$B = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} \\ \frac{\partial f_3}{\partial u_1} & \frac{\partial f_3}{\partial u_2} \end{bmatrix} \Bigg|_{\substack{z=z_{ref} \\ u=u_{ref}}} \tag{6}$$

3. **USEFUL NOTE.** Since the LQR gain has been computed for the error model in Eqn. (3). the controller is $\delta u(t) = -K \cdot e(t)$ which can be rewritten as:

$$u(t) = -K(t) \cdot (z(t) - z_{ref}(t)) + u_{ref}(t) \tag{7}$$

This control law is known as a "gain scheduled linear controller". In addition this controller has also a feedforward term $u_{ref}(t)$.

4. Use the following line in your python function to obtain K using the **controlpy** toolbox:

K, X, closedLoopEigVals = controlpy.synthesis.controller_lqr_discrete_from_continuous_time(Ac, Bc, Q, R, dt)

Task 4 Testing on BARC

It is highly recommended to use VNCViewer for this portion of the lab because it will be helpful to see the camera images on the odroid screen.

1. Modify `low_level_PID_controller.py` with your steering mapping equation from Lab 4. This file uses a PID controller for velocity but you can also use your derived transfer function instead. Be sure to read this script carefully to understand what is going on!

2. Run `roslaunch labs image_processing_lane_keeping.launch` to launch the image processing node, low level PID node, and other necessary nodes.

Note: *image_processing_lane_keeping.py* has a safety function. You need to press the ↑ key in the terminal in order to make the BARC move. If it detects that the path in front of it is being blocked or the path is too narrow, the BARC will stop moving. You will need to press ↑ again to make it move after removing the obstacle.