Final Race Challenge

Tareq Dandachi, Mikey Peña, Daniel Kuang
Steven Goldy, Tom Benavides

February 6, 2022

Table of Contents

1 Introduction

Author: Steven Goldy

Editors: Daniel Kuang

The goal of Robotic Science and systems (RSS, 6.141/16.405) is to develop an autonomous robot to compete in a race. The original plan was to develop and test software to be used on the RACECAR hardware platform and race in the basement of the Stata Center or the Indoor track. Due to the unfortunate circumstances surrounding COVID-19 the competition needed to be moved online. To accommodate this shift, the fantastic group of RSS teacher's assistants created a racing simulation using Lincoln Labs' Tesse framework. Tesse was developed in the well known Unity 3D game engine and offered a wide array of features. A full suite of sensors and cameras complete with noise and realistic driving dynamics made this simulated competition more interesting. While the code was a little too intense for some machines to run, the photo-realistic graphics made the competition interesting. The races were conducted on a closed city scene complete with lane lines, poles and guardrails. To equalize the hardware being used, all cases were run on the MIT Lincoln Lab Supercloud cluster.

The final challenge was divided into two parts: a timed race course and an obstacle avoidance course. The objective of both challenges was to complete the course in the fastest time possible while minimizing collisions. Head-to-head racing was out of the scope of the class so each component was assigned a scoring function to determine the top competitors. In both parts we utilized skills and algorithms developed in previous labs. These modules include, a safety controller, wall following, Monte Carlo Localization, path planning and computer vision. Both challenges did not use all of the components and not in the same way. The goal of the race circuit was to develop a robot that could find itself in an environment then efficiently navigate to a known destination. This exercise tests many difficult facets of robotics, their coordination, and gives insight into potential advances in the field. The goal of the obstacle avoidance portion was to develop a robot that could navigate safely through a crowded unknown environment.

Both of these tasks are important to the field of robotics and autonomous vehicles because they prove capability for agents to operate in urban environments. Using on board sensors, a rudimentary knowledge of the environment, and a great deal of tuning, cars can be developed to traverse a city with no collisions.

2 Technical Overview

Authors: Steven Goldy

Editors: Mikey Penña

As this is a final project, we made the design decision to adapt our existing code to the new environment, tune parameters to the specific races and optimize for speed. The idea is each of the modular labs would fit into the new final system and create an autonomous system. We approached the two parts of the challenge with a different combination of modules.

For the race course portion we planned to use our particle filter localization method from Lab 5 to provide an up to date position of the robot. With this information, we intended to have the robot to calculate a path to an intermediate milestone or the finish line using the A* path planning algorithm from Lab 6. Once it had a trajectory, the robot would use the pure pursuit controller also from Lab 6. To avoid collisions and trigger a restart in calculations, we planned to run the safely controller from Lab 3.

But, after examining the Tesse simulator and receiving recommendations from the teaching staff, we made the design decision to forego path planning and simply use a predefined trajectory. We could do this because the race was conducted on a predefined closed course. This defined trajectory enabled us to further optimize for speed and computational efficiency.

For the obstacle avoidance portion we decided to leverage the open-endedness of the challenge and the sensors on the simulated car to create an innovative machine vision solution. This was a difficult design decision because of a lack of familiarity with using camera data and technical difficulties with receiving the data from the simulator. In the end we were able to process the segmentation and depth camera data, apply a birds eye view filter and then use a modified version of the wall follower algorithm. The controller worked by finding the middle of the road based on the color data from the segmentation camera and using that as a "wall" for the robot to track.

3 Technical Approach - Race Course

Author: Mikey Peña, Daniel Kuang, Tom Benavides

Editors: Steven Goldy

The following sub-sections outline the details of sensors, control algorithms, and integration methods used for the final race.

3.1 Interoceptive Sensors

In the end, our race course algorithm only depended on the odometry provided by our Monte Carlo localization algorithm. The Monte Carlo localization algorithm projects the car into many particles each with their own different position and orientation. Then using each particle, it compares its hypothetical LIDAR beams with the actual LIDAR beams collected from the car's LIDAR sensors. The Monte Carlo localization algorithm will then keep the most probable odometry.

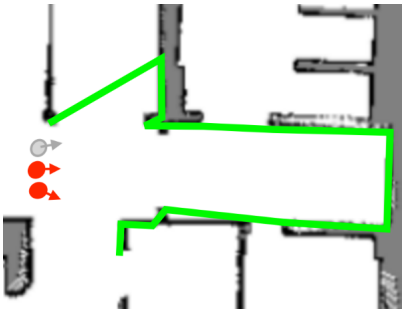


Figure 1: The Monte Carlo Localization (MCL) algorithm in action. Notice how the first particle (one possible odometry of the car) is eliminated because its lidar beams (the green shape) is ruled as unlikely. The middle particle will be ruled as most probable, hence it'll be kept as the most probable odometry.

Our implementation only depended on the provided x and y coordinates, but not the orientation. Since we encountered difficulties in receiving the correct orientation of our car due to map offsets, we decided to calculate our car's orientation with three points: the closest point on the trajectory, the goal point toward which the car will approach and the global position of the car relative to the map frame.

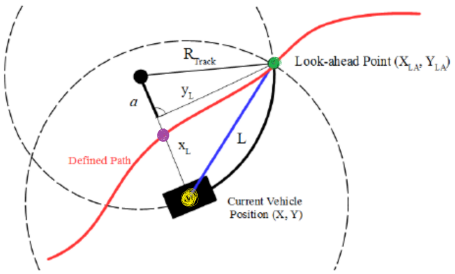


Figure 2: A visualization of the closest point on the trajectory (marked purple), the goal point (marked green) and the global position of the car (marked yellow).

The orientation we calculated with the closest point, the goal point and the car's coordinate was the angle between the vector from the car to the goal point and the vector from the goal to the closest point. The reason we want this angle is the pure pursuit algorithm. The pure pursuit algorithm needs the car's orientation to calculate the perpendicular distance from the car to the goal point:

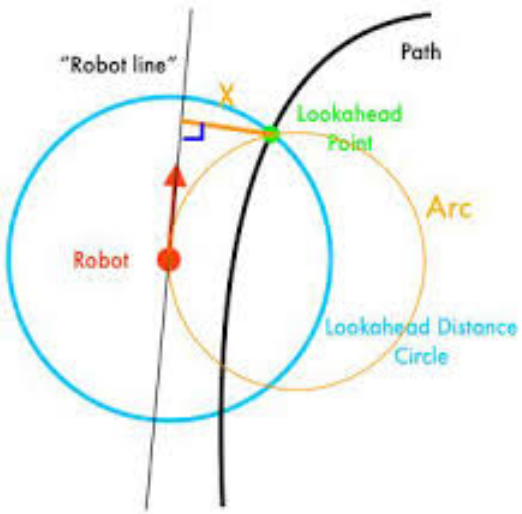


Figure 3: A visualization of the perpendicular distance of the car to the goal point. The goal point is colored green and the perpendicular distance is labeled as X colored in orange

The perpendicular distance is necessary for the curvature equation, which is used to calculate the steering angle of the car to remain close to the trajectory. $L_{car-close}$ is the distance from the car to the closest point on the trajectory and $L_{goal-car}$ is the distance from the goal to the car.

$$\delta = \tan^{-1}(L_{goal-car}/R)$$

$$\gamma = \frac{2 * L_{car-close}}{L^2_{goal-car}}$$

$$R = \frac{1}{\gamma}$$

Our trajectory is composed of line segments concatenated together. Hence, if we visualize the triangle with the goal point, closest point and car's coordinate as vertices, we will see that it's a right triangle:

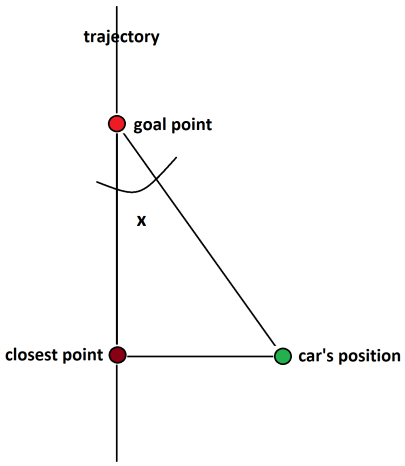


Figure 3.5: A visualization of the closest point, the goal point and the car's position connected as a triangle. Note the closest point is coined as "closest" because the closest point on the trajectory from the car's position is defined to be the perpendicular line connecting the segment to the car's position

We simply calculate the perpendicular distance with the euclidean distance formula: $P_{dist} = \sqrt{(carX - closeX)^2 + (carY - closeY)^2}$ where (carX, carY) is the 2D coordinate of the car and (closeX, closeY) is the 2D coordinate of the closest point. The problem is we don't know the sign of the distance. The sign of the distance is necessary to determine which side the car should turn around.

To determine which side the car should turn around, we take the sign of the cross product of the vector goal-close and the vector goal-car. We applied a right handed convention to define the desired angle x in figure 3.5. So depending on the direction of the turn we have positive $\hat{z}$ angle vector or a negative $\hat{z}$ angle vector.

If the $\hat{z}$ angle vector is positive, the car is to the right of the trajectory as indicated in figure 3.5. This is true no matter where the trajectory is positioned. To get the car to steer left (back to the trajectory), its angle needs to be negative. As of now, the sign of the cross product of the goal-close and goal-car vectors is positive (since the thumb is pointing out of the paper). Hence, we multiply the sign of the cross product of goal-close vector and goal-car vector by -1 to get the desired sign. A symmetric argument applies if the car is to the left of the trajectory.

Now we have the correct steering angle magnitude and sign to command the car to steer to its trajectory correctly.

3.2 Race Course Algorithm

3.2.1 Path Planning

The intent was to have our A* algorithm to predict the shortest path to the goal. The A* algorithm also needs to account for the fact that the car needs to travel the whole course.

Figure 1: Race Map and Track

Note the green point is the start and blue point is the goal. The car must follow the red path before reaching its goal.

If we ran A* at the start of the race, the shortest path would be the purple path, which is illegal. An alternative to using A* is creating the trajectory with trajectory_builder.py from path_planning lab (lab 6). Then, once the car loads on the simulator, it will take the given path and follow it.

An advantage of selectively choosing a path is how we avoid complications from avoiding shortcuts potentially found by A*. In addition, we save time from the potential overhead from running A*.

Thus, we decided to forgo A* and worked on fine-tuning our pure pursuit algorithm to follow our built path.

3.2.2 Pure Pursuit

Recall the objective of pure pursuit is to calculate the steering angle for the car to reach its goal or lookahead point along an arc (figure 2). If we prevent the car from recalculating its new lookahead point, the car will smoothly reach to its current lookahead point along an arc. If we have the lookahead point to keep updating until the lookahead point is the endpoint of a trajectory, the car will essentially be following its provided path. The nature of this algorithm is akin to the carrot-and-stick principle.

Due to the map offset of the simulator, our car was unable to receive its orientation. We resolved this issue under the above section "Interoceptive Sensors." However, there was another issue, and it's regarding the fact that there are overlapping line segments at certain parts of the race.

Figure 1: Race Map and Track

When the car reaches the region in the blue circle, its closest point will be ambiguous. The car is programmed to calculate the closest line segment and the closest point. However, due to noise, the car can skip or repeat sections of the course depending on what the car believes as its closest point. During the race, since we did not notice this flaw, our car either skipped the loop or stayed on the loop (the loop is circled with a brown circle).

Fortunately, we noticed this error on our second try of the race course. Hence, our implemented solution is to have the car keep track of the current line segment it is on. In addition, we restrict the car to seeing only three line segments ahead of the line segment it is on now. Hence, to the car, it never

notices an overlap on any line segment during its completion of its provided trajectory.

3.2.3 PID Controller

The final step of the pure pursuit algorithm was to include a PID controller in the steering angle and speed calculations, so that the race car's path-following behavior could be tuned in simulation. The car's speed only required an additional derivative control, whereas we included a full PID algorithm to tune the car's steering angle, which was more impactful in keeping the race car on the path and completing the course.

The parameters for the PID algorithms were determined one component at a time, tuning the values with a binary search. We first tuned the proportional component, then the derivative component, and finally the integral component (when applicable for the steering angle calculation). If the car's performance became worse after tuning one component, we would then take a step back and re-tune the previous components to account for the down-stream changes.

Moreover, our algorithm's speed around corners also played a role in tuning our PID controller. The PID controller naturally handles curves by using the proportional gain to keep the car on track. The derivative gain corrects this a bit by not allowing for large jumps in the turning angle. One thing we noticed while testing is that the speed at which we took the turn had a large impact. This intuitively makes sense, as if you are moving faster, then the algorithm has less chances to get the right turning angle and thus would over or under correct. Thus, when calculating speed, we took into account the curvature of the upcoming section of the path. Our speed was then calculated as:

$$V = min(max_v, K/abs(curvature)$$

where V is the final speed, max_v is our pre-determined maximum speed around the track on straights and K is a constant that we chose to tune the parameter. In practice a K of 1-2 seemed to work best with our final value of 1.8.

4 Fast Obstacle Avoidance

Author: Tareq El Dandachi

Editors: Daniel Kuang

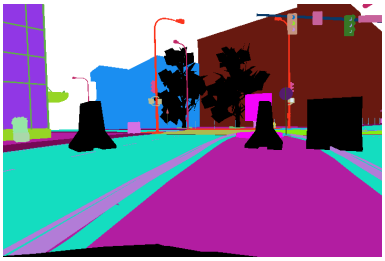
4.1 Exteroceptive Sensors

The Fast Obstacle Avoidance algorithm uses 3 main sensors: the segmentation camera, the depth camera and the LIDAR. Both cameras are placed on the front bumper of the car; They provide front facing, low-angle POV shots with a wide angle of view as depicted in figure 4.

The segmentation camera emulates the output of a high accuracy convolutional neural network (CNN) that segments the objects in the scene with $\approx 100\%$ accuracy and colors them based on what the objects are. Figure 4(a) shows the output where the road has a turquoise color, the pavement is purple, the concrete barriers are black, etc. The result from the CNN is a BGR color space image (Blue-Green-Red matrix) similar to images you take with your phone, where every objects is uniform in color and represents an object as defined in a csv file that defines the color mapping.

The depth camera returns a grey image, i.e. every coordinate has one value associated with it as opposed to BGR or HSVs' three values. The color values represent the distance between the camera and the object, the darker an object is the closer the camera is (with the exception of the sky, which isn't considered an object). This can be seen in figure 4(b), where the building far away from camera is much brighter than the obstacles and the road near the camera.

The LIDAR is also used to calculate distances from the objects but in a 1D fashion. This is used for two main components of the algorithm, localization as explained in the race course algorithm and a simpler distance triangulation for the car that takes into account objects out of the view of the depth camera.



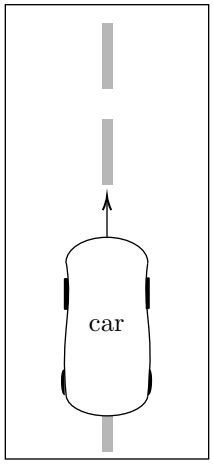
(a) Segmentation Camera

(b) Depth Camera

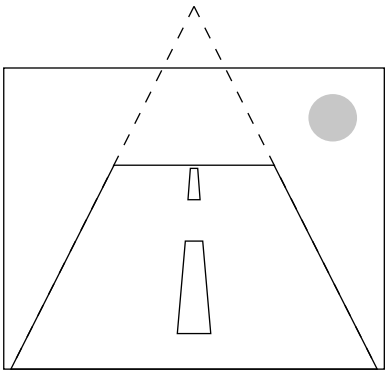
Figure 4: Sample output from the cameras in different locations

4.2 Processing Sensor Data

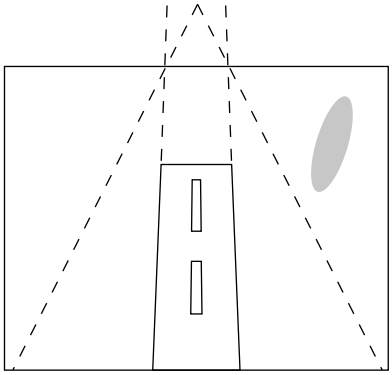
The output from the cameras morphs the geometry of the image due to its POV nature, two parallel lines pointing roughly in the direction of the car appear to intersect, we need to apply a transformation that preserves the parallel nature of these lines before processing. Figure 5 showcases the camera output (b) and the desired output (c).



(a) Top View



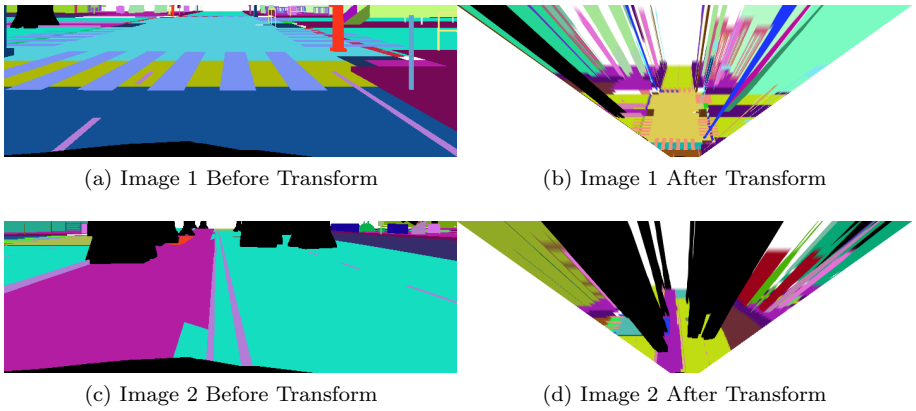
(b) Camera Output



(c) Camera Output After Transform

Figure 5: Application of the homography transform on an image to fix the perspective of the camera on the road

Since our primary focus is the road in front of the car and not the sky, we remove the top half of the image to remove the extra redundant information and then apply a perspective shift using the homogrpahy transform as defined in the previous paragraph. Examples of the transform can be seen in figure 6.



(a) Image 1 Before Transform

(b) Image 1 After Transform

(c) Image 2 Before Transform

(d) Image 2 After Transform

Figure 6: Examples of the homography transform applied to the output of the segmentation camera (the recoloring can be ignored, its due to the conversion between BGR to RGB)

After transforming the segmentation camera output, we create 4 separate sub-images:

1. Floor Map, a small portion of the transformed image (10 pixels by 10 pixels) at its bottom that is used to detect what surface the car is on and any changes in surfaces, this helps tell apart road intersection, pavements and the road itself. It can also be used to detect super close collisions.
2. Front View, this portion has the same height as the image but the width of the homography transform at the bottom of the image. This is used for detecting straight on collisions and helps with identifying if a path straight ahead is clear from obstacles.
3. Left View, everything to the left of the vertical center-line of the image, it is used for detecting obstacles on the left of the car and deciding whether deflecting to the left is a safe choice.
4. Right View, everything to the right of the vertical center-line of the image, it has the same purpose as the left view.

Due to the highly discrete nature of segmentation, and the fact that the number of objects, n that can be identified by the segmentation camera is $n \ll 256^3$ and also $n \ll 256$, then we know that forcing the image that has 3 bins of 256 values (a BGR image) have only 1 bin (a gray image) would not cause us to lose any information stored in the segmentation but reduce the size of the matrices by at least a factor of 3, making the algorithm faster and less memory intensive.

The LIDAR data is also partitioned into 3 separate point clouds, they represent the front, left and right of the car. Their construction is identical to the segmentation image reconstruction, however, they contain no intersection of readings and are 1 dimensional, since LIDAR is 1 dimensional.

4.3 The Fast Obstacle Avoidance Algorithm

At this point, all the sensor data is "meaningful", as in it has been segmented into separate useful components that can be added up to control the car. The only shared part between the Fast Obstacle Avoidance algorithm and Race Course algorithm is the localization module, which is used for locating the car on the map and defining the robots position with respect to a start point and an end point.

The Obstacle Avoidance algorithm has two types of points, local and global, i.e. there is a local start point, local end point, global start point and a global end point. The local ones are used for traversing a single path on the map, such as a straight line connecting two intersecting roads on a turn-point, while the global points represent the actual start point of the car on a map and where it wants to reach. The localization module is used whenever turn-points on the map are reached, these turn-points correspond to intersections on the map. Whenever the car enters an intersection, the floor map segmentation can detect an interface and can change the local start and end point of the algorithm so it can traverse a segment on the map.

The local start and end points, $\vec{p}_s$ and $\vec{p}_e$ respectively, are joined by a line l and the distance of the car's center from the line l corresponds to an error value e_p. This error value is supposed to correct the angle of the wheels the robot is using to get from $\vec{p}_s$ to $\vec{p}_e$. In an ideal case where no obstacles are present, a controller minimizing the e_p value will force the robot to move in a direction parallel to l with no deflections at all.

$$e_p = \kappa_p \frac{|(\vec{p}_s - \vec{p}_e) \times (\vec{p}_s - \vec{p}_r)|}{|(\vec{p}_s - \vec{p}_e)|}$$

To account for deflections based on objects surrounding the car, we have more error parameters that go into our total error, e_t. The first of which is the LIDAR error which is reserved for super close-by objects that might be in blind spots that the segmentation camera might not see. The LIDAR error is defined as $e_l = \sum_f \left(\kappa_f \sum_x \frac{1}{x}\right)$, where x represents the closest LIDAR measurements in a certain frame f from the LIDAR segmentation.

The most important error terms that correspond to the actual deflection of the car are swerve error term, e_s, and the fixed deflection term, e_d. The swerve

error term is defined as the difference in the number of pixels representing obstacles in the left and right segment of the image, i.e.

$$e_s = \kappa_s \mu_s \left(\sum_{c \in \text{left}} (1 \iff c \text{ is an obstacle}) - \sum_{d \in \text{right}} (1 \iff d \text{ is an obstacle}) \right)$$

This term is important since it accounts for the difference in (1) the number of obstacles on the left and right side of the car and (2) how close the obstacles are, since a closer obstacle stretches vertically more which means more pixels represent it. This means the right and left components of e_s are each proportional to the number of obstacles on that side and how close they are to the car. The κ_s is the "swerve" coefficient, which dictates how much the error should scale with the difference in pixels and μ_s is a normalizing coefficient that is solely dependent on the resolution of the input image, since a higher resolution image can give a much larger swerve and the system to become over responsive.

The e_d term is a term that deflects the car by a constant amount κ_d based on whether the average proximity of the car to other objects is past a certain threshold γ_d.

$$e_d = \begin{cases} \kappa_d & \text{if avg}(\mathbf{x} \in \text{on the left of the car}) > \gamma_d. \\ -\kappa_d & \text{if avg}(\mathbf{x} \in \text{on the right of the car}) > \gamma_d. \\ 0 & \text{otherwise} \end{cases}$$

We wanted to maximize the velocity of the car, while keeping it safe from collisions. To do that we defined our speed v such that

$$v = v_q \min \begin{cases} 1 \\ \sqrt{\epsilon_f |\sum_x L|} & \text{where } x \in \text{the closest 10 points} \\ & \text{from the front LIDAR measurements} \end{cases}$$

$$v_q = \begin{cases} v_f \sqrt{1 - \epsilon_p |e_p|} & \text{if } 1 - \epsilon_p |e_p| > \epsilon_n \\ v_f \mu_n \epsilon_n & \text{otherwise} \end{cases}$$

Where ϵ_f is a constant that scales the front lidar measurements such that $\epsilon_f |\sum_x L| < 1$, $\epsilon_p \ll 1$ is a constant that scales the path offset error, this constant helps overcome over steering into walls, $0 < \epsilon_n < 1$ is a cutoff for when the path offset becomes large enough to need to be accounted for and $\mu_n > 1$ is a scaling coefficient that controls the effect of ϵ_n on velocity.

The fixed velocity v_f used is set to 30ms^{-1} so that the car can drive at high speeds and let the lowering of the speed be controlled by the other terms st. the force applied to achieve a v that is $20 < v < 40$ at all times.

4.4 The PID Controller

Now that we have an error signal $e_t = e_p + e_l + e_s + e_d$ that accounts for the deflection from the goal and the amount of obstacles the car can collide with, we need a controller that accounts for this error. We settled on using a PID controller, with a similar construction to that of the final race) that changes the angle of the wheels α as follows:

$$\alpha = k_p e_t + k_i \int e_t dt + k_d \frac{de_t}{dt}$$

The only remaining task is tuning the whole system (algorithm + controller). We tuned the algorithm by isolating separate subsystems, creating test environments that suit them and tuning them alone on a P (proportional) controller only, and modifying larger parameters when joining these subsystems. For instance, to tune the path offset components, we had a path for it to follow with no obstacles. For tuning the LIDAR errors, we turned off all components and spawned the objects close to the car, out of the range of view of the cameras and made sure no collisions happen. For tuning swerve errors, we had a course filled with obstacles away from the car and modified the swerve such that it can come close to the barriers without hitting walls while still giving it an open window for more accurate control by another subsystem.

After tuning all the systems, we plugged them all into a PID controller and tuned it using the heuristics defined in the Zeigler-Nichols method. This is done by tuning the P controller separately, followed by the integral part then the derivative part.

5 Experimental Evaluation

Author: Daniel Kuang

Editors: Mikey Peña

5.1 Race Course

Much of the time used for race course is spent on fine-tuning the PID controller parameters for both the steering angle and speed of the car. The metric for improvement is how much the car oscillates along its provided path and whether or not it can complete the whole course.

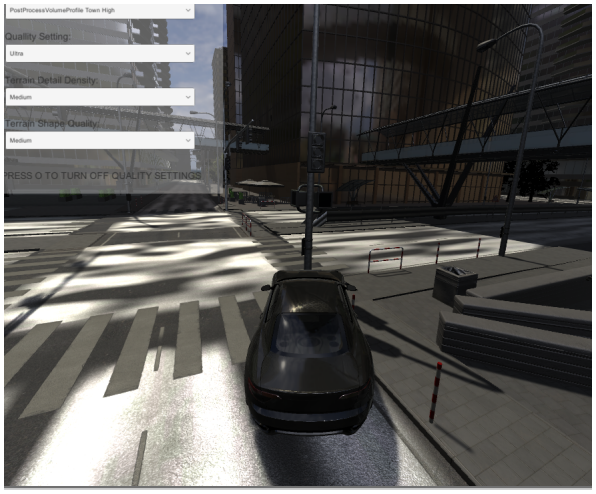


Figure 7: Visualization of the car's inability to complete the course. If this occurs, we return to our PID controllers and continue fine-tuning

Our method of fine-tuning is as follows:

1. Fine-tune the parameter for the proportional component of PID with binary search
2. Fine-tune the parameter for the derivative component of PID with binary search
3. Fine-tune the parameter for the integral component of PID with binary search

If the results worsen, we take one step back before advancing onto the next step. In the end, our optimal parameters are

$$k_p = 0.095; k_d = 2.5; k_i = 0.0019$$

for the steering angle PID controller and

$$k_p = 1.0; k_d = 0.001$$

for the speed PID controller.

5.2 Fast Obstacle Avoidance

Improvements in the fast obstacle avoidance are similar because the fast obstacle avoidance algorithm also uses a PID controller. In the end, the optimal PID constants for the car's steering angle are

$$k_p = 0.7; k_i = 0.01; k_d = 0.02$$

6 Conclusion

Author: Steven Goldy

Editors: Tom Benavides

While we were not able to complete scoring runs of either challenge on the live race stream, our race car still demonstrated impressive capabilities and later completed both racing challenges. There are still a few shortcomings that we can address, but we nonetheless developed a functional autonomous racing algorithm capable of completing the course at fast speeds, even if it is inconsistent. The computer vision obstacle avoidance algorithm was very innovative and effective, and could be improved even more with further tuning. The localization and pure pursuit methods in the final race were not perfect but still ensured a safe trajectory most of the time. We have learned a great deal from the final challenge as well as the class as a whole.

6.1 Future Work

The race is over but there are still ways for our system to be improved. The next realm of challenges is to make the robot robust to new environments. It is possible that we will refine our localization methods and maybe integrate them with Google Cartographer to use Simultaneous Localization and Mapping (SLAM). We would also like to integrate our CV obstacle avoidance work with the pure pursuit algorithm to be able to more safely navigate novel and changing environments. Lastly, the pure pursuit controller could be refined to support greater speed and precision in a race scenario.

6.2 Lessons Learned

Tareq

This project was an opportunity to put everything we learned in RSS together, from both the communication and the technical aspects of RSS. The team dynamics have strengthened a lot and I feel confident in saying I was able to blindly trust my teammates throughout these two weeks and I wasn't wrong. We ended up setting a lot of fires and I think most of the credit can be attributed to trusting my teammates. This project also involved a lot of creativity in terms of applying what we learned in RSS, we ended up using material covered in all of the labs and had fun coming up with new creative algorithms.

Mikey

On the technical side I learned how to integrate older code into a new setting and adapt choices I made for one situation for another. On the communications side, I learned how to use comments from other people and older code to quickly understand what is going on and what might be wrong.

Daniel

On the technical side, I learned how to flexibly apply PID controllers to achieve a desired state in a different context. Before, I thought PID controllers are used alone, but they complement well with other algorithms like pure pursuit. On the communication side, I learned the importance of maintaining contact despite how dire the situations seemed during the race.

Goldy

On the communications side I learned the importance of taking notes while you work. It is much easier to collect your thoughts to present information when you have notes recorded than trying to delve into your memory for what you did three weeks ago. On the technical side I learned the importance of asking the right questions. I learned a great deal about machine vision by googling the right questions and discussing with Tareq.

Tom

On the technical side, I learned the value of tackling challenges from different perspectives, and how small increases of algorithmic complexity can cause large pay offs in performance. On the communication side I better appreciated commenting code to improve working between different people.

7 Acknowledgements

Thank you to the technical and communication teaching staff - your time and expertise were much appreciated and we learned a great deal.

Thank you to the fantastic team of TA's that answered our questions and helped when things were falling apart

Special thanks to Marcus for taking the time to be our TA while still developing the whole Tesse-ROS system