A Basic Autonomous Vehicle For A Track That Uses Lane Detection Via Canny Edge Detection And Hough Line Processing Using OpenCV

Nathaniel Biddle

University of Cincinnati

Abstract

This project is a demonstration of lane detection and navigation via those lanes. Navigation was achieved via canny edge detection, followed by creation of Hough lines from the detected edges, and then angular adjustment via calculation of the vehicle's angle in relationship with the angle of the vanishing point of the detected lines. The angular adjustment is controlled via the PID controller, which utilizes established PID gains to dictate the final angular command sent to the robot. The controller created allows for a simulated autonomous vehicle with simulated kinect depth camera to, starting on a track with variable width yellow track lines, to navigate around that track. In the project, I demonstrate this controller working and plot the results of the autonomous vehicle going around the track several times. I was successful in implementing the desired behavior and identified areas where future improvement could be made and where there are limitations in the control method.

Introduction

The motivation for this project comes from the goal of creating a controller which can respond to the environment without predefined waypoints, for instance, based on sensor feedback. Existing literature and concepts that this project was based on are the concept of a proportional-integral-derivative controller, Canny edge detection, Hough line transformation, and projective geometry. It additionally builds on concepts from a tutorial and established functional implementations within OpenCV and Numpy for various transforms.

The proportional-integral-derivative controller (PID controller) was introduced in theoretical form in 1922 by Nicolas Minorsky in regards to steering ships (Minorsky, 1922). Having gone over this concept in class, I decided to utilize this process, with the main purpose of this endeavor to, instead of utilizing pre-existing waypoints, to take sensor data in the form of images provided by a depth camera, and calculate error based on that. PID controllers utilize a set of control terms (gains) which are coefficients used to fine-tune the model for success with a given task. Limitations of this controller style will be discussed in the Conclusions section.

Various methodology exists for gathering useful information for a given task from an image. In the case of lane detection, two useful methods are Canny edge detection and Hough line transformations. Canny edge detection was established by John Canny in 1983 (Canny 1983) and is an edge detection algorithm that can detect edges in an image and be tuned using various parameters. Hough line transformations, created originally for machine analysis of bubble chamber photographs (Hough 1959), but the rho-theta methodology utilized in this project and used in modern contexts was described initially by Richard Duda (Duda 1972). Hough transforms allow for detection of any defined shape and so allows us to detect lane lines from the perspective of a camera.

Finally, projective geometry, a subsection of the broad field of perspective geometry, the projective subsection regarding the study of geometric features of projections of one coordinate space into another coordinate space. Of particular importance to this project is the concept of a vanishing point and the recognition that it can be utilized to align vehicle trajectory. Projective geometry is a field that became formalized by various artists of the Renaissance period like Leon Battista Alberti (Wright 1984), with the concept of aligning a vehicle's trajectory utilizing vanishing point not having a clear initial conception. Utilizing this concept and the other aforementioned concepts, I was able to implement a PID controller which finds lane lines in images, then calculates error for the PID controller utilizing the vanishing point calculated for the two lane lines at a given time.

Approach

The PID controller was implemented utilizing the methodology described in class, based on the initial implementation provided in the homework assignment #8. Changes I have made include elimination of the waypoint usage, robot position tracking, and changing of the error calculation function

to utilize lane detection via image recognition. I additionally no longer needed to subscribe to odometry data, so I removed that, but continued to publish cmd_vel and angular_vel.

To capture image data, I utilized the ROS depth camera integration (which simulates a kinect depth camera: https://classic.gazebosim.org/tutorials?tut=ros_depth_camera), utilizing the tutorial from class for setup. This camera returns 640x480 resolution color images and the feed from the camera can be subscribed to at /camera/color/image_raw. Once I had an image feed to subscribe to within ROS gazebo, I could begin crafting my lane detection algorithm.

The approach to lane detection that I use was influenced by an open-source tutorial on medium titled 'Simple Lane Detection with OpenCV' (https://medium.com/@mrhwick/simple-lane-detection-with-opencv-bfeb6ae54ec0), but with many changes to the approach as a result of the specialized requirements of the project. That tutorial instructs on Canny edge detection, Hough transformation, and how to do it on a frame-by-frame basis. They applied it then to a video, but not to a live video feed of a robot, which is where the methodology had to deviate significantly. Additionally, my use case was in a different environment and required various changes to functions described in the tutorial related to matrix format for the image, OpenCV color transformations, different coordinate system considerations, and, of course, the actual utility of a vanishing point, something not considered in the tutorial, which only covered detecting lines in an image.

The integral methods of the lane detection algorithm are the pipeline method and the calculate_steering_angle_from_lines method. The pipeline method is a heavily altered version of the concepts in the above mentioned tutorial. It does also call out to two boilerplate methods, region_of_interest and draw_lines, which were used with slight alteration with attribution to the author, attributions in the code. The calculate_steering_angle_from_lines method is my own code, based on the concept of vanishing point of the lane lines generated in the pipeline method.

The pipeline method does a series of OpenCV transforms. OpenCV is a programming library for real-time computer vision. The first transform carried out is the cvtColor method, which converts the raw image from the BGR color space to the HSV color space. The reason this is done is to make it so that the newly represented image separates color from lighting. Once that is done, having defined the values within which the yellow color for the lane lines falls within, I created a binary mask using the range, which converts all yellow colors to white and all others to black; then, using OpenCV's bitwise_and method, I isolate that which was white from that which was black, thus making it so that the only remaining content in the image is the yellow lines. This allowed more versatility for the algorithm, making it so that it can function in any area where there are a lot of lines, as long as the other lines are not within the yellow range, and it also served the purpose of explicitly solving a canny edge detection horizon line problem I was running into. Prior to isolating the yellow, the Canny edge detection was picking up the horizon, which was problematic for the logic of the controller, issue shown in figure 1 below:

Figure 1: Horizon Line Interference Example

Still part of the pipeline method, after the image with isolated yellow lanes was created, I used the cvtColor method from OpenCV to transform to gray scale for edge detection. OpenCV has a built-in method called Canny which does Canny edge detection (https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga04723e007ed888ddf11d9ba04e2232de), taking 3 parameters, those being the image, threshold1, and threshold2. Threshold 1 is the hysteresis threshold lower bound, and 2 is the upper bound. Hysteresis thresholding is an aspect of the Canny algorithm where edges are determined to either actually be edges or be false edges. After canny edge detection is done, I then use OpenCV's HoughLinesP method (https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga8618180a5948286384e3b7ca02f6feeb), which does the Hough transforms based on input parameters image, rho, theta, threshold, lines, minLineLength, and maxLineGap. These parameters dictate which of the edges generated by the Canny edge detection are actually lines that we are interested in. For instance, if a determined line is below minLineLength, it will be rejected. Below shows an example of the image transforms that occur during the pipeline method:

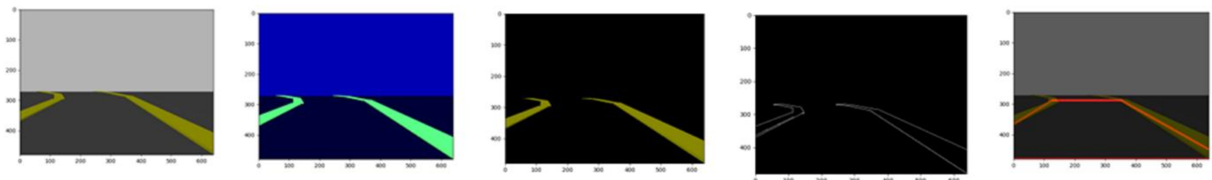

Figure 2: Full Lane Detection Image Progression Visualization

Once the Hough transforms have occurred, the rest of the pipeline method determines logic for deciding if the robot is not heading toward the vanishing point and, if not, what steering_angle to return if it isn't. With the lanes determined, there are 3 possible scenarios (on a compatible terrain that doesn't have more than 2 yellow lines at a given time), those possibilities being that the camera detects both lines, one line, or no line. I hold Boolean values locally in the pipeline method, hasLeft and hasRight, which remain false unless two lines are returned by the Hough transform after all of the edge detection occurs. The happy path scenario is that two lines are returned, in which case I calculate the slopes of both, which are used in calculating the steering_angle below.

The next two code blocks in pipeline, which are blocks starting with conditionals "if left_line_y and left_line_x:" and "if right_line_x and right_line_y:" respectively, their purpose serves to let the pipeline algorithm know whether or not hasLeft and hasRight are true, as steering_angle relies on it. Those code blocks also can be used for logging functionality (the above data visualization's far righthand side image is an example of the outputted image, where the lines are drawn on for a researcher/user to log to some file or to visualize what is happening in the algorithm). Steering_angle is initialized as None, and if hasLeft and hasRight are true, the calculate_steering_angle_from_lines method is called. If hasLeft or hasRight (or both) are False, then the current rudimentary corrective measure is to return steering_angle of -1 or 1, depending on if hasLeft is true or hasRight is true, respectively, else return 0.

The calculate_steering_angle_from_lines is the method that utilizes vanishing point to determine the steering_angle to return to the calculate_error method. This method takes the lines from pipeline as parameters and finds the intersection point, which is their vanishing point. The lines are represented in the form $Ax + By = C$, then the determinant of the system of two equations is calculated to check if the lines are parallel (in which case, there would be no vanishing point) and, if not, the point of intersection is calculated. There is a redundancy for scenarios where this method is called but less than two lines are provided, though for that to occur, there would have to be an unknown bug in the script, as it appears that that cannot occur based on current script. The next step is to adjust the coordinates of the point of

intersection so that instead of the origin being upper left, as is the case for OpenCV images, the origin is bottom center, which is the point closest to the simulated camera (for all intents and purposes). Once adjusted, the vanishing point angle from a horizontal line at the bottom center of the image is determined, then returned in radians, returned back to the pipeline method, which returns the steering_angle to the calculate_error method.

To create these two methods was difficult to do by testing directly in ROS Gazebo, so to make it more manageable, my process for implementing the code was to save opencv image data, then to test the algorithm as a simulation outside of gazebo to see if it would follow the expected path, then once confirmed, I added it to my gazebo controller as the control method. Initially, I tested on a single image, to fine tune my parameters for the Canny and Hough algorithms, to test encoding related things, and to figure out appropriate coordinate system adjustments. Once I was satisfied with determined lines on a static image, I tested against a video; the way that I created the video was utilizing line 81-85 (currently commented out) in my finalprojectcontroller script, which wrote images to a local location on my desktop. I then utilized ffmpeg, a video/image manipulation software that allows you to create videos from many images. I used ffmpeg to create the video, then tested the lane detection on the video. The testing on the video brought awareness to edge cases like the algorithm falsely identifying the horizon line as a lane, which led to isolating the yellow color space of the road lanes. It additionally helped in bringing awareness to several limitations of the algorithm, which will be discussed in the conclusion section.

Once I was successful in tracking lanes through a whole video, I moved on to making it the navigation scheme for the PID controller. The calculate_error method takes an input image, runs the lane detection pipeline on it, then returns 0 if None is returned, else it returns the steering_adjustment angle, restricted to an error value between -pi and pi. This is assigned to self.error, which is utilized in the pid_controller method, which carries out, based on gain coefficients, adjustments to trajectory of the autonomous vehicle around the track. This is all initiated by the run method.

The run method sets the rate of image capture, counts steps for measuring previous and current error, publishes twist messages to change trajectory based on the changes in variables based on the calculate_error and pid_controller methods being run, and, at current time, it progresses indefinitely, so will continuously go around the track (unless the navigational system fails to do so in a given iteration, which can be influenced by things like loop iteration and GPU speed, particularly in a simulated setting. The run method is initiated after the FinalProjectPID class is instantiated via the main method, which occurs after ROS is initialized via the rospy.init_node command.

Results

I was successful in implementing a control algorithm that uses lane detection to autonomously drive a simulated robotic vehicle around a yellow laned track. I tested several times, finding that the wider the lane (likely up to a given lane width), the more successful the algorithm would be, as there is not great error handling, particularly if the vehicle goes outside the outer lane, with some semblance of correction should it go outside the inner lane. I plotted the results of the vehicle going around the track 3 times utilizing the control algorithm, as shown in Figure 3 below. You will also find video demonstrations of two track scenarios attached to submission of the assignment, as well as visualization videos of lane detection before an after (with an overlay of lane delineation in the after portion).

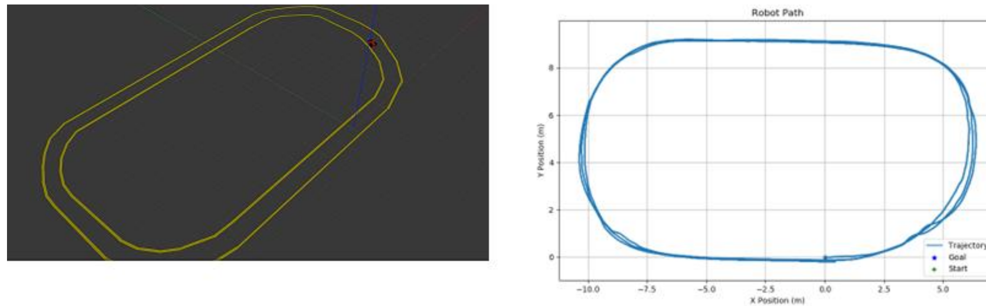

Figure 3: Autonomous Robot Vehicle Path Plot

The result was a successful control algorithm, with some limitations, but that is successful in its task, albeit not perfect at this point. Additionally, the below figure shows an alternative track path example:

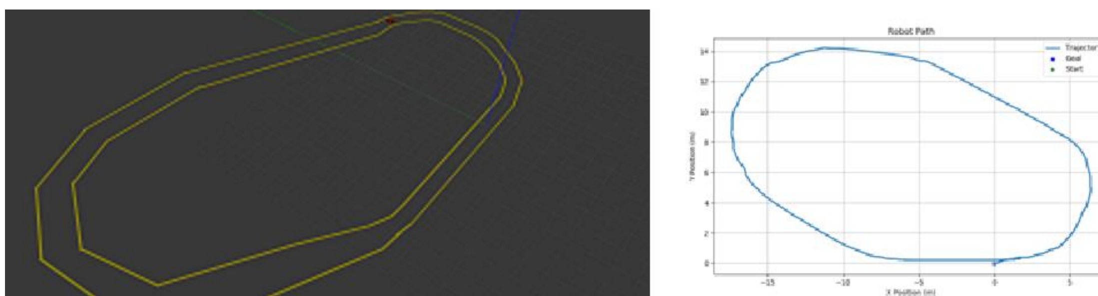

Figure 4: Alternative Track Robot Vehicle Path

This control algorithm won't work on every track with yellow lines, with exceptions being discussed in the conclusions section below. The control algorithm is fuzzy to some degree, as a result of the iteration loop and it being a PID controller, with gains, rate of image capture, and gpu/cpu functionality (for instance, a faster gpu/cpu might lead to overshooting in some scenarios if this is simulated). Overall, the result was a functional lane detection algorithm for use with a simulated autonomous vehicle that uses a PID controller in ROS Gazebo.

Conclusions

There were several lessons learned through this project. I learned more about the utility and effectiveness of lane detection and edge detection algorithms, as well as the complexity and nuance involved with them. Additionally, I learned of limitations to my implementation, as well as limitations to a PID controller. I was able to achieve my goal of creating an autonomous control system that allows for navigation around a track utilizing lane detection and OpenCV.

The current system only works in some environments because it uses static PID gains and it uses static Canny and Hough parameters. In a future iteration, I could conceive of the idea of adjusting PID gains dynamically or adjusting Hough/Canny parameters dynamically. A limitation of delineating by a defined yellow color space is that you could have various yellow objects in the environment and, in a real world scenario for example, a line could have breaks in it or have something in front of it. The PID controller was limiting to some degree in its sensitivity to changes in processing time for methods in the controller class (the equivalent of a time-sensitive method being degraded in reliability due to things like

dropped frames/stale frames in a 3D graphics setting, for instance). As a result of this fuzziness, there does remain a non-zero chance that (especially in very thin yellow lanes), that the vehicle may veer outside of the lanes, and the error handling is not very robust at this point for those scenarios. Other limitations of the current methodology include that it requires consistent elevation of the navigation path and does not handle extremely sharp turns well (and as lane width decreases, inability to handle sharp turns increases).

Some things that could be done in the future are adding additional logic for stopping, slowing down, and edge detection features that would allow for more than 2 yellow lines to be within the defined region of interest for the camera without error. A much larger task, but one that could also be added, would be the recognition of other moving objects in front of you, such as other cars, which can be used as a linear velocity reference point and a trajectory cue in lieu of lane lines that they may be obfuscating with their car. Error handling for scenarios where the vehicle leaves the lanes for whatever reason would be a good feature to add in the future too, to increase robustness and reliability of the system. Finally, I could add a test suite which uses procedural generation to create a number of different road shapes to test against. The system is effective for its purposes at current time, with many opportunities for increasing reliability, flexibility, and robustness existing for future iteration.

References

1. Minorsky., N. (1922), DIRECTIONAL STABILITY OF AUTOMATICALLY STEERED BODIES. Journal of the American Society for Naval Engineers, 34: 280-309. https://doi.org/10.1111/j.1559-3584.1922.tb04958.x
2. Canny, John F.. “A Variational Approach to Edge Detection.” *AAAI Conference on Artificial Intelligence* (1983).
3. Schlafly, Roger G. "Phased Array Antenna System." U.S. Patent No. 3,069,654, 18 Dec. 1962, https://patents.google.com/patent/US
4. Duda, R.O.; Hart, P. E. (January 1972). "Use of the Hough Transformation to Detect Lines and Curves in Pictures". Comm. ACM. 15: 11–15. doi:10.1145/361237.361242. S2CID 1105637.
5. Wright, D. R. Edward (1984). "Alberti's De Pictura: Its Literary Structure and Purpose". Journal of the Warburg and Courtauld Institutes. 47: 52–71. doi:10.2307/751438. JSTOR 751438. S2CID 195046955.