ME131: Vehicle Dynamics and Controls
Final Project Report

Trey Fortmuller and Sangli Teng

Spring 2019

1 Abstract

We present a gain-scheduling, reference-tracking LQR lane-keeping controller with feedforward and a PID longitudinal velocity controller for autonomous driving within a lane. This system is demonstrated on a 1/10th scale vehicle. The control system employs an optical encoder at each wheel for velocity estimates and a front facing camera as sensors. An Odroid XU4 embedded Linux computer and an Arduino Nano microcontroller comprise the onboard computing capability. We demonstrate safe lanekeeping and reliable performance in our test environment. A video of the vehicle driving around a track can be found here (https://youtu.be/7w4f1qW-Uts).

2 Introduction

The advent of small and inexpensive embedded computing and higher quality sensors in recent years has increased the capability of robots to navigate the world autonomously. Of principle interest to the field of vehicle dynamics and control, these same advancements bring about the possibility of more advanced safety systems and even fully autonomous driving of passenger vehicles on public roads. Freeway driving, when the vehicle spends most of its time at a constant velocity in one lane, is the most approachable problem in the domain of self-driving because it is a more structured and predictable environment than city driving. As a solution to this problem of autonomous steering and velocity control in a single lane at constant velocity, we develop two controllers which run simultaneously onboard an embedded computer.

3 Experimental Setup

The test vehicle used for development and demonstration of this system is the Berkeley Autonomous Racecar (BARC) [1] developed by the MPC Lab [2] at UC Berkeley. Is it a 1/10th scale, all-wheel-drive electric racecar, show in Fig. 1.

Figure 1: The Berkeley Autonomous Racecar

3.1 Powertrain

The vehicle is powered by a 2S, 8 Ah lithium-polymer battery. The drivetrain includes an electronic speed controller (ESC) which converts DC power from the battery to 3-phase AC for the brushless motor. The ESC also contains a 5V battery eliminator circuit (BEC) and serves as power distribution for the steering servo.

3.2 Sensors

The BARC features an optical encoder on each wheel with eight counts per wheel revolution, allowing for an angular velocity estimate of each wheel. It also features a front facing digital camera which communicates data over serial to the Odroid for processing at 30 frames per second. Color or intensity gradient thresholding on the image stream from the camera allows for edge detection within these frames, and thus for semantic data regarding relative position of the vehicle to a lane, to be extracted. The BARC also features an inertial measurement unit (IMU) consisting of an accelerometer and rate gyroscope for measurement of proper acceleration and angular rates about the IMU axis. The IMU is not utilized in our lane keeping experiement.

3.3 Compute

Low level processes are handled by an Arduino Nano microcontroller. This device reads sensor data and relays the data over serial to the Odroid. The microcontroller also receives control commands over serial from the Odroid and communicates via pulse width modulation (PWM) with the ESC. High level processes are handled by an Odroid XU4 embedded Linux computer running an image of Ubuntu Mate 16.04 maintained by the Berkeley MPC Lab [3]. The Odroid has a 2Ghz ARM-based processor and 2Gb of DDR3 RAM making it a capable platform for soft real time applications. The Odroid has several I/O ports and publishes a wifi access point for wireless communication with a host computer via secure shell (SSH) or virtual network computing (VNC).

3.4 Software

The Robot Operating System (ROS) is heavily used for inter-process communication onboard the BARC. ROS Kinetic is installed with the MPC lab's distribution of Ubuntu Mate 16.04 for the Odroid. Processes

are segmented into "nodes" which publish and/or subscribe to topics on which messages of a given data structure are communicated. Often nodes are associated with hardware interfaces, data processing, controllers, estimators, etc. The data processing, estimators, and controllers for the lane keeping system are implemented in python leveraging several open source libraries including OpenCV [4] and controlPy [5].

4 Approach and Methods

We use the lane detection code provided by the course staff but add several of our own scripts for data processing and control. We also modified the existing `low_level_PID_controller.py` to publish a vehicle longitudinal velocity estimate based on an average of encoder readings. Three of four encoders on our vehicle had reliable readings so we averaged their count rate as an input to the second order backwards finite difference method as a vehicle velocity estimate.

We added a script called `compute_radius.py` which subscribed to the `reference_trajectory` topic and computed the radius of curvature of the track ahead as an input the LQR controller. Seven x-y points in the body-fixed frame of the vehicle are returned by the lane detection system. The first point (closest to the vehicle's center of mass), and last point (furthest ahead of the vehicle), and the center point between them were used in the radius computation described in the Lab 10 documentation.

Finally, we added an `LQR_synth.py` script which subscribes to the track radius, vehicle velocity estimate, the ECU steering command, and the loop time dt to compute a discrete LQR gain matrix on each loop, then output the controller's velocity and steering angle commands to the `/uOpt` topic. The `rqt_graph` of the nodes and topics running on our system is shown in Fig. 2.

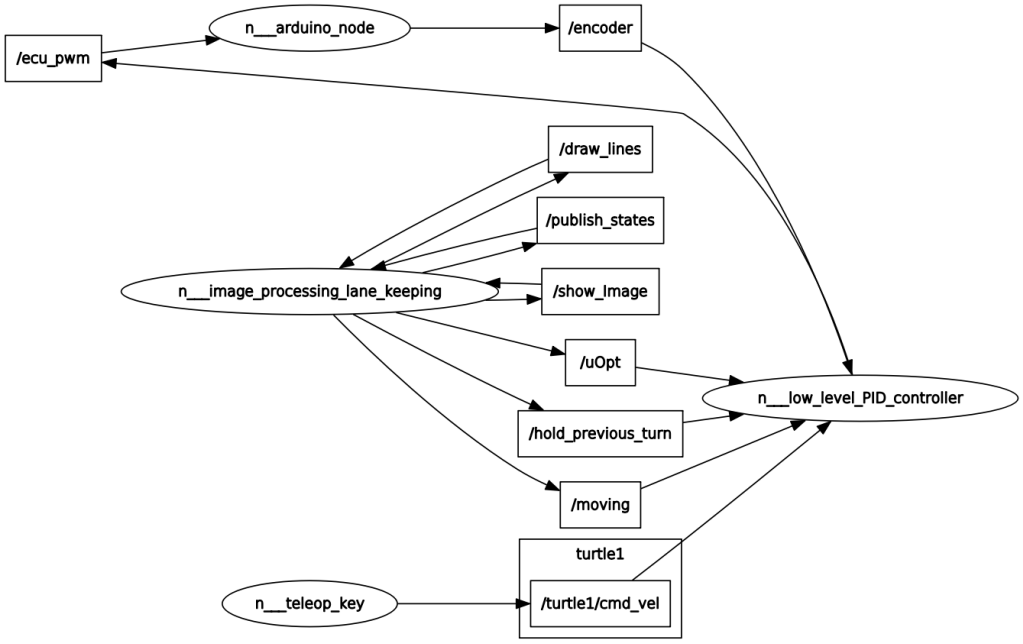

Figure 2: The RQT graph of ROS nodes and topics for the lane keeping system

We choose the nonlinear kinematic bicycle model of the vehicle as means to determine the optimal gain. The Jacobian linearization of these dynamics are computed as state transition matrices for our LQR gain matrix calculation, which is performed with the controlPy library. The control must be able to handle non-static setpoints so we choose a reference tracking LQR controller which brings the error between estimated and reference states to zero. We also include a feedforward term computed directly by the kinematic bicycle

model. The full gain scheduled, reference tracking LQR lane keeping controller with feedforward can be represented as

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

Where $K(t)$ is time time varying LQR gain matrix, $z(t) - z_{ref}(t)$ is the state error, and $u_{ref}(t)$ is the feedforward term from the kinematic bicycle model.

5 Results

After having implemented all the nodes necessary for our system, we ran into issues that made it difficult to debug our controller related to lane detection. After moving to a driving space with a homogeneous, non-reflective floor, we had success detecting lanes robustly with with our vehicle. An image of the lane detection and reference trajectory is shown in Fig. 3.

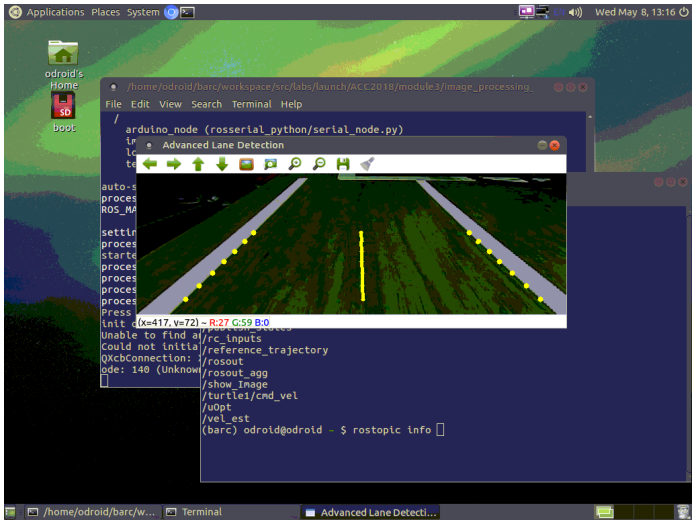

Figure 3: The lane detection and reference trajectory shown in a VNC desktop of the Odroid

We found our controller to be overactuating with the LQR cost matrices as identities. We lowered the Q matrix's cost by three orders of magnitude to avoid penalizing errors in the states so heavily. This improved the robustness of the tracking but, as expected, caused some extra variation away from the centerline on average. Upon increasing the reference velocity to 1 m/s, we saw reduced noise in the centerline trajectory and smoother driving. This performance is demonstrated in our video.

6 Discussion

The problem of autonomous navigation is hard because it consists of several sub-problems which are difficult in their own right, and solutions to these problems must work in harmony to deliver a robust system. Perception, trajectory generation, and control modules must deliver simultaneously for reliable driving. Our system focused on the control aspects of the problem, and thus shortcuts were taken in the perception and trajectory generation steps which lead to most of the issues we debugged after implementing our algorithms. The lane localization system used a simple edge detection algorithm based on finding intensity gradients greater than a hard coded threshold. This approach is not robust to changes in lighting, texture, or reflectance of the driving surface. Our vehicle struggled to reliably find lanes on shiny floors so we found a large space

with dark carpeting and used bright white tape as lane lines for maximum contrast which improved tracking performance.

Another issue we struggled with throughout the project was wireless connectivity and lag while using VNC to connect to the Odroid to start nodes and view camera images. The next generation BARC vehicle should feature higher bandwidth, 5.8Ghz wireless connectivity. If nodes for image transmission over ROS were included in the BARC libraries and local machines were configured to communicate via ROS over the network then all of the debugging could take place with SSH alone. This would prevent the wasted overhead of a full virtual desktop which is slow and unreliable.

References

[1] Zhang F. Li K. Gonzales, J. and F. Borrelli. Autonomous drifting using onboard sensors. 2016.

[2] Model Predictive Control Laboratory, 2018. http://www.mpc.berkeley.edu/.

[3] Model Predictive Control Laboratory. Barc ubuntu mate image, 2018. https://github.com/MPC-Berkeley/barc/blob/master/docs/FlashingEMMC.md.

[4] OpenCV, 2018. https://opencv.org/.

[5] M. W. Mueller. controlpy, 2017. https://github.com/markwmuller/controlpy.