Final Challenge: Utilizing Computer Vision for Lane Following and Traffic Optimized A* Trajectory Planning for City Driving

Team 10

Cynthia Cao
Trey Gurga
Grace Jiang
Toya Takahashi
Jonathan Zhang

RSS

May 14, 2024

Contents

1 Introduction

(Authors: Cynthia Cao, Jonathan Zhang)

The final challenge was composed of two parts; first, successfully driving around the Johnson track at 4 m/s while staying in a designated lane, and two, driving through a simulated city environment in the Stata basement while obeying traffic laws. Both tasks require effective integration of various tasks our team had worked on in previous labs, including computer vision, PID control, localization, path planning, and obstacle avoidance. Similarly, in the real world, useful autonomous robots need to be capable of a similar range of tasks, whether they are self driving cars providing ride-sharing services such as WayMo or self-piloting military drones such as Anduril.

Our implementation of track racing can be split into two key components: line detection and line following. The robot finds the lane lines using a combination of color segmentation, to filter out non-white lines, and Hough transforms, a computer vision technique suited to finding lines and circles in images. Using the detected lines, a goal point can be found, which the robot then drives towards guided by a PD controller. The difficulty of this task lay in the need for incredibly precise tuning when driving at high speeds.

Our second objective, successful city driving, requires a car that can plan paths to three goal locations, stopping for five seconds when each goal is reached. Furthermore, the car is required to drive on the right side of a pre-drawn trajectory at all times, stop a safe distance in front of stop signs and traffic lights, and avoid hitting pedestrians and other obstacles. To accomplish this task, each task had a dedicated solution: a modified path-planning algorithm was used to ensure the robot drove on the correct side of the road, computer vision was used to identify stop signs and traffic lights on the way, and our safety controller implementation from Lab 3 was used to avoid hitting pedestrians.

2 Final Race

2.1 Technical Approach

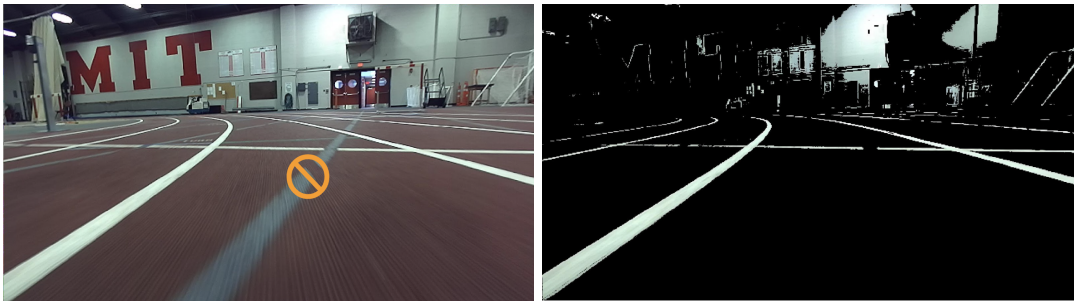
The goal of the final race portion of the lab was to drive our car at 4 m/s in a full loop around the Johnson track while staying in a designated lane and avoiding collisions with other cars or objects. Lane crossings are counted as infractions and are to be avoided.

To complete this portion of the final challenge, we used a combination of color segmentation, Hough transform, homography transform, and PD control.

2.1.1 Lane Detection

(Authors: Cynthia Cao, Toya Takahashi)

Lanes are detected using the image from the onboard ZED camera. Initially, we filter out unwanted lines using color segmentation. Specifically, the grey lines along the track are detected by our Canny Edge Detector, motivating us to filter out grey lines by hue. This is done using color segmentation in the hue-saturation-value (HSV) color space. Specifically, we use a minimum value of $[45, 0, 180]$ and a maximum value of $[114, 47, 255]$ to filter out non-white lines while still effectively detecting the lanes. The process is clearly visualized in Fig. 1, shown below.



(a) Track with Unfiltered Gray Lines (b) Track after Color Segmentation

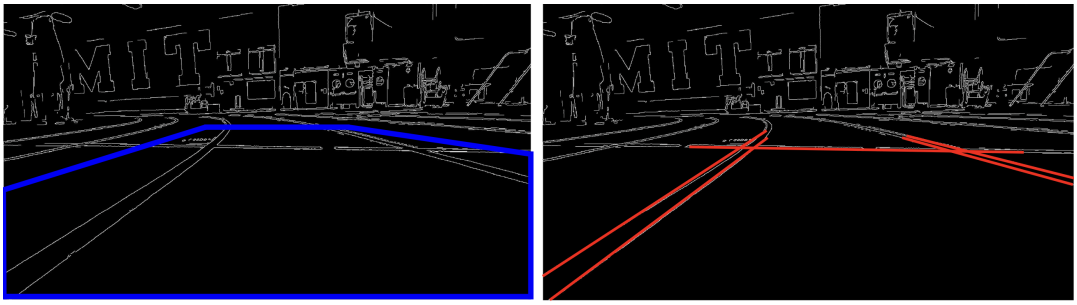
Fig. 1: **The gray lines detected in (a) are clearly filtered out in (b), allowing the robot to accurately find lane lines**. These values are specifically tuned to the lighting conditions in the track, and may not be as robust elsewhere.

Next, a Gaussian filter is applied to the color segmented image to remove noise from the image. The Gaussian filter is an $n \times n$ symmetric matrix computed as $G(x, y) = \frac{1}{2\pi\sigma^2} \exp(-\frac{x^2+y^2}{2\sigma^2})$ where for our purposes, $n = 5$. This kernel is passed through each pixel, and reduces the amount of unnecessary sharp edges that may be detected by the Canny Edge Detector.

Once the Gaussian filter is applied, the Canny Edge Detector extracts the edges from the input image. The edge detection works by using four filters to detect horizontal, vertical, and diagonal edges in the blurred image, and from these new images, the edge gradient, which shows the direction of maximum change in intensity, is calculated. Based on the calculated gradients and threshold values we define, the algorithm chooses which points to keep, and a binary image with only the edges is returned.

Fig. 2: **Gaussian filter and Canny Edge Detection extract pixels on the edge of the lanes.** The Gaussian filter blurs the image by convolving it with a 5×5 kernel, preventing the edge detection algorithm from picking up noises.

Finally, we overlay a polygonal mask to the edge image to only keep our region of interest (ROI) which contains the lanes the robot is following as shown in Fig. 3. The cropped image then gets passed to the Probabilistic Hough Transform which identifies the lines. The Probabilistic Hough Transform is an efficient implementation of the standard Hough Transform and works by converting each point in the image space into a line in the Hough space and searching for intersection points.



(a) Region of Interest for Finding Lines (b) Finding Track Lines

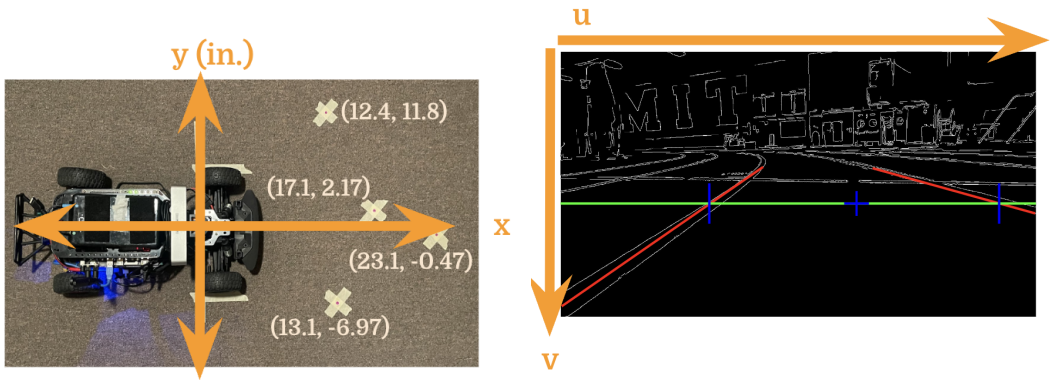
Fig. 3: **The camera image is cropped to a specific region of interest in order for the robot to find the correct lane lines to drive in**. This region is tuned for counter-clockwise driving around the track, and allows our robot to stay within the lanes while racing around the track.

To filter out the outliers which are white lanes crossing the track, we also ensure the slope of the lines we are detecting to be greater than 0.2, which is a number we determined through experimentation and recording multiple rosbag files. Once all lines except ones which are on the edge of the lane the robot is following are filtered, the y-intercept and slope of lines with the same slope sign are averaged. In other words, we assume that the left line has negative slope in the image space (since the origin is at the top left corner) and the right line has positive slope, and we average out the lines on each side of the lane to find the left and right lines in the image.

2.1.2 Lane Following

(Authors: Trey Gurga, Toya Takahashi)

Once the two lines which represent the edge of the lane the robot is on are calculated, we find the point for the robot to follow by finding the midpoint between the two lines at a "lookahead" y-value in the image space. The midpoint is then inputted to the homography transform we developed in Lab 4 to convert it into robot space coordinates (x, y) as shown in Fig. 4. Then, we find the desired angle θ for the robot to follow by calculating $\theta = \arctan(y, x)$. While angle θ is the only feedback used in our controller loop, using the homography transform rather than using data directly from the image space was necessary because the ZED image topic we were listening to was only published from the left camera, which caused unwanted distortions in the image.



(a) Finding Homography Matrix (b) Homography Transform on Track Lines

Fig. 4: **The homography transform converts 2D pixel points into a point on the ground for the robot to follow.** The matrix is found using the method shown in (a) measuring real points on the ground and comparing it to their pixel locations. The green line in (b) is the "lookahead" y value and the blue cross hair represents the point the robot follows.

Our control system is visualized in Fig. 5. For our feedback controller, we used a PD controller with gains $K_p = 0.2$ and $K_d = 0.1$, angle θ as the measured feedback, and output steering angle ϕ to quickly reach our desired setpoint of 0° while preventing excessive oscillations. We did not use an integral term to prevent integral windup which would cause the robot to develop an underdamped response, overshooting the desired setpoint. Additionally, since even a small steering angle can cause the robot to turn a significant amount at a high speed of 4m/s, we limited the controller output ϕ to be in the range of ±0.04rad.

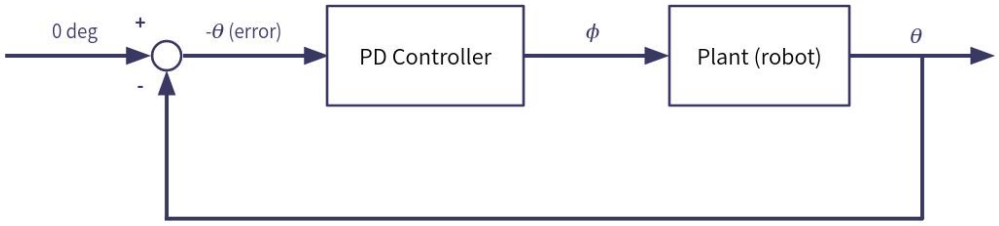


Fig. 5: **Our PD controller allows the robot to accurately track the lane without excessive oscillations.** The controller outputs the robot steering angle ϕ, and the lane angle θ, which is calculated using the lane detection algorithm, is used as the negative feedback in the control loop.

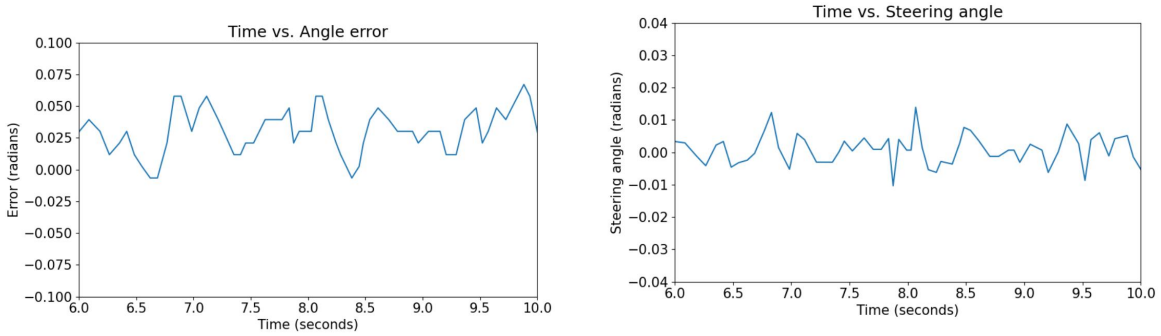
2.2 Experimental Evaluation

(Authors: Toya Takahashi)

With our implementation of the lane detection and lane following algorithms, our robot was able to complete a loop around the Johnson Track on any lanes, and during the final challenge, our robot finished the race in 53 seconds with no penalties.

Throughout our time testing, we had difficulties with the robot occasionally failing to subscribe to the image topic published by the ZED, despite experimenting with the QOS queue size for the ROS2 subscriber. The exact cause of this issue remained unclear, and even when the ZED images were published, it sometimes only published at a slow rate of 5Hz. Furthermore, our original robot base had a significant drift which caused the robot to inconsistently veer left, even when adding an offset steering angle. While developing a more robust line following algorithm by transforming the entire lane into robot-space coordinates may have helped compensate for drift, we decided solve this issue by replacing the wheelbase of our robot, significantly reducing the drift we saw earlier in lab. Despite these challenges we faced, our robot was able to successfully accomplish the task without penalties.

In order to evaluate the robot's ability to follow the racetrack, we measured the calculated angle θ to the lookahead point and the steering angle ϕ when moving at a constant speed of 4m/s. As shown in the generated plots in Fig. 6, the robot is able to accurately follow the racetrack without line breaches, with average angle error between 3 to 4 degrees. While we found the PD gains of $K_p = 0.2$ and $K_d = 0.1$ to be adequate for this challenge, given more time, a more accurate tuning of the gains could have improved our time to complete a lap around the Johnson Track.



(a) Plot of time vs. angle to the center of the lane

(b) Plot of time vs. steering angle (control effort)

Fig. 6: **When going in a straight line, the PD controller allows the robot to track the lane with an average error between 3 to 4 degrees.** To compensate for this error, the controller outputs a small steering angle between -1 to 1 degrees.

3 City Driving

(Authors: Trey Gurga, Cynthia Cao, Jonathan Zhang, Grace Jiang)

3.1 Technical Approach

The goal of the city driving portion of the lab is to successfully reach three goal locations while adhering to the rules of the road. The robot must navigate through a modified Stata basement complete with traffic lights, stop signs, pedestrians, and a center lane. Staying on the right side of the road, the robot must reach three locations selected by a TA and stop at each for 5 seconds. Running a red light, ignoring a stop sign, crashing into a pedestrian, and crossing the center lane are penalized.

To accomplish these objectives, we combined a variety of ideas from previous labs, including A*, localization, and pure pursuit, along with new ideas that were brainstormed for city driving specifically.

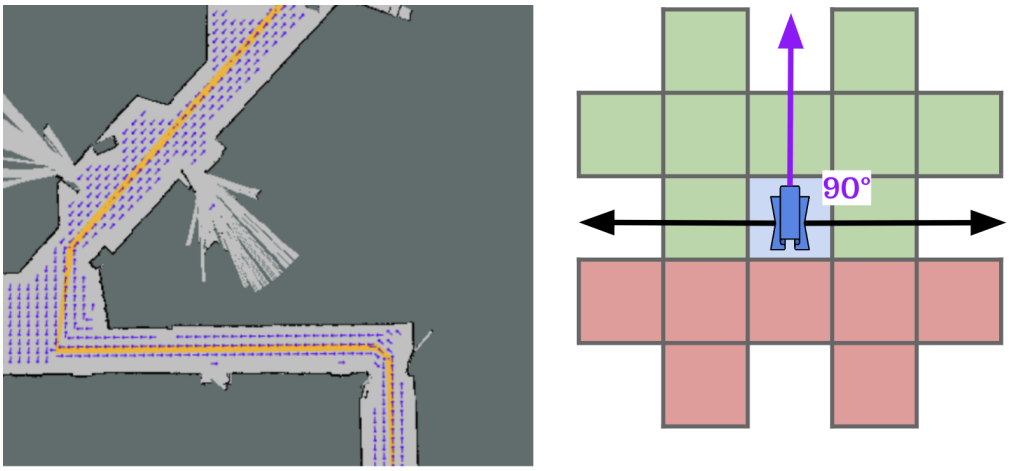
3.1.1 Right Lane Following

The primary technical challenge of city driving that differs from previous path planning exercises arises in the integration of traffic laws. Ensuring the vehicle remains on the correct side of the road poses a significant hurdle, particularly when incorporating U turn functionality. Allowing the vehicle to cross lanes during U-turns complicates matters as simply treating the line like an obstacle is rendered ineffective.

We approach the problem from a path generating perspective. If we can generate paths that follow traffic laws, we can keep pure pursuit and localization the same and the robot will end up following traffic laws just by following our paths. Furthermore, if we can edit the map in some way, we can continue to use A* and find optimal paths. The question that follows is how we can edit the map.

We accomplish this through adding a vector field to the map. For each unit cell in the 2d grid representation of the map, we define a vector pointing in the direction of the flow of traffic at that x,y coordinate space (as seen in Fig. 7a). We accomplish this through the algorithm as follows. We first define the road centerline as a series of directed line segments in the map (with the points given to us by the TAs). Next, iterating through each x and y in our 2d grid representation, we find the directed line segment within the road centerline closest to that point. After doing so, we take the cross product between that directed line segment and the x,y coordinate to find the side of the directed line segment that the coordinate is on. A positive cross product signifies being on the left of that directed line segment, and a negative cross product signifies being on the right side of that directed line segment. If the point is on the right side, we give it the unit vector of that directed line segment, and if the point is on the left side, we give it the opposite. In this way we define traffic for every point in our map.

We then use this vector field to modify our A* algorithm. For every unit cell in the 2d grid representation, instead of defining neighbors in all the cells around it as we did previously, we now take out the neighbors opposite the direction of the flow of traffic at that point, which we accomplish through seeing if the dot product between the traffic vector and neighbor vector is positive or negative (negative meaning it's on the other side and we take it out) (see Fig. 7b).



(a) Vector Field (b) Modified A* Neighbor Finding

Fig. 7: **A vector field around the center trajectory modifies the neighbor-finding in the A* path planning algorithm to avoid crossing the center line**. (a) visualizes the vector field generated around the orange trajectory, while (b) shows the valid neighbors for the car when in line with the vector field.

In this way our modified map and A* algorithm will allow our robot to follow traffic laws (see Fig. 8). The only options for our path to take is to follow the direction of traffic because we take out the neighbors that go in the opposite direction. Furthermore, this implementation itself will also include U-turn implementation, since the robot can only cross to the other side by crossing perpendicular with the road centerline, and finally, the algorithm will be efficient and optimal because we continue to use A*.

(a) Path Planning on the Outer Lane (b) Path Planing on the Inner Line

Fig. 8: **(a) shows accurate path planning while driving along the outer lane. (b) similarly shows our robot planning paths without crossing the center while driving along the inner lane**. Our robot is able to remain on the correct side of the track at all times when generating paths, allowing it to city drive without incurring traffic penalties.

3.1.2 Map Modifications

While testing the path planning algorithm in simulation, we found that our modified A* algorithm incorporating the vector field could not find a path through certain regions of the map. For these edge cases, we removed the vector field and allowed any neighbors to be selected for the next step of A*. In these cases, we cannot violate the traffic rule of driving on the right side of the road, as the regions are in locations where the robot will not be crossing into an illegal part of the road. The modified vector field with these regions removed is pictured in Fig. 9 below.

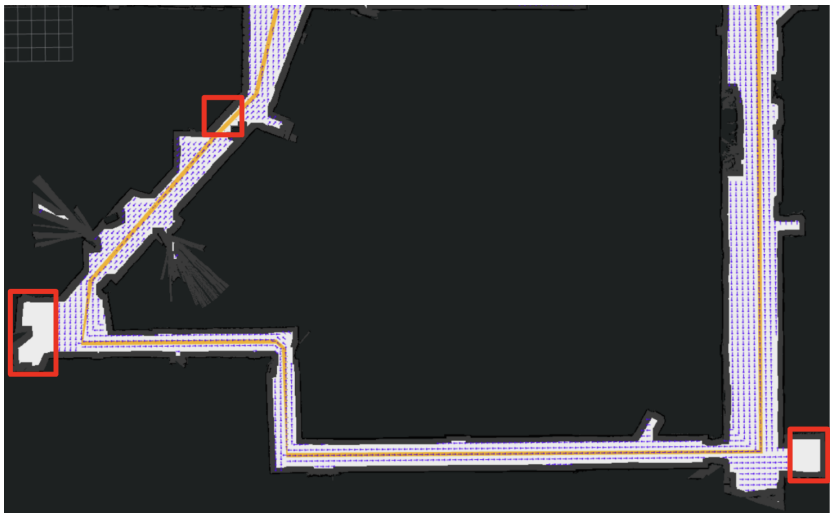


Fig. 9: **Removing the vector field in certain regions allows the robot to travel freely in those areas, preventing cases where no path can be planned.** In the figure, the bounding boxes outlined in red highlight areas where we removed the vector field rules for neighbor finding and allowed all neighbors in A* search.

For our implementation to work in real life, we needed to further adjust our map representation. First, we dilated the center lane divider by a radius 5 pixels with a cost added to the A* algorithm if the path lands on any of the dilated pixels. The robot has a width in real life, so this dilation allows the width of the robot to avoid the center line rather than be almost on top of it. Additionally, since the center trajectory given to us in simulation does not exactly match the one taped on the floor of the Stata basement, this edit ensures that the robot will avoid the center line even if it is shifted slightly in real life. The visualized dilation on the modified map can be seen in Fig. 10, along with our dilation of the walls similar to what we did in Lab 6 for regular path planning.

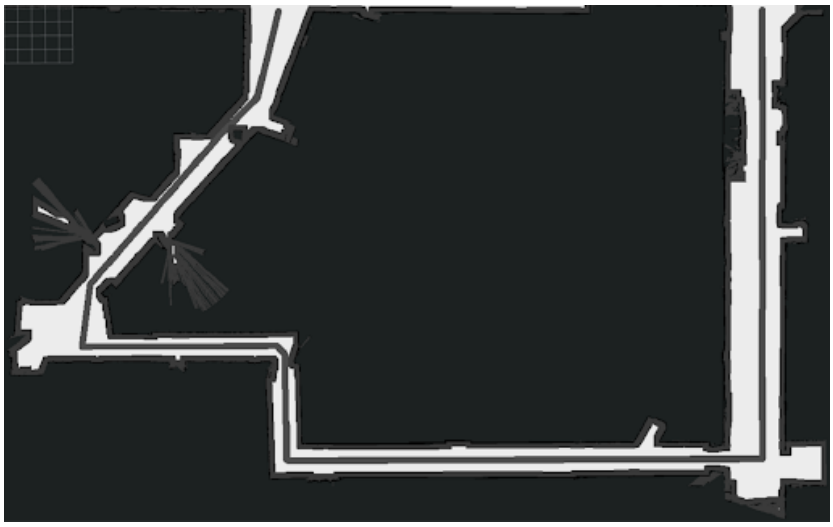


Fig. 10: **Dilating the center trajectory gave a safety margin for the robot to avoid crossing the orange tape in the physical Stata basement.** The center trajectory has been dilated 5 pixels on both the right and left side, which we calculate by iterating over white regions on the map and adding a cost to the map if a pixel has a Euclidean distance within 5 pixels of the closest point on the trajectory. These costs are later incorporated in A*.

3.1.3 Monte Carlo Localization

In order for our robot to know when a goal is reached and where to plan paths along its trajectory, we use Monte Carlo localization. In Monte Carlo localization, a particle filter composed for a motion model and sensor model allow the robot to predict its current localization. The motion model updates each particle prediction using odometry data, while the sensor model computes the possibility that the robot is at any given particle using the LiDAR. There are many locations in Stata basement that appear uniform to the LiDAR scan, necessitating careful tuning of the odometry in order for our robot to successfully localize. We found that adding Gaussion noise with standard deviation of 0.2 to the x odometry, standard deviation of 0.1 to the y odometry, and standard deviation of $\pi/24$ s to the θ component of the pose, allowed fro robust localization throughout the entire basement. Fig. 11 shows our robot successfully localizing along a trajectory in the city driving environment.

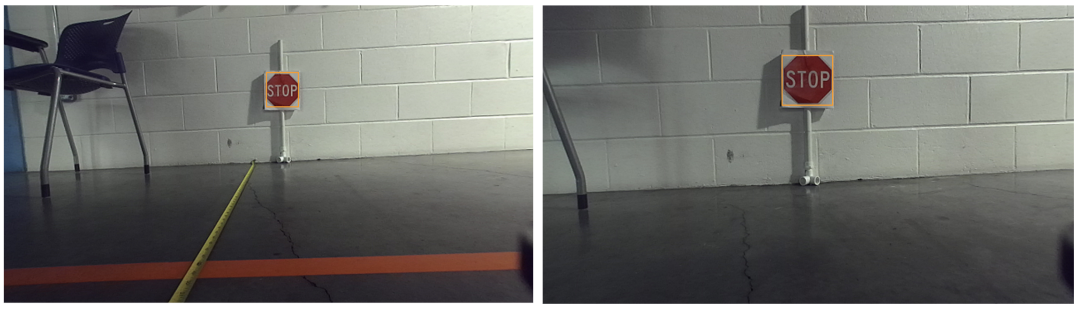
(a) Localizing at Initial Goal Point (b) Localizing Along New Trajectory (c) Localization at Final Goal Point

Fig. 11: **(a), (b), (c) show the robot successfully localizing during city driving along various points in its trajectory.**

3.1.4 Stop Sign Detection

(Authors: Trey Gurga)

Detecting and appropriately responding stop signs was crucial for the navigation of autonomous vehicles in city driving.We implemented stop sign detection system using a ZED camera and a YOLO model. The system is designed to recognize stop signs in real-time, issue stop commands, and ensure the vehicle resumes driving after a brief stop, even if it still sees the stop sign.



(a) Filtered Out Stop Sign (b) Detected Stop Sign

Fig. 12: **The bounding box around the stop sign in (a) is smaller than our threshold, while the bounding box in (b) crosses our threshold**. Thus, the robot stops when it sees images akin to (b), but continues driving when seeing images such as (a). This allows our robot to stop 1 meter away from stop signs.

System Overview The main components of the system include:

- **Camera Input:** Capturing live video feed from the vehicle's surroundings using a ZED camera.
- **Image Processing:** Converting the ROS image messages to OpenCV format for processing.
- **YOLO Model:** Utilizing a pre-trained YOLO model to detect stop signs in the video feed.
- **Vehicle Control:** Publishing commands to stop or resume driving based on detection results.

When the stop detector receives an image message, it converts the image from ROS format to OpenCV format using CvBridge. The YOLO model processes the image to detect stop signs and their bounding boxes. If a stop sign is detected, the function calculates the area of the bounding box to estimate the distance to the stop sign. A bounding box exceeding a certain threshold indicates the stop sign is close, prompting the system to start a stopping sequence. If the stop sign is deemed far, the system ignores it and continues driving. This threshold was set by placing the stop sign 1 meter away and calculating the bounding box area; this yielded an area threshold of 2500px. At higher speeds this threshold was lowered so that the robot had adequate time to come to a stop within 0.5-1 meter away from the stop sign. This boundary of detection is illustrated in Figure 12.

When a close stop sign is detected, the system begins a timer to stop for a set time of 3 seconds. 3 seconds ensured that the robot came to a complete stop and not a "California stop". After stopping, the robot is still identifying a close stop sign, so the system ignores further stop signs for 5 seconds to prevent repeated stops. The stop command is published to the safety topic in the Ackermann mux ensuring that no other commands take priority over the stopping command.

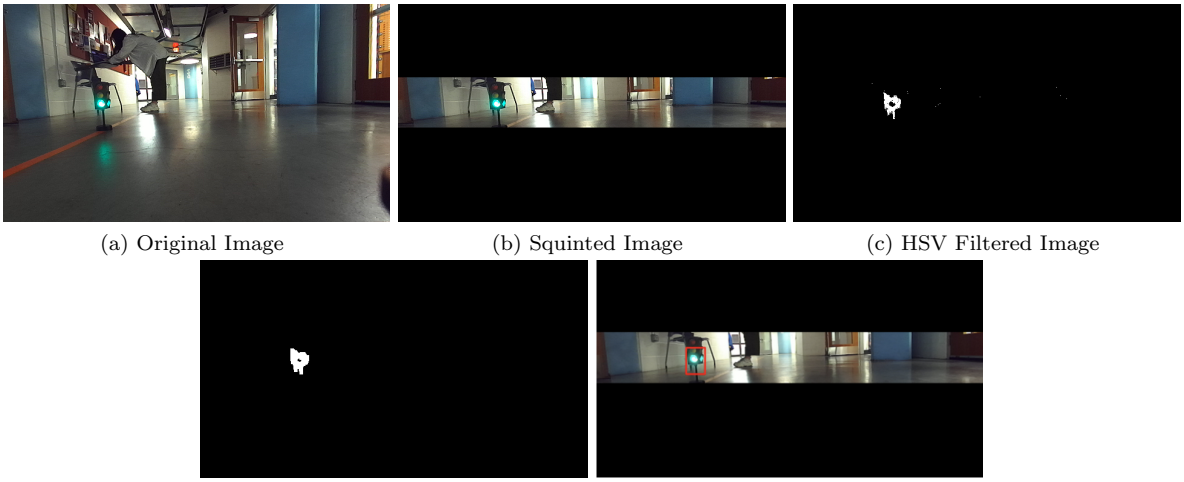
YOLO (You Only Look Once) is a real-time object detection system. Unlike traditional object detection models that apply the model to an image at multiple locations and scales, YOLO reframes object

detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. This approach allows YOLO to achieve high detection accuracy with remarkable speed, making it highly suitable for driving at fast speeds where the robot must react quickly to identified stop indicators. Our Zed camera only published images at 10 frames per second, so the quick processing of the YOLO model allowed adequate time to process and stop before the stop sign. Thus, the stop sign detection system leverages computer vision and machine learning to ensure the vehicle can autonomously detect and respond to stop signs. This integration enables the vehicle to navigate safely and efficiently in environments with traffic control signals, enhancing the overall reliability and safety of autonomous navigation systems.

3.1.5 Traffic Light Detection

(Authors: Grace Jiang)

Traffic lights are detected using color segmentation. We tuned HSV values based on images of the traffic light as seen through the robot's ZED camera, and eroded and dilated the resulting filtered image to filter out noisy colors and enlarge the detected areas for the traffic light color. An illustration of this process can be seen in Fig. 13 below.



(a) Original Image (b) Squinted Image (c) HSV Filtered Image

(d) Eroded and Dilated Image (e) Visualized Bounding Box

Fig. 13: **Color segmentation accurately determines the existence of a green light.** With a mask, we first black out the top 1/3 and bottom 4/7 of the ZED camera image (squinting, b). We then filter out all pixels not in an HSV value range (c). After eroding and dilating this filtered image (d), we then find a bounding box for the green traffic light (e). In this particular example, the bounding box was found to be 1333 pixels in area.

In certain designated traffic light locations pictured in Fig. 14, we begin to look for green or yellow light through the ZED camera and issue stop commands to the safety topic if no bounding box with a size of at least 100 pixels is detected, filtering on both green and yellow HSV values.

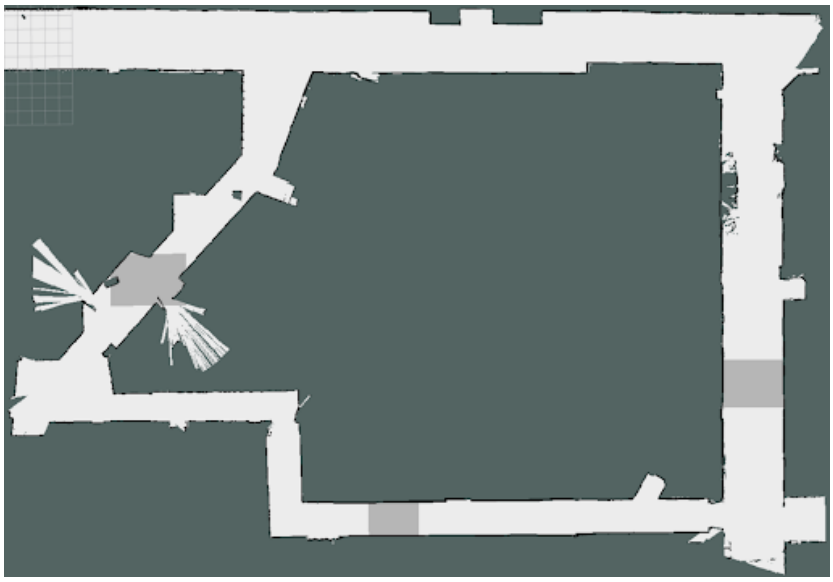


Fig. 14: **Stopping only in designated traffic light regions prevents unnecessary robot stops.** In the above map of the Stata basement, white regions are regions where the robot cannot encounter a traffic light, while light gray regions are those where a traffic light is visible.

If a green or yellow light is detected, it is safe to continue forwards and the robot will resume to follow drive commands from the trajectory follower. After the robot has traveled 0.5m since entering a traffic light region, the view of the ZED camera will have passed the traffic light. As there are no green or yellow lights to detect, stop commands will no longer be issued regardless of whether a bounding box for green or yellow light is detected.

3.2 Experimental Evaluation

Our implementation of city driving was capable of gaining reaching three goal locations and returning to the starting location in 1 minute and 20 seconds. Furthermore, we were able to successfully stop at one stop sign and traffic light during our runs of city driving. This process required many iterations on dilation of the map, variable penalties for driving too close to the line, and further tuning of our pure pursuit controller.

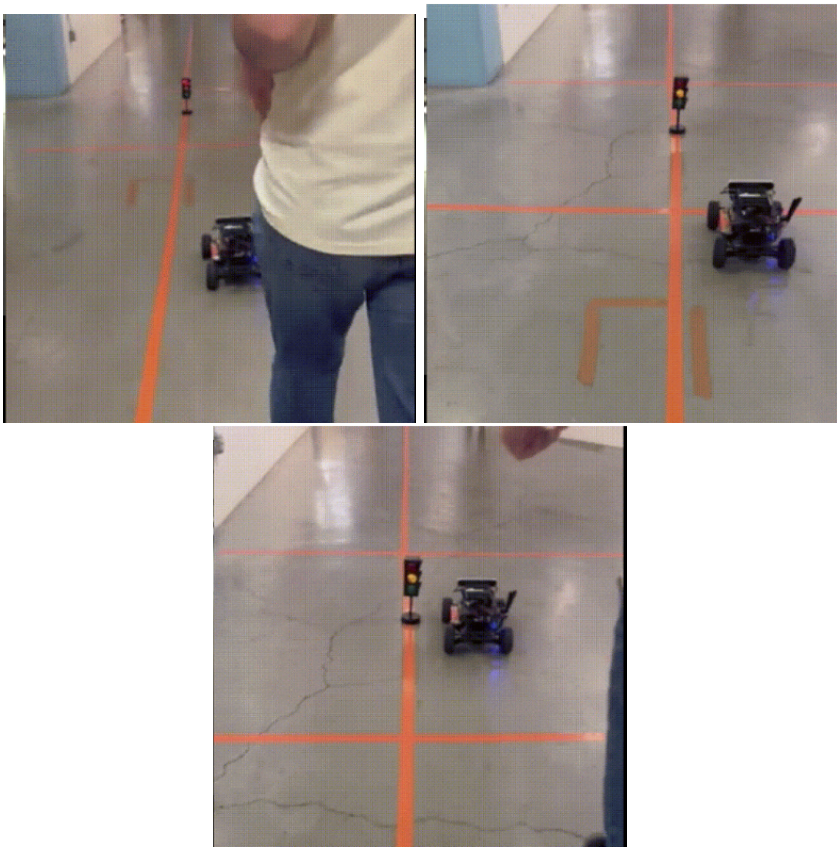


Fig. 15: **The robot successfully stops at a red light, before continuing through the yellow light.** Although inconsistent, we are able to successfully stop and go in real life in the city environment

4 Conclusion

(Author: Cynthia Cao)

This paper presents our approach to the final challenge, implementing solutions to tasks such as lane following, path planning, stop sign and traffic light detection, and localization. Our robot was able to successfully drive along a race track at 4 m/s while adhering to lane boundaries and achieved city driving, obeying traffic laws and responding appropriately to stop signs and traffic lights.

We used a combination of color segmentation, edge detection, and Hough transforms to detect lane boundaries and a PD controller to steer the robot along the path during the Johnson track portion of the challenge. Although we faced issues with our hardware, especially in regards to the chassis of our car and the update speed of our Zed camera, we were able to robustly lane follow and complete all task objectives of track racing.

Our implementation of city driving allowed us to effectively stay on the correct side of the road while driving to each goal point. By incorporating a vector field into the neighbor-finding of the A* algorithm, our robot generated paths to each goal point without violating traffic laws, and could U-turn when necessary. Further modifications to the map allowed for more robustness when dealing with edge cases.

However, there were clear areas for improvement in our solutions. When it came to the Johnson track, hardware issues made tuning the robot difficult; if we had had more time, we would have done more fine-tuning on the PD controller in order to minimize oscillations and complete the track in a shorter amount of time. Similarly, the city driving challenge would have significantly benefited from more time. The center trajectory and pure pursuit follower required additional tuning to drive fully correctly during the city-driving challenge. Additionally, though we had designed a stop detector and traffic light detector, insufficient testing in the Stata basement led to them inconsistently performing during the actual challenge. An obvious next step is to improve the implementations of these two tasks for a more complete city-driving solution.

Ultimately, despite these setbacks, we were able to integrate various previous labs into one system and accomplish the task objectives in a satisfactory manner, with successful completion of both the Johnson track race and city driving, and end our RSS experience on a high note.

5 Lessons Learned

Cynthia Cao Through this lab, I gained a lot of experience dealing with highly frustrating hardware issues. There were many instances over the course of the lab where the problem wasn't in our code, but with the hardware. Having experienced these issues, I would say that I am better prepared for dealing with them in the future. I learned about hough transforms and line detection, which I didn't know about at all going. I also gained a lot more experience in tuning PD controllers. In terms of city driving, it served as an interesting culmination of all the labs that preceded it. I learned more about ROS levels and debugging topics and nodes. Finally, this lab probably required the most collaboration, and really tested my abilities when it comes to effective teamwork in stressful circumstances. I also developed a new hatred for Jetson TX2s.

Trey Gurga This lab was technically challenging but the result of two implementations of autonomous driving made it the highlight of this class. This lab could have had many different implementations so our team took the brainstorming process a lot more seriously than other labs which were more straightforward which gave a lot of insight into teamwork and planning. I learned better strategies to tune the robot's controls and dove much deeper into command line and hardware debugging. Our racecar had many hardware issues which aligned closer with real life and gave great experience with hardware.

Grace Jiang I found this lab interesting because of how open-ended it was, especially for the city driving portion. Though I'm in theory satisfied with our implementation for final race and city driving, many pieces of our ideas failed when put into practice in real life, often times due to hardware issues that were difficult to solve, and other times due to our own fault. The same code that worked one day on the robot could fail the next day or even the next hour, sometimes due to reasons that we still aren't sure about. In the end, we managed to get all components (mostly) working, and I'm really impressed that the team pulled through the difficulties instead of giving up along the way.

Toya Takahashi This lab was particularly challenging with so many parts. We also had hardware issues with the robot and had to replace our robot base. However, I believe this lab needed the most collaboration and teamwork, and while it was the most time-consuming, it was the most interesting challenge to solve. I learned to implement a lane following robot using the hough transform and homography transformation. Since we were coding in the Johnson Track where the robot had no access to the internet, I became more familiar with terminal commands as well. Finally, city driving was a great way to combine everything in this course, and I learned to think more creatively as a group to solve new challenges.

Jonathan Zhang In this lab I probably went through both the most highs and lows out of any of the other labs. Working on path planning for city driving was extremely fulfilling. Being given a non-trivially difficult algorithmic question, coming up with the solution of a vector field and implementing it and watching it work was a really great experience. I feel like I really got the experience of working on a real world robotics project.

Working on the robot however was not the greatest experience for me. I feel like we got things working on sim so well, and so spending 8 hours the night before just randomly not even being able to run our code anymore was a bit crushing. Going through this low probably made me stronger though and prepared me for the next inevitable hardware issue.

Overall though I think this final project really locked in a lot of the concepts we learned in class. Technically wise, I feel like I fully understand all the different ideas we learned of localization, path planning, pure pursuit, computer vision, lidar scans, ros, topics, pid controls and so much more. Communications wise, I feel like I really got a lot better working in a team, especially being comfortable splitting up work.

(Editors: Cynthia Cao, Trey Gurga, Grace Jiang, Toya Takahashi, Jonathan Zhang)