Pololu Lane Keeping

Adrian Botvinik, Megan Joseph, Jesus Medina
abotvinik@berkeley.edu, meganajoseph@berkeley.edu, jesseomedina@berkeley.edu
Repository: github.com/abotvinik/Pololu-Lane-Keeping

1 Motivation

We were all interested in autonomous cars so we decided to focus on lane keeping on an embedded platform with the Pololu 3pi+ 2040 Robot. The goal for the project was to have the robot use a camera to stay within lanes and navigate a track. The camera would take in data and we would perform image processing via OpenCV. The positions of the lanes would then be sent to the Pololu which would figure out what mode to be in based on the information given.

We modified the Pololu by connecting a Raspberry Pi HQ Camera and wide angle lens to a Raspberry Pi 4. The pi was then connected to the Pololu Robot.

Figure 1: **The Pololu with our modifications on it.**

2. Design

2.1 Project Architecture The General Architecture is to take an Image from the road of the lane lines - these would be captured by the Raspberry Pi High Quality Camera sensor, processed by OpenCV and the relevant image information would be sent to the Pololu over an I/O protocol. The Pololu would use this information for control. The high level architecture of our design is illustrated in Figure 2.

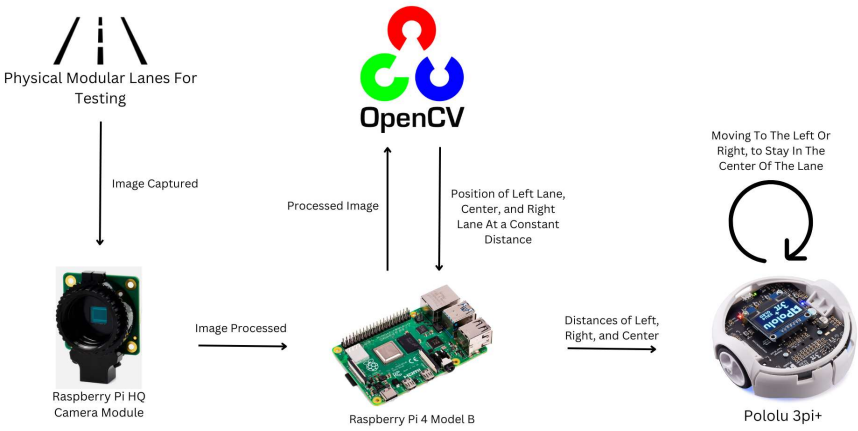

Figure 2: **High Level Project Architecture**

2.2 Class Topics. We integrated four class topics into our project. One topic we used was modal models. We used this to switch between the states Straight, Right, Left, Merge Right, and Merge Left. We also incorporated sensors - our main sensor used was the added camera. Feedback control was used to make sure the robot stays centered while driving straight and to make turns more or less aggressive. The last topic we had was input/output by connecting the Raspberry Pi to the Pololu through an I/O protocol.

2.3 Robot Design. Once we became set on the exact products we were using to implement our idea, we had to create a form factor that could support all our parts from scratch. Due to the use of the Pololu we had to figure out a way to mount the Raspberry Pi without any permanent alterations to the Pololu. Due to the fast prototyping capabilities of 3D printing this became our choice of fabrication. For our first iteration we tried to make sure we could successfully mount a 3D print to the Pololu. Using a 3D model available on the Pololu website we were able to make a negative of the shell and along with a flat extrusion on our own 3D model design, it was able to slide under the lip of the Pololu device, successfully mounting the 3D print onto the robot without any movement. We then created multiple iterations in order to fit the battery pack that is used to power the Raspberry Pi. In order to get our Camera in the correct position, we created a hinge mechanism in our camera mount that allows for easy alterations to the view of the camera. Lastly, threaded inserts were heated up and assembled into our 3D print to allow for the Raspberry Pi

and its camera to be mounted to our 3D print in a safe and reversible fashion.

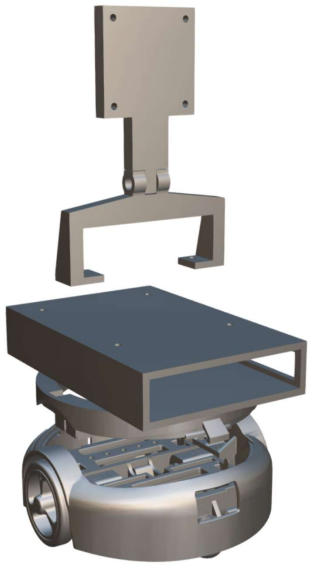

Figure 3: **3D CAD model of Our Full Robot**

3. Implementation

3.1 Image Processing. We needed to design some way to process images. Our goal for the final step was to be able to have a high contrast road for the robot to maneuver on, the intention being a very light colored road surface and very dark colored lane lines.

Instead of taking a constant video and taking stills out of this video, we decided to constantly take individual stills, and process them one by one, discarding them after that still was finished processing. This was done to conserve memory and ensure that our image processing pipeline was as efficient as possible. We landed on a resolution of 640x480, which we found to be a good mix between conserving the accuracy of the image and minimizing the space the image took up. These images were then promptly converted to greyscale. Grayscale images are stored as a matrix of single values corresponding to pixel brightness, rather than a matrix of 3 tuples that colored images are stored as corresponding to their respective RGB values. As such, not only do grayscale images perform better for various detection algorithms due to no potential of color affecting the output, but anything that processes a grayscale algorithm will run significantly faster faster, as fewer computations will need to be performed on the single value of a pixel compared to a 3 tuple.

Before investigating the specific image processing algorithm we were going to use, we developed a lane position algorithm that we could use. The idea we were going to leverage is to count the number of white colored pixels on the left and right sides of the image in a tight vertical range close to the front of the car. The position of all horizontal pixels for each side would be combined, averaged together, with the output being an approximate pixel position of the left and right lanes. These lane positions are what we intended to use for robot control and to tell what motion the robot needed to do. This algorithm is implemented as the `lr_detector()` function in `ImageProc/LaneDetect.py` within the project repository [1].

To process images such as these, we decided to first investigate a method of Edge Detection. A common Edge Detection method used and provided by the OpenCV library is called Canny Edge Detection [2]. This algorithm leverages the gradients between consecutive pixels vertically and horizontally. We first eliminated any noise in the grayscale image using a Gaussian Filter, also provided by the OpenCV library. We then fed our image to the Canny Edge Detection algorithm. During initial testing, this algorithm performed very well, outputting accurate representation of test lanes, without including much unneeded output. Unfortunately during further rigorous testing once the rest of the system was working, we found that in rare cases an abrupt change in motion of the robot would cause the image to not fully show all of the lanes. We think this is due to potential light reflection changes that may happen during abrupt movements. We also found that when we used tape to connect road segments together, light would aggressively reflect off the tape and seem like an edge to the camera, feeding false data to the car. As such, we decided to find a more robust solution.

Our final working solution is to use an algorithm far simpler than Canny Edge Detection - a simple binary filter. Once the image once again went through a gaussian filter to eliminate any unwanted noise, we fed it through the Image Thresholding algorithm [3] provided by OpenCV, with the Binary Filter flag. This solution was simpler than Canny Edge Detection, as it just analyzes the intensity of each pixel and replaces it, rather than relying on matrix multiplication that edge detection uses. As such, not only was our output more reliable, but it was also faster. Additionally, since the algorithm detected full lanes rather than just the edges, the lane positioning algorithm was more robust to potential noise that could show up. The output of our camera algorithm is shown in Figure 4, with the first image illustrating the lane detection algorithm as well.

For the algorithm to work, it was important to determine what values correspond to the left and right lane lines being centered. Once the camera was affixed to the Pololu and angled to where the robot was barely in view, placed the robot down on a straight section of our track and noted down the values the camera output. These would be the values we used to determine if the left and right lane lines were centered, and would be hard coded onto the Pololu for this purpose of checking for the center.

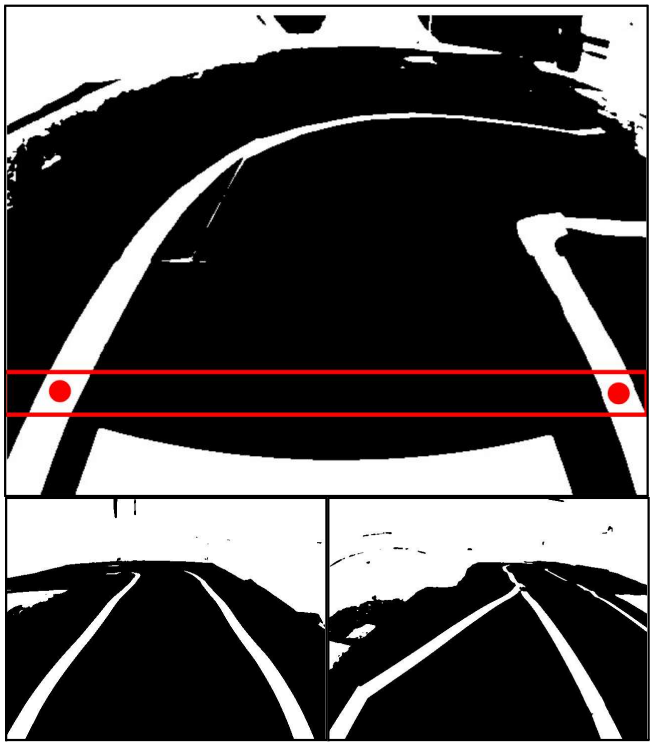

Figure 4: **Processed Images and Lane Detection**

3.2 Pi to Pololu Linking. Once the image is processed, we are left with two integer values that contain the left and right lanes. To minimize potential for error in transmission, we decided to encode both of the integers into one integer - the integer values are constrained by the horizontal resolution of 640, so we can easily mask the one number into the other and decode it on the receiving end. The result is a single integer, whose most significant 2 bytes are the left lane value and the least significant 2 bytes are the right lane value.

Our chosen protocol for transmission between the Raspberry Pi and the Pololu was UART. We chose UART compared to other Serial protocols for its ease of use, use of only two pins, and the need to only communicate with one other device, as compared to I2C which is recommended to be used to connect multiple devices. Along with no need for a clock signal and easy implementation, UART became our choice for cross communication between the Raspberry Pi and the Pololu.

The Pololu polls for the UART signal once every 100 ms. In the event that a UART signal is present, the Pololu sends the encoded integer to a standalone Lingua Franca reactor. Executing as a timer every 100 milliseconds, this reactor will check and take in the encoded integer containing our two left and right lane values, finishing once it receives the end of the integer. This reactor processes the single integer sent into the needed values for the left and right lanes, and sets them as outputs of the reactor. This way, our robot within the main reactor can react to changes to this lane position, these values are then used by both the modal model as well as the feedback controller.

3.3 Modal Models. Modal models were used to switch states based on the positions of the lanes. We first made functions to check where the lane is. We check if the left and right lanes are approximately centered, relative to the hard coded centers and with a certain threshold since lane values will never be exactly centered. This threshold can be easily modified, and it was tuned to make the car respond as accurately as possible.

Our use of Lingua Franca allowed for straightforward Modal Model implementation in our code. We define all the relevant modes for Straight, Turns and Merges, and in reaction to changes to the lanes, the modal model reacts, changing speed according to our control scheme as well as checking if the car needs to enter a different mode for turns. We also added a standby mode - this mode is the initial state of our program and it is only switched to once the Raspberry Pi starts feeding information to the Pololu. Within the regular movement modes, you can stop the Raspberry Pi processing, sending a signal to the Pololu to change the mode to standby and wait for movement input to restart.

Our initial mode is in Standby where it is stopped. If a lane is detected, it will move into the straight mode. Proportional control is implemented here so that the car will stay centered and correct itself if it strays too far to either side. From there, if the left lane disappears, it will move into the left turn mode. Similarly, if the right lane disappears, it will move into the right turn mode. If the left lane is in the centered position but the right slowly converges to the middle, it will move into the left merge mode. Similarly, if the right lane is in the centered position and the left lane slowly converges to the middle, it will move into the right merge mode. These two merge modes are analogous to a real highway when the amount of lanes on the roads reduces, and one lane merges into the other, causing you as the driver to also need to merge. Lastly, if no lanes are detected, it will stop and move into standby mode until lanes are detected again.

In the left turn and right turn modes, proportional control is used to determine how aggressive the turns should be. They are based on how far away the turning lane is from its correct position. Once the lane line corresponding to the direction of the turn is back to the central position, the car goes into the straight mode. If the other lane line disappears in the middle of the turn before the car fully completes its turning maneuver, the new lane line being lost takes precedence, and it executes the turn in the opposing direction until it has recentered in that direction.

Within our code, we had to implement multiple sequential modes to accomplish a merge. Because lingua franca can only set motor speed once in any given reaction, so we cannot have the car both turn and go straight for the initial motion of the merge within one mode. In these modes, we

have the car first turn for 300ms in the direction of the merge. Then, it will drive straight until the opposite lane disappears and will move into the opposite turning mode to recenter itself in the lane.

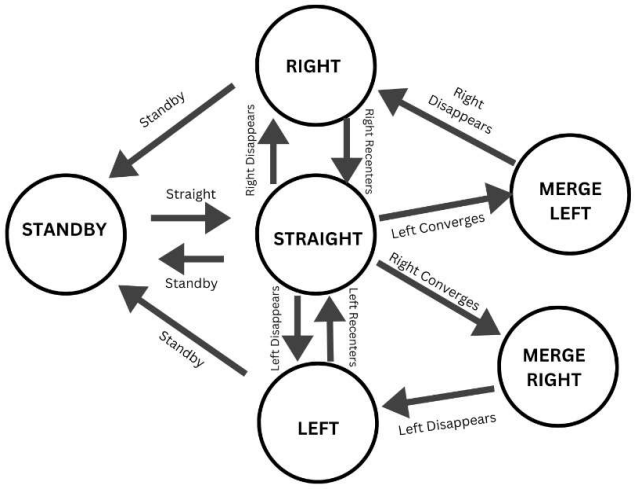

Figure 5: **Our modal model system**

3.4 Feedback Control To keep the robot stable during its motion, we decided to use a Proportional Feedback Control System. Our goal was to be able to keep the car stable within the middle of the lane while it was in the straight mode and so it would make turns tailored to their needed intensity, rather than fixed radius turns until it was centered.

In initial testing, we noticed that the Pololu would not go straight when equivalent motor power was applied to both sides. We determined that this was due to both the Pololu itself as well as the not symmetric weight distribution of the top hat with our added parts on top of the Pololu. With 15% power to each motor, the Pololu on its own would stray slightly to the left and with the added weight of the top hat it would stray further to the right. As such, we determined that adding an additional 1% power to the left motor as its base speed when going straight would most accurately keep the car straight. This 1% offset was used for both turns and straight modes for the base speed, before our proportional term was applied. When determining the proportional terms, we were mostly working off trial and error to accurately estimate the terms that we used to offset from the base motor speed. Our goal was for the offset to not be too aggressive to potentially send the car off the opposite side we are correcting for and not gradual enough to where it didn't do much.

While the car is in the straight mode, we wanted the car to be able to stay roughly centered while it was in motion. Because the image was 640 pixels wide, we wanted the car to be centered at around 320 pixels. Our control system for the straights would be to keep the car as close to a 320 pixel average as possible, a deviation to the left meant we needed to apply a right offset and a deviation to the right meant we applied a left offset. Through testing, we concluded that a 2% offset for every 100 pixel deviation from the center was the best speed . Our straight motion control system is as follows, where `avg` is the computed average between the left and right lanes and `base` is the base motor speed of 16 for the left and 15 for the right motors.

$$Motor\ Power\ \% = \ base\ + \frac{2(avg-320)}{100}$$

While the car is in one of the turning modes, we wanted the turning speed to be based on how aggressively the lane in the direction of the lane line on the outside of the turn, as that is the lane line that is still visible at all times through a turn. We determined through testing that we wanted for a 100 pixel offset, the motor on the inside of the turn would apply a -6% offset and the motor on the outside would apply a +4% offset. We also found that in rare cases, that the lane line on one side would fall into view of the camera algorithm for the other side, leading to extremely aggressive turns and even spins when this occurred. As such, we decided to cap the maximum offset to the offset for a 100 pixel deviation, which was enough to complete even the most aggressive turns but wouldn't cause the Pololu to go awry. Our turn motion control system is as follows, where `pos` is the position of the lane corresponding to the turn, `cntr` is the fixed center of the lane line that is turning and `base` is the base motor speed of 14 for the left motor and 13 for the right motor.

$$Inside\ Motor\ Power\ \% = \ base\ - \frac{6|pos-cntr|}{100}$$

$$Outer\ Motor\ Power\ \% = \ base\ + \frac{4|pos-cntr|}{100}$$

4. Evaluation

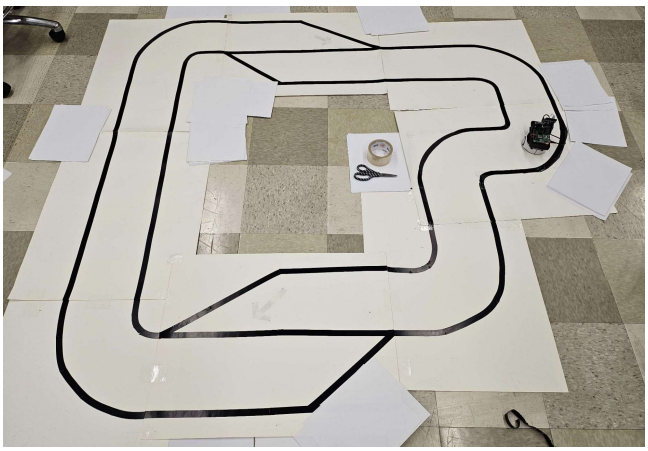

Figure 6: **Final Testing Track**

Our goal for evaluation was to make some track that the robot would be able to traverse completely without running

into any major issues while traversing it. This track needed to be able to test all major aspects of the robot, which are going on straights, making turns and executing merges in the event. We decided to make a full loop, where it would do all of these things, and the test track is shown in Figure 6. This full loop has both right and left turns as well as straits. It also has lanes converging both left and right to test the merge functionality of the robot. Although it would be more realistic to have dotted lines in the moments where there are 2 lanes, our car's reliance on lane line existence for keeping its straight trajectory prevented us from making dotted lines. With sufficient tuning and testing, we got the Pololu to complete the track consistently in the bright and consistent lighting environment of the class lab. Unfortunately in final testing, we noticed that the solution we had was very reliant on the very good lighting in the lab. As such, whenever there was inconsistent lighting the robot would stray off track. It would still reliably stay off the track for most times, so we aimed to do testing in an as well lit environment as possible.

Another element of our analysis that we did was to take a look at the lane values, to see the effectiveness of our algorithm on a minute scale though the data isn't very useful, it is still fairly interesting to see, as per Figure 7. In the vast majority of cases, whenever there are flat portions in one lane line, there is a spike in the opposite lane line - this is indicative of the turn functionality responding to the disappearance of a lane. The few times that spikes correspond on both sides are exactly when lane lines occur, as you can see before, one lane is going towards the middle as. You can see this happening at time step 55-69 for a left merge and time step 239-256 for a right merge. This shows that although our algorithm is not the most robust, it is still very effective at executing our goals.

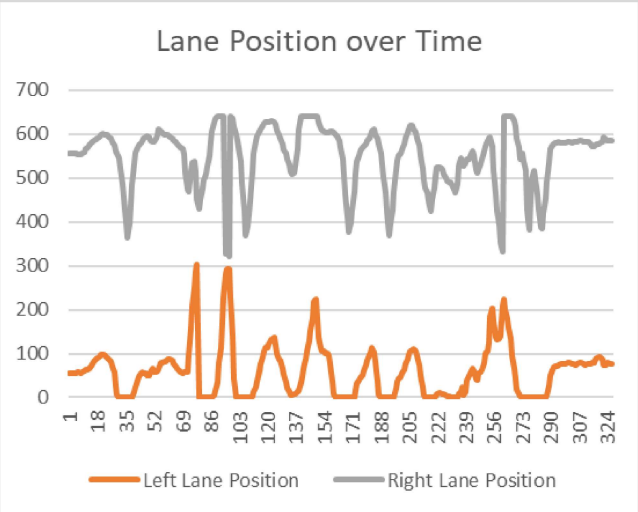

Figure 7: **Lane Position over Time**

5. Conclusion

5.1 Next Steps Although we are satisfied with the outcomes of this project, given more time, we definitely could have added far more capability. Currently our car is constrained to styles of road similar to those used in the final evaluation. Although it achieves our goals, there is a lot more room to expand. Additional features we could add in the future include merging onto our road from another road that comes from the side, similar to an exit on or off ramp from a traditional highway. We attempted to see if this would work with our current solution, but the reliance on the lane lines being present meant that passing this on ramp while not trying to take it meant that it would make an unexpected and unneeded turn, which is not what we wanted. In the future we could also expand and add a more distance based control system, where it would not only center itself in the lane, but it would detect the type of turns coming up and even the angle, preparing the control input it needed in advance to the car arriving. This would likely need additional sensors to accurately measure the distance to the next road feature, as well as a complete revamp of the control algorithm.

5.2 Closing Nonetheless, we are extremely satisfied with the work done for this project. We applied many of the topics we learned about in class and executed a very strong project. We have all learned a lot about a wider variety of topics outside the class. Using this, we all have more insight into the world of embedded systems, and how it applies to autonomous vehicles, an interest that all of us had. We are proud of the final result, and are excited to see where we can take this project to the next level.

REFERENCES

[1] *https://github.com/abotvinik/Pololu-Lane-Keeping*

[2] *Canny edge detection*. OpenCV. (n.d.). https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

[3] *Image thresholding*. OpenCV. (n.d.-b). https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html