Lab Report: Final Challenge

Team 1

Emmanuel Anteneh
Ben Evans
Vasu Kaker
Johlesa Orm
Michelle Zbizika

6.4200/16.405 (RSS)

May 14, 2024

1 Introduction

Author of section: Ben Evans

The final challenge(1) for RSS came in two parts. The first part was a racing challenge, with the goal being to complete a circuit of a track as fast as possible, while staying within an assigned lane as much as possible. The second half of the challenge was an autonomous driving exercise which involved navigating basement hallways while following standard rules of the road.

The racing challenge took place on the Johnson Athletics Center indoor track. The technical challenges associated with this task were primarily in computer vision, with it being necessary to accurately determine the position of the robot relative to the lane lines on the track. With the lines determined, a pure pursuit controller could then maintain a course between them.

The driving challenge took place in the basement of the Stata center, with orange tape placed to indicate lane markings. Three locations were chosen in advance as goal points to be reached, and on reaching all three, the robot should return to the starting location. In addition, stop signs and traffic lights were placed throughout the course. Computer vision was once again critical in identifying stop signs and red lights, as well as in following the lane line, if that method of navigation was planned. Path planning algorithms were also employed to find solutions for the map and desired locations, and a pure pursuit controller was could then follow the planned trajectory.

2 Technical Approach

2.1 Computer Vision on Car

Author of section: Vasu Kaker

2.1.1 Computer Vision Pipeline High Level

In this section we discuss the high level pipeline between how our car finds a goal point to steer towards from its camera.

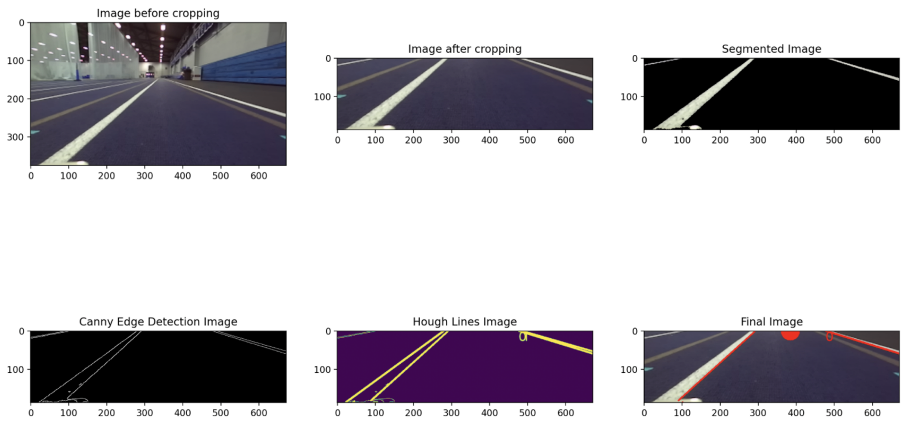


Figure 1: High Level Computer Vision Pipeline

As can be seen in Figure 1, we start with our original image. We crop it to remove extraneous background features, as only the track is necessary. We then segment out the white lines, apply canny edge detection, find the hough lines, and finally choose a suitable pair of lines that denote the lane, averaging their intersection with the boundary of the image to determine the goal point shown as a red dot.
Our methods of suitable choosing a pair of lines is what leads to robustness to extraneous features.
We iterate through all the hough lines, choosing those that are sufficiently large (above 30 pixels in length) and that have an absolute slope value of at least 0.2. This ignores a lot of noisy lines. Then we categorize lines into left and right groups, left including all lines with a negative slope, and right all those lines with a positive slope. Finally, we attempt to find a pair of lines, 1 from left and 1 from right, that are as close as possible to intersecting without actually intersecting.

If we are able to find this pair, then we simply return the midpoint of the two lines' intersection with the upper boundary of the image.

2.1.2 2.1.2

If we cannot return the midpoint of two lines, it is usually due to 1 of 2 adversarial cases, which we account for.

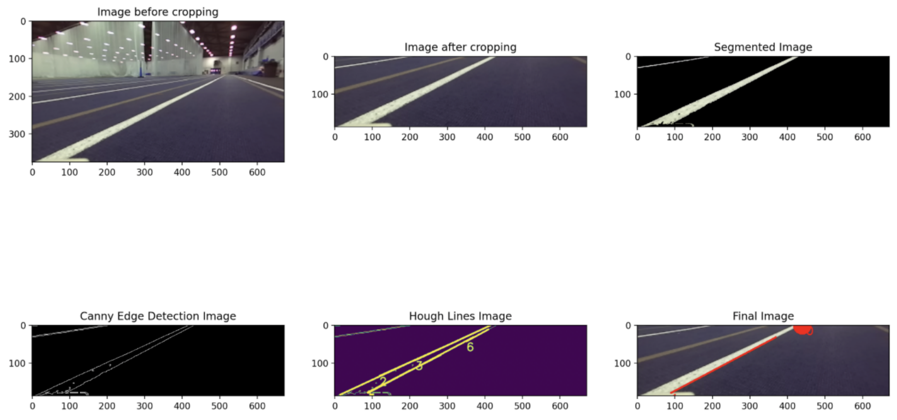


Figure 2: Adversarial Case 1

In Figure 2, we see the first adversarial case. Where only one group, the left group, contains detected lines.
In this case we simply extend the steepest line in the group containing the lines to the boundary, find the intersection, offset by 20 pixels (to the right if left group has lines, and to the left if right group has lines), and return that pixel. This is equivalent to creating an imaginary line through reflection and translation, and finding the new midpoint.
As can be seen in the image, this would cause the car to go right (based on the labelled red point), which would enable it to see more of the right line.
We then review adversarial case 2. This case involves the infamous "non vertical" lines in the Johnson track, which intersect the running track lines at odd angles.
Fortunately, our algorithm of choosing non intersecting pairs of lines means we only choose one line, either from the left or right group. We choose steeper lines in order to favor track lines, which are likely to be steeper in most car orientations than the cross / horizontal lines. We then proceed as in case 1, finding an goal point based on a single line.

2.1.3 2.1.3

Author of section: Vasu Kaker

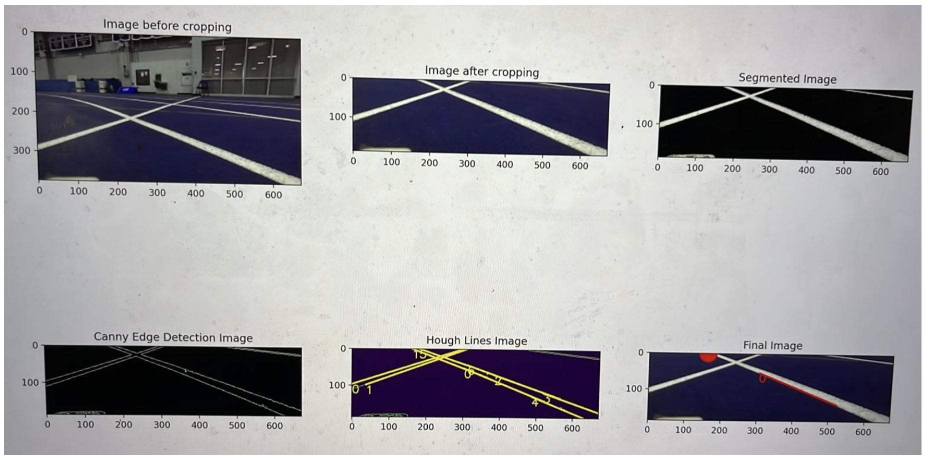


Figure 3: Adversarial Case 2

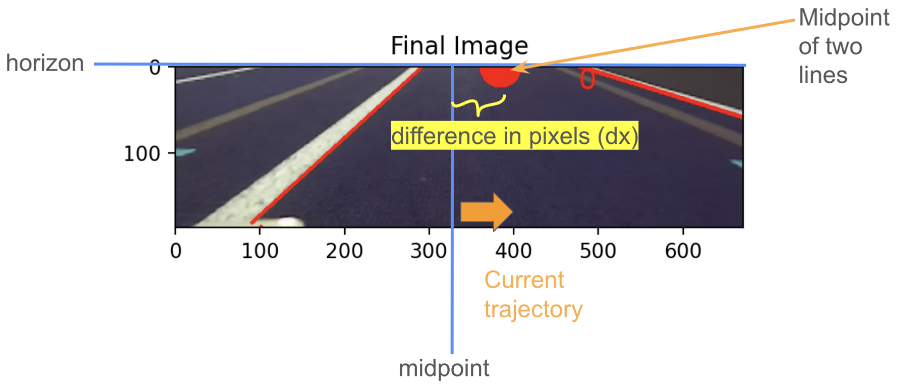


Figure 4: Drive Angle Pt 1

Our next step is to determine a suitable drive angle based on the goal point. As can be seen in Figure 4, we compute the dx, difference between goal point and midpoint of the image, and dy, which is held at a constant of 160 due to the height of the image. From here we compute the angle to drive at, as is visualized in Figure 5, with the following formula.
The angle θ in radians can be calculated as follows:

$$\theta = \arctan\left(\frac{dx}{dy}\right)$$

The final drive angle is then tuned in 2.2. As the lookahead is small, there was no major improvement from using a pure pursuit calculation for drive angle, and we found it was easier to tune this theta value compared to a pure pursuit theta value.

2.2 Angle tuning parameters

The track follower node takes an angle difference from the image, and must publish an actual drive angle. Transforming the angle difference to an actual drive angle proved to be an extremely intricate task with many non-linear relations between the parameters set and the actual performance of the robot. Many experiments were conducted. Below, the reasoning behind the initialization and tuning methodology for the parameters is explained.
Figure 6 outlines the final tuned parameter values and pipeline for finding the drive angle.

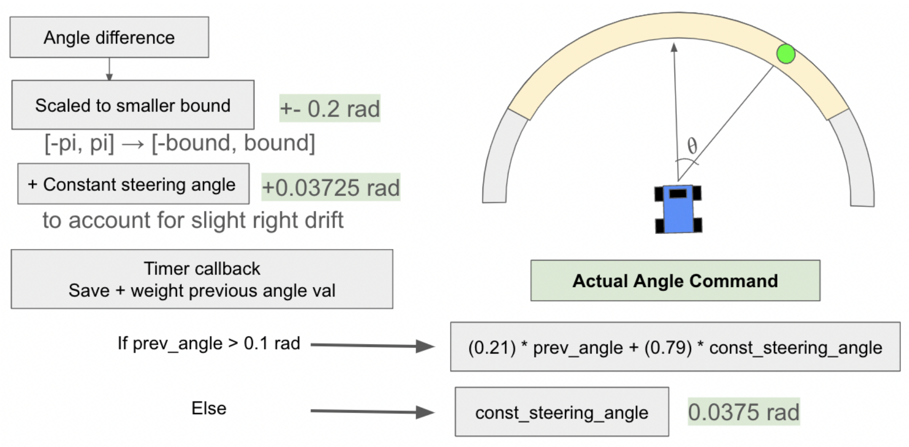
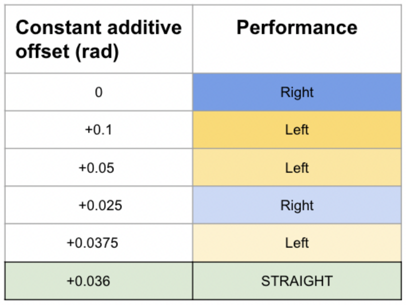

Angle difference
Scaled to smaller bound
+- 0.2 rad
[-pi, pi] → [-bound, bound]
+ Constant steering angle
+0.03725 rad
to account for slight right drift
Timer callback
Save + weight previous angle val
θ
Actual Angle Command
If prev_angle > 0.1 rad
(0.21) * prev_angle + (0.79) * const_steering_angle
Else
const_steering_angle
0.0375 rad

Figure 5: Final Tuned Parameters Overview

2.2.1 Constant Angle to Correct Natural Drift

Author of section: Michelle Zbizika

The robot seemed to naturally drift to the right when angles are published, so experiments were done on the robot to find the best offset to add to the robot steering angle at all times. The data from this tuning is included in Figure 6. Varying constant drive angles were published, and through a binary search of angle offset values, we found which constant drive angle value of 0.036 radians would cause the robot to go straight.



Constant additive offset (rad)	Performance
0	Right
+0.1	Left
+0.05	Left
+0.025	Right
+0.0375	Left
+0.036	STRAIGHT

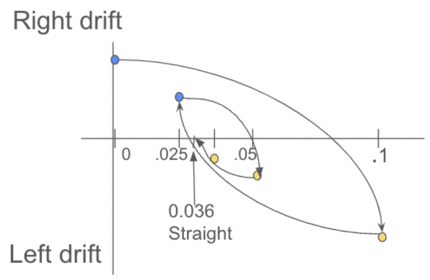


Figure 6: Constant Angle Offset Tuning Data

2.2.2 Angle bounds

Author of section: Michelle Zbizika

The angle found from the arctan(dx, dy) of the pixels was too severe to publish directly onto the robot, as it caused severe oscillations. At first, we directly clipped the angle, but the published angle would either be the constant angle or the upper and lower bounds of the clipping. This was because the natural range of the arctan is $[-\pi, \pi]$. Mapping this range $[-\pi, \pi]$ onto a new range [-bound, bound] using a numpy linear interpolation function yielded significantly better results, as the angles that were published became more fine-grained. We found the best bound was 0.2 rad.

2.2.3 Intermediate Publishing and Previous Angle Weighting

Author of section: Michelle Zbizika

The current trajectory and desired trajectory is only found when there is an image callback. The image callback does not publish often enough for the robot to accurately follow the curves and respond to changes in the line; when no intermediate drive commands are published, the robot either continues the previous command or stops. To mitigate this we have a timer callback that publishes drive commands at 15 hertz (more frequently than the image callback).
At first the timer callback publishes a default drive command at the constant steering angle, causing the robot to drive straight. This worked well at lower speeds but would cause the robot to drive straight off the track when the robot

is at high speed at a sharp turn. Thus, the previous drive angle command of the image callback is saved as the parameter prev angle. When the timer callback is set to the prev angle, there's oscillations that increase in severity and causes a lot of infractions. This is because every angle change is amplified since the car follows that command for longer.
To dampen oscillations caused by the timer callback publisher, a threshold was implemented: if the previous drive angle is above a certain threshold, publish the previous drive angle command, otherwise publish the straight line drive angle. This was to prevent smaller angles from being amplified, especially on straight line sections of the track where small angle changes only need to be executed briefly. This helped decrease oscillations for straight line sections but was still causing oscillations in turning sections. Thus the average of the previous and constant angle was published in the timer callback. This helped our performance but still had oscillations. Ultimately the best weighting found was (0.21)*prev angle + (0.79)*constant steering angle and the best threshold was 0.1 rad.
Below is a highly abbreviated overview of the track follower code, which handles the transformation from the angle difference in the image, and publishes the drive command.

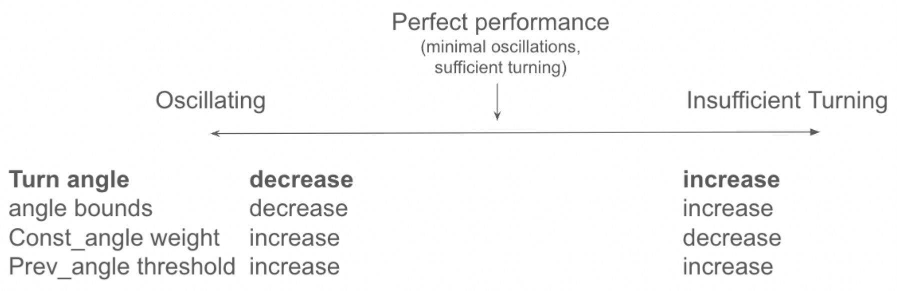
Listing 1: Track Follower Code (abbreviated)

```
class TrackFollowerController(Node):
    def __init__(self):
        super().__init__("track_follower")

        timer_period = 1/15 #seconds (15 hertz)
        self.timer = self.create_timer(timer_period, self.timer_callback)

        bound = 0.2
        const_steering_angle = 0.036


    def timer_callback(self):
        if abs(prev_steer_angle - constant_steering_angle) > 0.1:
            steering_angle = (0.79*prev_steer_angle + 0.21*constant_steering_ang
        else:
            steering_angle = self.constant_steering_angle

        publish(steering_angle)

    def image_callback(self):
        angle_diff = -np.arctan2(delta_x, delta_y)
        steering_angle = np.interp(angle_diff, (-np.pi, np.pi), (-bound, bound))
        steering_angle += self.const_steering_angle
        self.prev_angle = steering_angle
    publish(steering_angle)
```

2.2.4 Performance Evaluation and Tuning

Author of section: Michelle Zbizika

Tuning the parameters outlined above posed a significant challenge. The performance of the robot was evaluated quantitatively and qualitatively in several sections based on what the desired change in angles are: when it needs to goes from straight to turning (first half of turn), turning to straight (second half of turn), and straightaway sections. To simplify our observations, the general performance of the robot is conceptualized to be on an axis from oscillating too much, to not turning enough (Figure 7). Theoretically the best performance lies somewhere in the middle. To tune the parameters efficiently, we thought about general relationships between our parameters and their effect on oscillations/turning, and used the following heuristics.
Here are a few scenarios that happened, and our thought process when tuning the parameters.
If the robot struggles to go from straight to turning, we know it is not turning enough, so we increase the angle bounds, and/or decrease weighting of the previous turn, and/or decrease the threshold to start considering previous drive angle.
If the robot is oscillating in all the sections we know it is more likely previous angle weighting issue.
An interesting scenario is when there are confounding outcomes; like if the robot doesn't turn enough when going from straight to turning, but is oscillating when it goes from turning to straight. In this situation we might have needed decrease weighting of the previous turn, and decrease the threshold to start considering previous drive angle.



Turn angle	**decrease**	**increase**
angle bounds	decrease	increase
Const_angle weight	increase	decrease
Prev_angle threshold	increase	increase

Figure 7: Observed results and Tuning parameter heuristics

In general, if the robot is oscillating too much, that indicates that we need to decrease the published angles: by decreasing bounds, increasing the weight of the constant angle, and/or increasing the threshold of the previous turn. If the robot is not turning enough, that indicates that the angles published need to increase: by increasing bounds, increasing weight of the previous angle, and/or decreasing the threshold for the previous turn. At times, these parameters could be tuned independently, while other times multiple parameters were considered at once, which was difficult to conceptualize and figure out which ones to tune and in which direction. It was incredibly difficult to tune 3 different parameters since they all only affect the turn angle.

A week before race day, the optimal parameters for driving at 2.5 m/s were found, but our robot was not updated to go at 4 m/s until the day of race day. As a result, all the parameter tuning at 4 m/s was done at race day; quantitative data was noted at the time to help tune on the spot, but not much was recorded. Since we had very few opportunities to actually tune and evaluate the performance, many parameters were tuned in tandem using the mental heuristics outlined in Figure 7. Despite these challenges, we still yielded incredible results. Through careful observation, thinking, and luck, we were able to find good parameters. The final run of the robot on lane 1 was 47 seconds with 0 infractions.

2.3 Path Planning and Stop Controls

2.3.1 Path Planning

Author of section: Emmanuel Anteneh and Ben Evans

For path planning, we used the A* (A-Star) algorithm in order to optimize a path to a goal point while avoiding obstacles, then use particle filter localization and Pure Pursuit in order to follow the optimized path. To ensure that the path planning algorithm avoided driving in the other lane, we made sure to plot the lane line and make it an obstacle in our occupancy grid. We also used obstacle dilation in order to ensure the robot maintained sufficient clearance from walls and corners.

Lane lines were set as obstacles using a Bresenham-based supercover line algorithm. Using the algorithm, we were able to determine every pixel on each line segment from the lane trajectories, and then set each of these pixel values to obstructed on our map. In addition to the lane lines, we also inserted our own obstacle lines in various parts of the map, to introduce obstacles that would block the robot from going into areas outside of the city bounds.

The graph which was constructed for the A* search downsampled the pixel grid from the map in order to achieve a reasonably fast search time. This graph accounted for the dilated obstacles as well as the lane line. The amount of downsampling was set by a parameter in order to optimize the search time versus the degree to which we were able to dilate obstacles for additional clearance. A* ran using the Euclidian distance from each point to the relevant goal point as a consistent heuristic function h. After A* found a path, a simple smoothing algorithm was applied which eliminated unnecessary intermediate points.

A* SEARCH:

```
def a_star(start, goal):
    init list open_nodes with starting node $s$, $\mathbf{f}(s) = 0$, parent($s$) = $None$
    init list closed_nodes as empty list
    while open_nodes not empty:
        pop node $n$ with lowest $\mathbf{f}(n)$ from open_nodes
        if $n$ == goal:
            return parents($n$) to trace path back to start
        else:
            for node in adjacent_to($n$) and node not in closed_nodes
            add node to open_nodes with f(node) and parent(node) = $n$

def f(node, start, goal):
    g(node, start) + h(node, goal)

def g(node, start):
    cost to reach node from start

def h(node, goal):
    estimated cost from node to goal
```

Trajectory segments were planned individually between each point selected on the map in rviz. On completion they were each immediately added to a list of paths. We were then able to iterate through the list and publish a new trajectory as we completed each respective segment. The first segment was published as soon as its planning was completed so that the robot could begin driving while finishing path planning for the other segments.

2.3.2 Pure Pursuit

Author of section: Ben Evans

In order to smoothly follow the published trajectories, we implemented a Pure Pursuit controller. The piece-wise trajectory points returned by the path planner were further subdivided into segments of approximately 0.1 meters. From these sub-segments, the one nearest the robot would be found as a starting point. The algorithm would iterate forward in the list of points until it found the one which was closest to the desired look-ahead distance. This new point would then be used as a target point for calculation of a steering angle.

The steering angle formula is standard for pure pursuit combined with the bicycle model and Ackermann steering for vehicle dynamics.(2) The result is an output of a steering angle δ which indicates the degree of deflection for the front wheels of the car. Other variables include the wheelbase L, look-ahead distance L_1, and the angle η calculated as the difference between the vehicle heading and the heading to the target point. The pure pursuit steering angle formula for our vehicle is then as follows.

$$\delta = tan^{-1} \frac{2L * sin(\eta)}{L_1} \tag{1}$$

2.3.3 Stop Sign and Traffic Light Controls

Author of section: Johlesa Orm and Michelle Zbizika

We used color segmentation to identify the stop sign and traffic lights from the camera. For stop sign detection, we used the provided staff solution. For the traffic lights, we had tuned the color segmentation to identify and create bounding boxes around each of the traffic light colors, skewed towards lower saturation value (since the lights were almost white in a dark setting), and higher hue and value settings. From pixel location information gathered by the bounding box, we then used homography transformation in the x, y, and z coordinates in order to find the real-world location of the lights. We similarly use homography transformation to find the center of the bounding box returned from the stop sign detector. The real-world locations are then used to determine whether the car is within the required stopping distance for the stop sign and stop light.

When we are in the required distance for the stop sign and a stop sign, we implement a timed stop. For the traffic lights, however, we check to see the light status: a red light corresponded to sending a speed of 0.0 m/s and any other light meant that the default path following speed would be sent.

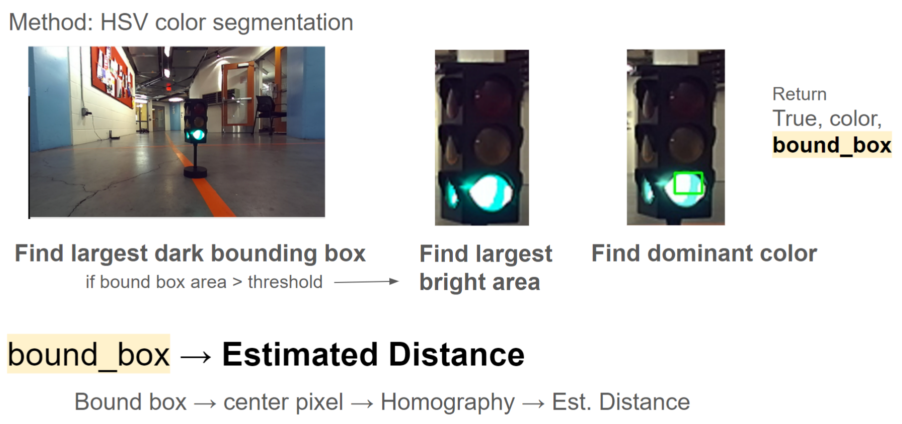

Method: HSV color segmentation
Return
True, color,
bound_box
Find largest dark bounding box
if bound box area > threshold
Find largest
bright area
Find dominant color
bound_box → Estimated Distance
Bound box → center pixel → Homography → Est. Distance

Figure 8: Traffic Light CV Pipeline

2.3.4 State Logic

Author of section: Ben Evans

The control logic for our robot's city driving program was built around a pure pursuit controller, with a master control loop running at 20Hz. Prior to calculating and sending the control values a number of checks would be performed, including checks for stoppages, distance to goal, and available navigation data. Information from these checks would be used to determine the appropriate action.

Stoppages include those for safety stops to avoid obstacles, stop signs, traffic lights, and no path/end of path states. Of these, safety stops were the only one to send a drive command directly, with the rest being called through the master control loop. The safety stop function was used to account for pedestrians as well, with a timeout allowing the vehicle to proceed at a slow pace and steer around obstacles that appear and do not quickly move out of our path. The master control loop would continuously check flags set by the stop sign and traffic light controllers for the stop signs and traffic lights respectively.

A stop sign flag would increment a counter which had to exceed a minimum threshold to prevent erroneous stops. The counter would continue to increase after a stop command had been sent until it exceeded a maximum threshold which then roughly indicated the amount of time to stop at the intersection. After this maximum threshold was reached, the vehicle would proceed to drive forward traveling more slowly until the stop sign flag was unset, at which point the counter would also be reset to zero.

A traffic light flag set for a red light would likewise increment a counter until a minimum threshold value was exceeded. At that point a stop command would be issued and the counter would stop increasing. Upon the traffic light flag being unset, the counter would decrement with each control loop and a command to resume driving would be issued when the counter reached zero. This ensured that an individual camera frame which failed to identify a red light would not prematurely allow the resumption of driving.

Distance to goal was calculated with each control loop as the Euclidean distance between the robot's current position, and the end (goal) point of its current path. This distance was used to determine when the vehicle was approaching the goal, when the goal had been reached, and if progress toward the goal was occurring.

Upon the distance to goal falling below a threshold, a 'goal nearby' flag would be set. This flag would cause the controller to modify the speed of its drive commands proportional to the distance. On the distance to goal falling below a much smaller threshold, a 'goal reached' flag would be set. The goal reached flag would trigger a stop command as well as incrementing a counter. Once this counter exceeded a threshold, the goal was considered to have been achieved and the next path was requested from the trajectory planner. This threshold was set both to prevent erroneous indications of having reached the goal, as well as to provide a means of controlling the amount of time spent stopped at the goal.

While the intent was to use localization with a LIDAR scanner and particle filter, it was recognized that this may not always be sufficiently reliable. To account for this, the control loop would check if a reliably identifiable lane line was available to follow. Provisions were included to use a line follower adapted from the track racing challenge for this method of navigation. An offset from the identified line would be used as a target point for which a steering angle would be calculated by the pure pursuit formula. While this alternate navigation logic was implemented, it was ultimately never tested due to the adequate performance of localization, as well as due to time constraints.

3 Experimental Evaluation

3.1 Simulation

3.1.1 Trajectory Planning and Following

Author of section: Ben Evans

A* was consistently able to find paths across the longest possible distances of the map within just a few seconds. Trajectories such as those for the city driving challenge were typically planned in less than one second with the initially selected parameters. As obstacle dilation was increased to provide more clearance around corners, the available paths became significantly narrower. It then became necessary to reduce the amount of downsampling used in generating the graph for the A* search. Though this reduced downsampling resulted in larger graph sizes and longer search times, A* was still able to find solution trajectories much faster than the robot was able to complete them.

Pure pursuit worked extremely well in simulation, with look-ahead distances of 1.5 to 2 meters providing excellent stability and virtually no oscillation. This look-ahead distance was eventually reduced in simulation to be nearer to 1 meter in order to reduce the likelihood of hitting corners in tight turns. This reduction, in combination with object dilation, was sufficient to prevent collisions with corners. The reduction did result in limited oscillation in some cases, but these oscillations were of small amplitude and were quickly damped.

3.1.2 Stop Sign Controls

Author of section: Johlesa Orm

The stop sign controller worked based on a timing mechanism. When the stop sign is found and within a certain distance as indicated by the homography transform function, then it will order the car to stop for about 4 seconds before continuing on its path. In simulation, we used the traffic cone as our stop sign and set a stopping distance of 0.1 meters. The car was able to successfully stop within the threshold distance and continue driving after the elasped buffer time.

3.2 Real World Testing

Author of section: Johlesa Orm

Trajectory planning and following: At first, the car would skate by the sides of the wall much too closely to progress along the path. Turning and driving was difficult since the wheels would frequently get caught on the wall. At the start of the path in Stata, the camera would also get caught in the supports underneath a long bench. In order to remedy this, we significantly dilated the wall obstacles on our map in order to force the path planning algorithm to create a path far enough away from the benches. A dilation factor of at least 50 was needed in order to clear the bench, when before, we had a factor of 10.

After this major change, the car was able to make its way to each of the shells, however, it sometimes crossed into other lanes on its way. We observed that it crossed three times, and noticed that it corresponded to the new path being planned once it reached another shell. Although we ran out of time to fully fix this issue, we suspect that the errors with lane-crossing are largely due to issues with localization. We noticed that in the hallway with lots of metal and a clear plastic tarp entryway, the localization was very poor since the lidar scans may not have been accurate enough. If we had a chance to do this again, we would make sure to tune the sampling parameters for localization of the car around that section, and any other difficult sections of the final path.

Stop sign controls: Unfortunately, we were unable to test the stop sign and traffic light controls on the real robot in parallel with the trajectory planning and following algorithms. However, when our old robot was still working, we were able to see that it successfully identified the stop sign and traffic lights and sent the correct drive speed messages when we echoed the topics. This indicates that if we were to fully integrate these two components, it would be likely that they would have performed as planned.

4 Conclusion

Section Author: Johlesa Orm

Overall, we were able to use many of the ideas from our previous labs on this final challenge. Some of these ideas were the Pure Pursuit algorithm, BFS and A* for path planning, color segmentation, and homography transformation. This challenge also required us to learn and implement new concepts, such as Hough Line transformations for the racetrack.

Our racetrack CV pipeline, involving Canny edge detection and Hough Line transformations allowed us to successfully identify the main lanes that we needed to use in order to travel down the track. Our classification of left and right lanes, and creating a substitute lines in the case that only one lane line would be found ensured that the robot would not incorrectly drift towards another lane or be thrown off by additional intersecting lines on the track. Finally, using linear interpolation to update the car's angle and setting the car at a constant heading in order to correct natural drift led to the success of our lane finding and following algorithm.

For Luigi's Mansion, we used the A* path finding algorithm, Pure Pursuit with the planned trajectory, and color segementation with homography transformation to locate the stop sign and traffic lights. By using intermittent path planning each time we got to a shell location, we were able to speed up path planning on the robot. Although the path following on the actual robot was imperfect, we were able to collect our shells. We have much to improve on by implementing our safety and stop controls.

This final challenge proved to be difficult and involved many topics from throughout the semester. However, we were able to showcase our understanding and celebrate some successes.

5 Lessons Learned

Michelle Zbizika: It's important to start early! When debugging, make sure not to get stuck in "sunk cost" thinking: when a certain approach isn't working, it might be a different component that is causing an issue, or a different solution might be required. I learned that I am much more motivated in a competitive environment, but at the same time everyone and every team moves at their own pace, and don't get too down about yourself if your performance isn't immediately amazing.
Ben Evans: Not only is it important to start early, it's important to get code running on the hardware early. Working out all the bugs in transitioning from simulation to hardware was a continual challenge, not helped in our case by the number of hardware failures which occurred. That said, this was an enormously fun challenge which was a great way to integrate what we've learned during the semester.

Emmanuel Anteneh:

Johlesa Orm: Through the various hardware and coding issues that we came across, I learned that it's important to keep my head up and try everything possible to get something to work! A lot of times, I'm convinced that something specific is the issue but through testing and reading error messages, I often learn that what's causing something to fail can be something entirely different than what I had originally thought. It was scary not having our robot working some time before the competition, but I'm so glad that there were teams kind enough to let us borrow their batteries and even a whole robot for the second challenge, when our robot broke down. My teammates did an insane job coding and tuning for this challenge, and I'm so happy that we were able to have something working on competition day.

Vasu Kaker:

References

[1] RSS_Course_Staff, "Final challenge 2024," Available at https://github.com/mit-rss/final$_{c}hallenge$2024(2024/05/13).

[2] R. C. Coulter *et al.*, *Implementation of the pure pursuit path tracking algorithm.* Carnegie Mellon University, The Robotics Institute, 1992.