Track Racing and City Driving on Racecar

Andy Li
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
andyli27@mit.edu

Antonio Avila
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
antonio3@mit.edu

Harmanpreet Kaur
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
hkaur@mit.edu

Mia Tian
MIT Department of Aeronautics and Astronautics
Cambridge, USA
miatian@mit.edu

Natalie Huang
MIT Department of Electrical Engineering and Computer Science
Cambridge, USA
huangn@mit.edu

1 Introduction

Mia Tian

The goal of the final challenge is to pull together all of our previous labs and knowledge to perform two tasks: race on a track and drive in a city environment.

The track racing involved image processing to detect the track lanes and also lane following. We used our previous implementation of pure pursuit for lane following, and we coded up the image processing pipeline as well as integrated all components to achieve the specifications. The challenge was graded on the speed of completing a lap around the Johnson track (200m) while staying in its lane.

The city driving component involved driving to three locations and following traffic laws. Specifically, it required driving on the right side of the road, stopping at traffic lights and stop signs, and stopping for people during crosswalks. We implemented an innovative path planning and following system which uses our previous A* implementation and ensures right-side driving. We updated our safety controller to take less risks for the cross-walk requirement. We used machine learning techniques to identify and stop at traffic lights and stop signs. Lastly, we tied all of this logic together with a state machine which controls the overall flow.

2 Technical Approach

2.1 Track Racing

Antonio Avila

Racing on the MIT Johnson track requires the car to stay inside the lane it starts in while also running at full speed.

2.1.1 Lane Detection: In order to stay in our lane, we need to know the lines that define our lane. To do this we use some basic computer vision with the camera mounted to the front of the car. Our vision pipeline is as follows.

Upon receiving an image, we crop out from the top to the horizon, and 10/32 of the image from the bottom. We found that this crop anticipated turns well while also not over anticipating. Then we mask the image for white pixels and find the edges in this mask. We then dilate the image of the edges to be able to detect lines easier. We then send the image through a Hough transform to get lines in the image, We then filter the lines for lines on the right and lines on the left with a slope greater than an absolute value of 0.25 since we found that, by testing empirically, lines that are horizontal in the image will have a slope less than this. We then average the left lines and the right lines to get our estimate of our lane lines. This assumes that we are always somewhere within our lane.

2.1.2 Lane Following: To follow these lane lines we get the angle bisector of the two lane lines that passes through the bottom of the image. This is possible because of the idea of a vanishing point. Another consequence of a vanishing point is that the angle bisector that passes through the bottom of the image is the line that the racecar should follow since it is theoretically the line equidistant and parallel to the lane lines. We instead of transforming this line to the world frame, to avoid too much error in the camera homography matrix, we choose an offset of 30 pixels. This offset is the distance down, in the image, from the intersection point of the lane lines. We then calculate the point in the image that lies on the bisector at this offset position in the image. We choose this point as our look-ahead point for a pure pursuit controller. Since we cannot follow a point in the image, we transform this point with our camera homography and follow that point.

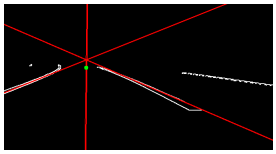

Fig. 1: The final output of our vision pipeline. The green point is our lookahead point. The line which it is on is the angle bisector. The other lines are the lane lines.

2.1.3 Assumptions and Risks: This approach assumes that the car will be in the same lane at all times and never leave it, however it is a good controller to keep the car in the lane it is in. That said, we have no method of recovery in the case that the car does completely leave its lane.

2.2 City Driving

Natalie Huang

We had 3 main modules in our city driving solution: the stop detector, which handles stopping due to stop signs and traffic lights; the safety controller, which avoids collision with pedestrians and walls; and the path planning/following module, which handled navigating the car to each of the goal points while staying on the right side of the lane.

2.2.1 Stop Detector: Harmanpreet Kaur

The stop detector operates on its own small state machine, as depicted in Algorithm 1 below.

Algorithm 1 Stop Detector State Machine

Setup:
$state \leftarrow$ DETECT

Image Callback:
$image \leftarrow$ received from car
if $state$ == DETECT **then**
  if StopSignDetected($image$) **then**
    $state \leftarrow$ WAIT
    $t_0 \leftarrow$ CurrentTime()
  else if TrafficLightDetected($image$) **then**
    $command \leftarrow$ STOP
  end if
else if $state$ == WAIT **then**
  if CurrentTime() $- t_0 >$ 5s **then**
    $state \leftarrow$ COOLDOWN
    $t_1 \leftarrow$ CurrentTime()
  else
    $command \leftarrow$ STOP
  end if
else if $state$ == COOLDOWN **then**
  if CurrentTime() $- t_1 >$ 5s **then**
    $state \leftarrow$ DETECT
  end if
end if

To analyze images from the racecar's camera, a machine learning model by YOLO that is pre-trained to detect both stop signs and traffic lights was used. The model takes in an image and returns a list of objects detected, along with the confidence for each object and a bounding box indicating where in the image the object was located. The stop detector modules analyzes each image from the camera through this model. Whenever a stop sign or traffic light is detected with high confidence (over 0.7), the area of the bounding box is checked to determine if the object is close enough to the robot for the stop detector to issue a stop command. For both stop signs and stop lights, it was determined through trial and error that an area of 1000 pixels^2 was a sufficient threshold for when either object was 1 meter away.

If the object detected is a stop sign, a stop command is issued to the car for the next 5 seconds. Then, a cool down period of 5 seconds is instilled, where no stop commands can be issued so the car is able to bypass the stop sign.

If the object is a traffic light, the image is analyzed to determined whether it is a red light. If so, a stop command is issued to the car until a red light is not longer detected.

Traffic Light Color Detection

To analyze the color of the traffic light, the original image containing a traffic light is cropped to the bounding box given by the YOLO detection model. Then, the image is split in half horizontally and the top half is accessed for the percentage of red pixels. This is done by creating a mask covering any pixels that would be considered in the set of Red-ish Pixels as defined below. The percentage of pixels covered by the mask is calculated and if it is over a certain threshold (such as 50%), then we can consider the traffic light to currently be a red light.

$$\text{Red-ish Pixels} ::= \{(r,g,b) \in \text{AllPixels} \mid (200,0,0) \leq (r,g,b) \leq (255,75,75)\}$$

These boundary points were qualitatively assessed by measuring the color detected by the racecar when the red light was on and when the red light was turned off. This led to omitting dark-red pixels (which was the color detected when the red light was off) and favoring bright red pixels (as detected when the red light was on).

Fig. 2: The green line is the default trajectory we manually built. It loops around the entirety of the Stata basement to ensure we can get close to any possible goal points.

2.2.2 Path Planning and Following: Natalie Huang

Our path planning strategy was to follow a pre-planned default trajectory around the entirety of the Stata Basement. We then used our A* algorithm implemented in the previous lab to plan shorter paths to goal points that were close by. We chose this strategy because our current A* implementation is not able plan paths that only stay on the right side of the

center line. Therefore, we restricted A* use to only plan from start and end points that were definitely on the same side and close to each other. An image of the path is shown in Fig. 2

Pre-Planned Trajectory

We used the staff-provided trajectory builder to lay out a path consisting of 43 points that looped around the Stata basement while staying on the right side at all times. We tested this path with our lab 6 path follower in simulation and real life to make sure it could accurately follow it. This path is what the racecar will follow in DEFAULT mode.

Planning

As the car follows the default path, it continuously checks

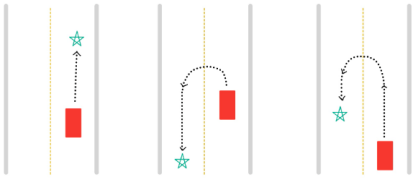

Fig. 3: The intended behavior of when the goal point is at 3 different possible positions.

for these cases. The cases are visualized in Fig. 3

1) The next goal point is in front of the car's position and on the same side of the road. This means the car doesn't have to turn and it continues to drive straight.
2) The next goal point is behind the current car position. In this case, since we always have to be on the right side, the car must u-turn regardless of the side of the basement the goal point is located. Once the car realizes this, it will switch to the U-TURN state.
3) The next goal point is in front of the car's position but on the other side of the road. In this case, we continue driving forward. Once the car drives far enough forward that it is in front of the goal point, case 1 will cause the car to uturn onto the correct side of the road.

Finally, if the car is in Case 1 and within 3.5 meters of the car, we use astar to plan a short trajectory from the car's pose to the goal point and the car transitions into PLANNED state. In this state, the car ignores the default trajectory and only follows the points published by astar. Once the car reaches the goal point, it waits for 5 seconds and returns to DEFAULT state, repeating the process until all goal points have been visited.

Path Planning Calculations

To implement the state machine described above, we needed methods to calculate which side of the center line a point was on. We did this by taking the cross product of two vectors. The first vector is from the projection of the car onto the centerline, and the second vector is the actual centerline. The sign of the cross product tells us if the car is currently on the right(positive) or left(negative). This is show in Fig. 4

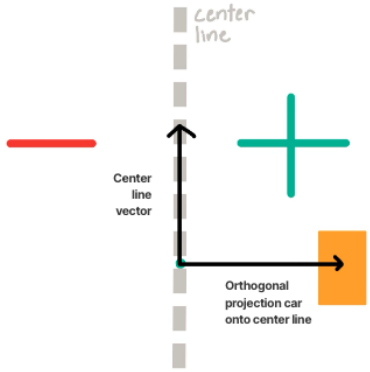

Fig. 4: A visualization of the cross product method we used to determine which side of a line a point is on. A positive cross product corresponds to the right side, while a negative one corresponds to the left.

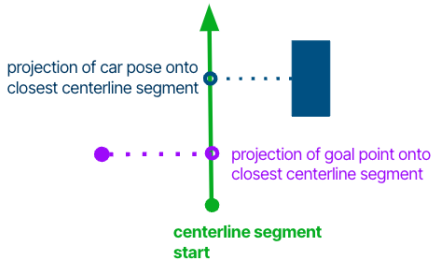

Fig. 5: A visualization of how to determine if the goal point is ahead of or behind the car, using the projection of the goal point and the car's pose onto the centerline.

Additionally, we needed a way of calculating if a car was in front of or behind a goal point. Again, we took advantage of the fact vectors forming the centerline were provided to us. Using some calculations from lab 6, we first calculated which segment on the centerline was closest to the car's current pose as well as the goal point. We then checked for 3 cases:

1) The goal point is closest to a centerline segment that is earlier in the path than the segment closest to the car's current pose. In this case, we know the goal point must be behind the car.
2) On the other hand, if the goal point is closest to a centerline segment that is later in the path than the segment closest to the car's current pose, we know the goal point must be in front the car.
3) Finally, if the car and goal point are closest to the same centerline segment, we calculate the projection of both points onto that segment and compare them. If the projection of the car is ahead of the goal point projection, then the car is in front, and vice versa. A visualization of this case is shown in Fig. 5.

These two functions allowed us to implement our planner described above, since at every point in time we could determine which side the car and goal point was on, and also whether the car was ahead of or behind the goal point

.

U-turns

To implement u-turns, we first found the projection of the car's current pose onto the centerline. Then, we had the car steer towards it with pure pursuit. We did this to ensure that the car would definitely go towards the centerline and cross it. We tried a more naive approach where the car just turns at the max steering angle for a set number of seconds, but found that it would either turn too far and hit something, or not enough and it wouldn't end up on the opposite side of the line. As mentioned above, we can always calculate which side of the centerline the car is on, so once we detect that the car has actually crossed the line it returns back to the DEFAULT state and continues following the trajectory.

Path Following

Our path follower was not a separate ROS node, but just a part of the planner. The planner determined both the points that the car should be following as well as the drive command, using pure pursuit, that the car needed to execute. This was mostly due to time constraints. The actual follower was the same one we used in lab 6, the naive controller. It simply looks through the list of points in the path for one that is not visited and not closer than the lookahead distance. One additional feature we implemented was that whenever the car made a uturn, the points on the path that it had visited would change. To handle this, whenever a turn was made, we reprocessed all points in the default trajectory and marked them as visited if they were behind the car's new position, and marked them as unvisited if they were in front of the car.

2.2.3 Safety Controller: Andy Li

Given the time constraints of the project, we wanted to reuse old code and combine functionality as much as possible. The safety controller is the best example. It is critical to the overall performance of the car in two ways. First, it prevents the car from collisions, as was its original goal. However, given the city driving challenge, we needed to ensure that it would make the car stop for pedestrians and facilitate multi-point U-turns in a tight hallway environment without complicated planning or conditional angle checks.

We modified our original safety controller to be more aggressive, as any false positives would result in unnecessary stops or backups that are detrimental to the speed-based assessment criteria of the challenge, and we simply needed to ensure that the car would not hit something directly in front of it. We also wanted to make it computationally minimal, allowing more processing power for the path planning/following and computer vision modules.

To do this, we convert each Lidar point into polar coordinates about the center of the turning radius determined through standard Ackermann kinematics. Points within a certain arclength and radius from the racecar are marked. When there are more than 3 marked particles (to prevent false positives), the safety controller returns that there is an obstacle. The math is outlined in Algorithm 2.

Algorithm 2 Safety Controller

$threshold \leftarrow 3$
$\Delta_R \leftarrow 0.3$
$D_stop \leftarrow 0.5$
$R \leftarrow wheelbase/\tan(steering\_angle)$
$c \leftarrow (0, R \cdot sign(steering\_angle))$
$count \leftarrow 0$
for $p \in$lidar_data$[\left\lfloor \frac{-\pi/4-\theta_{min}}{\theta_{incr}} \right\rfloor : \left\lceil \frac{\pi/4-\theta_{min}}{\theta_{incr}} \right\rceil]$ **do**
  $R_p \leftarrow \sqrt{(p[0]-c[0])^2 + (p[1]-c[1])^2}$
  $angle_p \leftarrow \arctan(|p[0]-c[0]|/|p[1]-c[1]|)$
  $arcl_p \leftarrow |angle_p * R|$
  if $R - \Delta_R \leq R_p \leq R + \Delta_RAND0 \leq arcl_p \leq D_stop$ **then**
    $count \leftarrow count + 1$
  end if
end for
return $count > threshold$

2.2.4 Control Flow: Andy Li

To combine all the different modules, we used a single node that subscribed to the Ackermann command published by the path following node as well as the Boolean outputs of the safety controller and stop controller nodes. It stores the past 50 outputs of the safety controller. If more than 75% of the outputs are positive, the state machine assumes that the racecar is in front of a wall, and reverses at $1.5m/s$ and $-0.1rad$ turn angle until the outputs change enough.

Otherwise, if either the current safety controller or stop detector output is positive, it stops the racecar with a zero velocity Ackermann command. If none of these conditions are met, it publishes the desired Ackermann command returned by the path following node so that the racecar makes progress along the path when no obstacles are present. Because this node is publishing at a constant 100 Hz, this also alleviates the issue of inconsistent driving command publishing speeds, as it publishes the last returned Ackermann command continuously at a consistent rate. The safety controller topic callback and timed control flow callback are outlined in Algorithms 3 and 4, respectively.

$N \leftarrow 50$
$ind \leftarrow 0$
safety_controller $\leftarrow \{0\} * 50$

Algorithm 3 Safety Controller Callback

safety_controller$[ind] \leftarrow$ int(sc_msg.data)
$ind \leftarrow (ind + 1)\%N$

Algorithm 4 Control Flow Callback

if $\sum_{i-N-1}^{i}$safety_controller$[ind] \geq 0.75N$ **then**
  $cmd \leftarrow$ ackermann_cmd$(-1.5, -0.1)$
else if last_sc_msg.data OR last_stop_msg.data **then**
  $cmd \leftarrow$ ackermann_cmd$(0, 0)$
else
  $cmd \leftarrow$ last_pf_cmd
end if
publish_drive_cmd(cmd)

3 Experimental Evaluation

3.1 Track Racing

Antonio Avila

The racecar's track racing was evaluated in both simulation and real life.

3.1.1 Simulation Testing: While there was no good way to test the car's vision system in simulation. We were able to use an array of images taken from the car at different positions on the track to tune our vision pipeline. The pure pursuit controller was tested in previous labs and was known to work well. The camera homography was completely redone for maximal accuracy in this lab and was plotted to make sure it seemed true to life.

3.1.2 Testing on Real Racecar: Antonio Avila, Andy Li

A video of our racecar following a lane on the track can be found here. Our score for the racetrack was 97, with a completion time of 53 seconds corresponding to an average of approximately 4 m/s (the given limit) and zero lane violations. Further testing confirmed that our average lap was about 53 seconds and about 0.5 short lane violations over 6 laps.

3.2 City Driving

3.2.1 Safety Controller and State Machine: Andy Li

The safety controller worked well during testing, stopping when a person or wall was immediately in front of it, while still allowing for aggressive turning maneuvers close to the wall. The reverse functionality of the state machine worked perfectly, combining with the U-turn path planning to turn the racecar around in a narrow section of the hallway by reversing to the right when a non-transient obstacle was detected. No pedestrians were encountered during the final challenge, and the only manual intervention was due to the path being too close to the wall, as the safety controller prevented any collisions. The integration of the stop detector was also effective, as the robot detected and stopped the adequate amount of time at the stop sign, although stoplight detection did not work. No conflicts between input sources or unexpected behaviors were ever encountered due to the robust logic formulation.

3.2.2 Stop Detector: Harmanpreet Kaur

The stop sign detection worked well during testing, while the traffic light detection preformed less well. The robot was able to recognize both stop signs and traffic lights when stationary at the correct distances (1 meter away). However, when driving the robot at high speeds, the camera struggled to take clear images without lag, which was detrimental to the traffic light recognition. The resolution of the images as well various lighting conditions made it difficult to analyze the current color of the traffic light. As you can see in Fig. 6, this blurry image outputted a bounding box with 0.2 confidence, which is much below our threshold of 0.7. We found that we could not lower the threshold to anywhere near 0.2, since that resulted in false positives, where in our case were worse than false negatives.

Fig. 6: An image taken from the robot, with a red rectangle highlighting its detection of the traffic light. This bounding box was given with a confidence level of 0.2, which is much under our threshold of 0.7.

With this limited hardware, our team had to decide between a faster racecar and moderate traffic light detection. After evaluating the scoring metrics, we found it advantageous to set the speed of our car high enough to likely preform better than the baseline time for the circuit yet low enough that the stop sign recognition would be intact. During the final challenge, this was found true as in our best run, we were able to complete the circuit in 2:07 minutes with stops at all stop signs.

3.2.3 Path Planning and Following: Natalie Huang Most of the testing was done in simulation by feeding the car a random 3 points and seeing if the car would be able to get to them. Our main qualitative metric was the time it took to reach all 3 points. However, this time heavily depended on the location of the points and we had no baseline to compare our time to so it was difficult to evaluate. Qualitatively, we had good results and a video of the planner reaching 3 goal points successfully can be found here. Due to time and space constraints(many other teams testing in the same area), we were not able to trial run a huge variety of points. However, on the day of the actual challenge, we were able to reach all 3 goal points on every trial, and each run was around 2 minutes 10 seconds.

4 Conclusion

Andy Li

Our team was able to combine all of our technical expertise learned throughout this class into writing code that effectively completed two complex challenges. Given the limitations of

the onboard processing power and limited time, we were able to maintain a balance between robust, complete solutions and heuristics that greatly reduced the computation power and testing time required.

For the racetrack, our iteration on previously implemented Pure Pursuit and homography matrix code proved to be the correct choice, as our solution ran flawlessly on the first attempt on Race Day. Using more complex code may have added robustness, for example in the case of large deviation from the starting lane, but creating a fast and accurate controller avoided having to account for this situation in the first place. The relatively simple Hough transform math runs rapidly enough for the car to adjust to curves on the track and maintain the maximum allowed speed.

The city driving challenge also reused much of our previous code, although machine learning for stop sign and stoplight detection was a new addition. A more robust and aggressive iteration of our safety controller proved to be perfect for preventing collisions with either walls or pedestrians. We refined our A-star path planning and Pure Pursuit point-based path following algorithms to ensure that the racecar always stayed on the right side of the road and navigated correctly to the goal points. It also gave us the flexibility to include U-turns as part of the path planning (albeit with assistance from the safe controller) rather than adding a more complex specific control sequence. Our implementation of a pre-generated path except when in close proximity to the goal points minimized our real-time usage of A-star, which is computationally expensive and results in delays while driving. Finally, our Monte Carlo localization was effective and fast enough to be preferable to the TA-given solution, although it likewise struggled in long, homogeneous hallways, a known issue with Lidar-based localization methods. Overall, we successfully completed the course (plus the bonus) on Race Day with only one manual intervention (where our pre-generated path failed) and no stop sign or crosswalk violations, which we consider a success and a testament to our cumulative experience with creating robust autonomous racecar algorithms.

5 Lessons Learned

5.1 Andy Li

I learned the value of having modular code, allowing each team member to work independently on specific parts that played to their strengths. Once each module was independently tested, I combined them with the state machine, which worked first-try, saving us a lot of testing time on the racecar. We also discovered the value of thinking outside-the-box, as our idea to reverse for non-transient safety controller violations allowed us to write a simple yet extremely robust and safe U-turn algorithm.

For communications, I have become a much better team worker. I found that even when not everyone on a team is doing urgent working, just being there physically can be a great support for those that are. Also, we all bonded throughout the semester, which allowed us to quickly resolve conflicts and enjoy our work together/stay dedicated and focused, more valuable than any contract or plan.

5.2 Antonio Avila

This final challenge has taught me many things, but I think most of all I learned to work with what you have and make the most of it. Besides this I have learned more about high-level planning and control of an autonomous robot through the second final challenge.

On the communications side of things I have learned to work well with my team to get things done under a time crunch. I also learned that taking a break with the team can greatly improve the team's morale.

5.3 Harmanpreet Kaur

Technical Lessons
CI Lessons

5.4 Mia Tian

Technical Lessons
CI Lessons

5.5 Natalie Huang

I enjoyed working on the path planning challenges presented by the goal points and centerline restrictions, and I'm proud of our team for working around time restrictions and coming up with an innovative state machine solution after realization astar would be able to handle everything we needed. I also learned how to better modularize parts of a larger challenge, and I think we did well combining all the parts together into a mostly working system.
This challenge definitely required us to communicate more on how to interface and use each of our modules, and understand each other's work more. Staying motivated throughout the entire night before was also difficult but the struggle definitely bonded us, and overall I think we worked very well together.