Lane Keeping and Navigation Assist System

Yash Sharma, The Cooper Union
Vishnu Kaimal, The Cooper Union

Abstract—We built a miniature autonomous vehicle which can navigate through maps consisting of various road topologies. Our system is comprised of a perception module, for detecting lanes and intersections, and a control module, for lane keeping and turn making. A map was constructed for validation, and we verify that our vehicle can navigate while keeping within the lane lines.

Index Terms—Self-Driving Cars, Lane Keeping, Navigation, Perception, Control, Street Map

1 Introduction

The possibility of self-driving cars has fascinated us for decades now, but only recently have they appeared to be an impending reality. By 2007, with the final DARPA Urban Challenge finishing in a success, the seeds were planted for research in this field, and now the first commercial self-driving cars appear to be close to market. Google, Tesla, Uber, and other large players in this space, already have self-driving vehicles navigating through neighborhoods. At this point, the tasks remaining for such companies are building self-driving vehicles with inexpensive equipment such that they can be viable commercial products, and convincing the market of the safety of such systems, as one malfunction could result in death.

Despite the progress that has been made in the field, little public research is available, as most researchers working on this topic are in industry, aiming to beat their competitors in bringing an autonomous vehicle to the market. This makes the project of building a self-driving car one ripe with exploration. Therefore, we have decided to test in a controlled environment, so real-world complexity can be added for experimentation and not ever-present, making the project inviable.

We ventured out to see if we could build a lane keeping and navigation assist system which could enable vehicles to navigate a model of a real-world street map. After building our vehicle, we first implemented the lane keeping system and validated its performance on a set of lanes with varying curvature. We then concatenated the lane configurations into a map for navigation, and enabled the vehicle to handle intersections.

The rest of this paper is organized as follows: Section 2 describes our experimental setup, namely our vehicle, lane configurations, and map design. Section 3 describes our approach, our implementation of a lane keeping and navigation assist system. Section 4 describes our experimental results, both qualitative and quantitative. Finally, Section 5 concludes the paper.

2 Experimental Setup

2.1 Vehicle

In order to test a lane keeping and navigation assist system, a vehicle is needed. A prototype we used for testing can be seen in Figure 1.

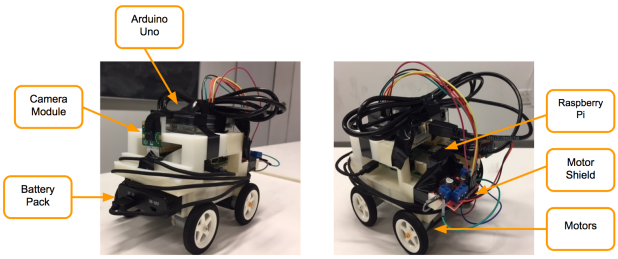

Fig. 1. Our vehicle prototype, with hardware components labeled

The vehicle was designed in Solidworks, and subsequently 3D printed in parts. As specified in the figure, the components include an Arduino Uno for motor control, a Raspberry Pi 3 for on-board processing, an L298D motor driver, a front-facing Pi Camera module, and motors to move the wheels. Separate battery packs are provided for the Arduino, Pi, and the motor driver. Holes were placed appropriately to screw the Pi into the chassis. Additionally, the axles are connected to the body via trusses. Finally, the Pi Camera is stood up using a simple cardboard stand. As can be seen, the body of the vehicle has 3 side panels to make sure the sensitive parts are safe.

The vehicle was designed so it would be 2WD. The vehicle is run off the back two wheels, the back axle is propelled by the motor. making the driving system "rear-wheel drive". Rear-wheel drive was chosen as its balance and acceleration advantages are crucial to keeping to curved lanes. The steering mechanism is implemented with differential steering, distributing different levels of power to each wheel in order to steer.

2.2 Lane Configurations

We designed our lanes as follows: For left curved lanes of angle θ, we draw a straight line from the bottom-left vertex to the right edge so the angle between the line at the bottom edge is θ. For right curved lanes of angle θ, we draw a straight line from the bottom-right vertex to the left edge so the angle between the line at the bottom edge is θ. We then fit the two vertices with a curve. We found the minimal realistic θ for lane keeping to be 40°, and thus we constructed left and right lanes at 60°, 50°, and 40°.

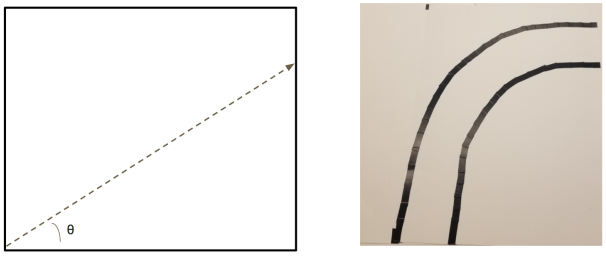

Fig. 2. An example of a curved lane used for testing

For our initial lane keeping tests, we taped the configured lanes to posterboard, as seen in Figure 2.

2.3 Navigation Map

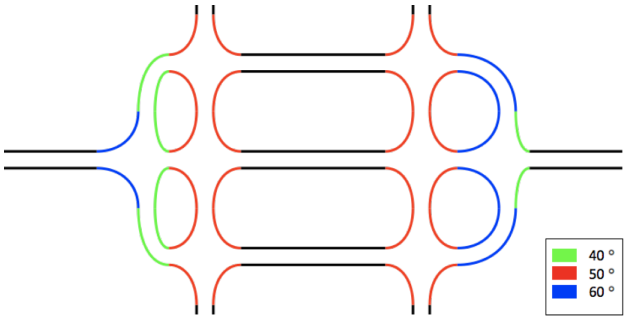

Fig. 3. Representation of our map design

Figure 3 details a labeled diagram of our navigation map. The map's size is 20 feet by 10 feet (width x height). The curves were fit with 60% bezier curves, where a 100% curve corresponds to a 90° (right) angle, and a 0% curve corresponds to a diagonal line. As can be seen, the map is composed of all of our tested lane configurations.

3 Approach

The goal of our project is to build a system which can enable a vehicle to navigate a model of a real-world street map. The basis for such a system is lane keeping, the ability to keep within a lane. However, real-world street maps are more than many lanes concatenated together, they incorporate intersections, which is what a navigation system handles. We will describe the lane keeping and navigation systems, as well as the software architecture in this section.

3.1 Lane Keeping

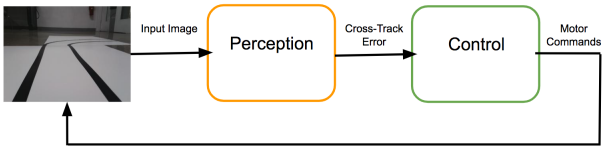

Fig. 4. System Diagram

A simplistic diagram of our proposed system can be seen in Figure 4. An input image will be fed to our perception module, which will output the cross-track error. The cross-track error will then be fed to the control module which will output motor commands that actuate the vehicle's motors, change it's state, and yield a new input image for the system to operate upon. As discussed, the goal of the system is to perform lane keeping qualitatively, and quantitatively minimize the average cross-track error throughout the time horizon.

In our system diagram, the input image is represented by an image from a front-facing camera. This is because in our system, a single front-facing camera is our only sensor modality. In addition, the output of our perception module which serves as input to the control module is solely the cross-track error. This is because we are using a PID Controller for actuating our motors.

3.1.1 Perception

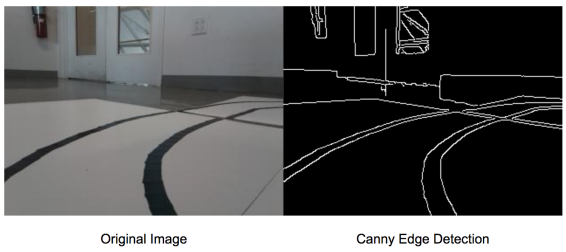

Fig. 5. Canny edge detection

In order to detect edges in the images, we applied Canny edge detection, due to its robustness. An example of its application to a sample image is shown in Figure 5. Clearly, it works very well at detecting edges, and hence it's a part of our pipeline. We then applied a perspective transform to gain a birds-eye view of the lanes. Depending on the size of the source quadrangle specified, a birds-eye view of a larger segment of the lanes is seen. This is important for avoiding latency problems by providing a look-ahead to the control algorithm.

To detect lane lines, we compute a histogram on the bottom of the image, with the peaks on the left and right halves giving us a good estimate of where the lane lines start. We then search small windows starting from the bottom and essentially follow the lane lines all the way to the top until all pixels in each line are identified. Finally, a polynomial curve is applied to the identified pixels, yielding the lane lines. As the vehicle moves, the search continues online. The histogram is recomputed after intersections. The sliding window size was tuned to be the minimal size needed to still detect the lane of maximal curvature.

In order to handle cases where the vehicle is at an angle and thus in the original image one or both of the lanes start at the left or right edges, if less than two peaks are found when computing the histogram on the bottom edge, the left and right edges are then searched for starting points.

Finally, with the lane lines detected, the cross-track error (CTE) needs to be computed to serve as input to the control algorithm. With the birds-eye view, finding the CTE is simple. As the camera is centered on the vehicle, the middle of the image is the car position, and the midpoint between

the detected lanes is the center point. If only a single lane is seen, the midpoint is taken between the detected lane and the opposite edge. The difference between the midpoint and the car position is the cross-track error. As discussed, one can define the lookahead based upon how far ahead the cross-track error is found. We find the cross-track error at the topmost point of the perspective transform output, therefore we can vary the lookahead by expanding or downsizing the source quadrangle we take the perspective transform over.

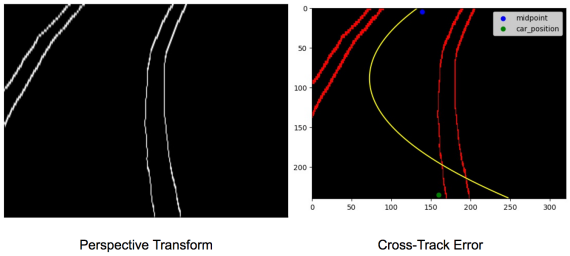

Fig. 6. Perspective Transform & Detected Lane Lines and CTE

An example of the perspective transform output and our final results, with the detected lane lines and CTE estimated, is shown in Figure 6. The yellow line in the final image is our attempt at fitting a polynomial line to serve as the center trajectory, in order to smooth CTE transitions. As you can see, the polynomial fit fails when only one lane is seen, therefore in our final system, we solely applied a large lookahead to stabilize the system.

3.1.2 Control

As discussed, the algorithm we are using for our control module is a PID controller operating on the CTE. The full mathematical form of the PID controller is defined as being $u(t) = K_p e(t) + K_i \int_{t_i}^{t} e(\tau)d\tau + K_d \frac{de(t)}{dt}$ where each of the additive terms are respectively the proportional, integral, and derivative terms, $e(t)$ is the error (CTE), and $u(t)$ represents the output control variable that should be adjusted. The proportional, integral, and derivative terms of the PID controller are used to adjust the output control variable by an amount proportional to the error, sum of past errors, and the rate of change of the error over time, respectively. The weights for each of these terms need to be tuned in order to optimize performance.

We were able to easily lane keep to a straight line by simply increasing the proportional gain, thus the real challenge was tuning the PID parameters in order to keep to curved lanes, in particular the lane of maximal curvature. In order to do this, we used the Ziegler-Nichols method, and after a reasonable number of tuning iterations, we were able to succeed with the following parameters: $K_p = 1.50$, $K_d = 0.20$, and $K_i = 0.02$.

3.2 Navigation

The challenge of navigating a map becomes exponentially easier when the map is known. In the real world, vehicles are typically equipped with GPS, which informs the driver of the topology of the road network, and localizes one's own vehicle within the map at a resolution sufficient enough for preparing the vehicle for incoming turns and intersections, but not sufficient enough for localizing other vehicles.

As GPS cannot be used indoors, we provide the vehicle with map information. We enable the vehicle to navigate from a specified source to a destination by using the map information to plan a path. This is input to the navigation system by specifying the decision (straight, left, right) to be made at each intersection in the path. A code is given for each lane segment in the map, which is used by the user to specify the source location and the desired destination.

3.2.1 Intersections

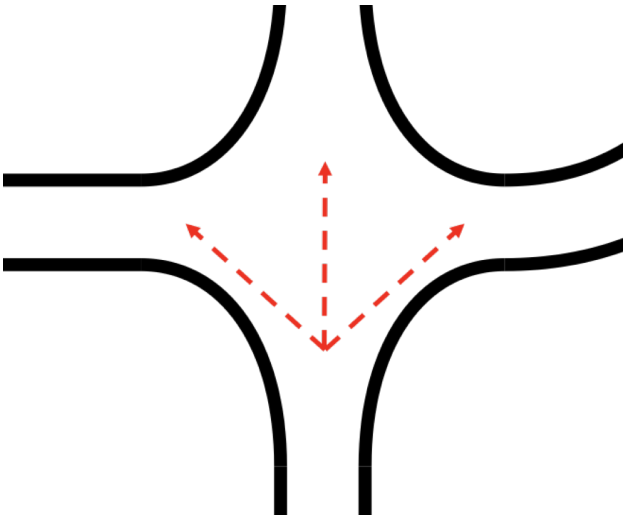

Fig. 7. An example of a intersection situation in our map

An example of a "intersection" can be seen in Figure 7. This intersection might look odd, typical intersections on real-world roads are box-shaped. This wasn't possible in our setting as the reason why vehicles can safely turn at such junctures is due to the fact that real-world roads are multi-lane. With this, vehicles can safely enter the center in order to curve into the appropriate left or right lane. In a single-lane road, turns at box-shaped intersections would require 90° shifts, which is hardly reasonable. Therefore, we gave each edge a level of curvature, enabling curved trajectories on a single-lane intersection.

Handling intersections reduces to the navigation system's ability to detect the intersection, and instruct the vehicle with the necessary motor commands needed to safely lane keep.

An intersection can be detected by simply seeing when the tracked lane lines break. Furthermore, histograms are computed on the perspective transform output, enabling the detection of the curved edges. The source quadrangle is sized such that the curved edges are present in the birds-eye view. If the specified direction is "straight", the desired trajectory is the vector mean of the directions perpendicular to the detected turn edges. On the other hand, if the path planner instructs the vehicle to make a turn, lane keeping is conducted with the associated curved edge. As the standardized lane width distance is given to the system, the desired trajectory is simply the detected edge + a $\frac{lane\_width}{2}$ offset. The lane width was set to 6 inches in our map. We

found that the PID controller used for standard lane keeping still worked for this task, meaning a second PID controller did not have to be tuned for navigation.

3.3 Software

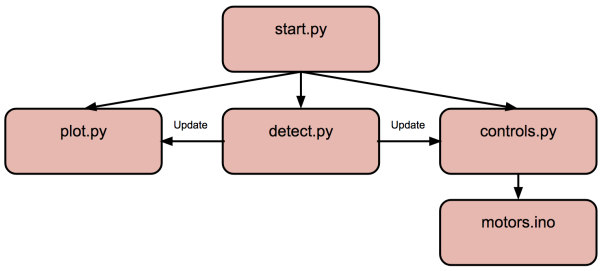

Fig. 8. Multi-threaded Architecture

In order to minimize the latency, we decided to move away from a sequential pipeline and move towards a multi-threaded architecture. A simple diagram detailing it can be seen in Figure 8.

Our multithreaded architecture is the following. We have a startup script, which instantiates 3 classes: Lane_Detection, Plot, and Control. First, the infinite loop for lane detection is started in a thread, taking in a continuous stream of images via the Raspberry Pi camera module and applying the perception pipeline, outputting the detected lanes and the cross-track error. The detected lanes are sent to the Plot class, while the CTE is sent to the Control class, via pipes. So, after a short delay (to make sure we have at least one set of detected lanes and a CTE estimate), we then instantiate the plot and control threads. The control thread takes the current CTE variable value and converts that into power to the left and right motors, which is then sent to the Arduino (via serial connection) to power the motors. The plot thread gives a continuous view on the intermediate outputs of the perception pipeline in real-time.

4 Results

Qualitatively, we were able to verify that the vehicle was successful at navigating the map and was able to keep within each lane configuration. Please view the demo video at the following link [1].

For quantitative results, we decided to setup our perception code on an overhead camera, continuously measuring the CTE for all lane configurations & turns. As an overhead camera view should give a better estimate for the CTE than a front-facing camera, we used its CTE measurements as the ground truth. We measured the magnitude of the CTE via the overhead camera for 5 vehicle runs on each lane configuration, uniformly sampling 50 points on the lane and averaging the CTE results, then finally averaged the CTE results across runs. The results are shown in Figure 9.

For reference, we found the distance between the lanes in pixels from our overhead view at a fixed altitude to be 80. Therefore, a CTE measure of 40 indicates the vehicle is riding the lane marker. As expected, the lowest CTE

1. https://www.youtube.com/watch?v=SnWBqAnbh2Q

Lane Configuration	Sampled Avg. Cross-Track Error (Mag)
Straight	7.23
Left Angled - 60°	14.56
Left Angled - 50°	22.27
Left Angled - 40°	27.11
Right Angled - 60°	13.33
Right Angled - 50°	24.24
Right Angled - 40°	30.33

Fig. 9. Quantitative Results

was yielded when keeping to the straight line, but more interestingly, decreasing θ appears to significantly increase the difficulty in performing at the lane keeping task.

5 Conclusion

Our goal was to design a lane keeping and navigation assist system which can enable a vehicle to navigate a model of a real-world street map. In order to experiment, we built a test vehicle and printed out a map consisting of lane configurations with varying levels of curvature. Our lane keeping system was composed of perception and control modules interacting with the vehicle in a feedback loop. The following cycle continuously takes place: the front-facing camera takes an image, the perception module yields the cross-track error, and the control module yields the motor commands which would propel the vehicle forward. The perception module consisted of detecting the edges in the image, acquiring a "birds-eye view" of the lanes, detecting the lane lines with a sliding window approach, and computing the error between the car position and the midpoint between the lanes. The PID controller was tuned to minimize that error. For navigation, map information was provided to the system to allow a path to be planned between a specified source and destination. Turns were handled by appropriately lane keeping to the intersection's curved edge, which was required due to the single-lane nature of our map. To minimize latency, the described systems were implemented in a multithreaded software architecture. Qualitatively, we verified that the vehicle was capable of navigating the map while staying within the lane lines, while our quantitative results indicated that increasing the curvature scales up the difficulty of the lane keeping task considerably. In future work, we hope to add real-world complexities to our controlled environment, enabling the system to navigate under environmental noise and with dynamic obstacles (other vehicles) present.

Please view the demo video showing our vehicle navigate the 20′ x 10′ map here [1].