Near-Future Pose Estimation of an Autonomous Car Using AprilTag

Christopher Kao
University of Pennsylvania
chriskao@seas.upenn.edu
May 1, 2019

Abstract
The goal of this paper is to predict the near-future pose of an autonomous car in a race setting. Near-future refers to up to 1 second in the future. I use the F1/10 autonomous cars platform, with 2 cars. The goal is for the car behind to be able to predict where the car in front will be 1 second in the future, given information about the leading car's pose, which consists of xyz position and orientation. I use the AprilTag system with an RGB USB webcam mounted on the trailing car in order to obtain such information on position and orientation. I describe an algorithm which combines a linear dynamic model and a pure pursuit waypoint-following model in order to predict a car's near-future pose. The algorithm performs well on straightaways but is inaccurate in predicting the car's future pose on turns. The algorithm often predicts that the car will turn left too early.

The Github repository for the code and extensive README can be found here:
www.github.com/mlab-upenn/future_pose_estimator

An overview video of the near-future pose-estimation results can be found here:
https://www.youtube.com/watch?v=uQ1Bp-Bjlzk

Introduction

The F1/10 autonomous racecar is comprised of a Traxxas Rally 1/10th scale RC car with a Jetson TX2 computer, Hokuyo lidar sensor, Vedder Electronic Speed Control (VESC), brushless motor, power board, and battery. Penn's F1/10th team has hosted autonomous races for the past 3 years all over the world, where teams have come from all over to compete in autonomous races for the fastest lap times. All races up to now have been individual car races, meaning that cars run time trials. There have not been any official head-to-head races yet. F1/10th races began in fixed worlds. The best example of this is the 2016 F1/10th race held at Carnegie Mellon University where autonomous cars raced in the hallways. This is called a fixed world because the walls in the corridors do not deform. A nice contrast to a fixed world is a dynamic world, such as a race track set up in an open space. If a car crashes into the walls of the track, then the walls deform. Hence the world is not a fixed world. For purposes of this paper, I assume a fixed world because it makes the research problem easier since I can get better localization results knowing that the map does not change. I also assume that we know the map and the general path that each car will take in this race setting.

Why is the problem of near-future pose estimation important in an autonomous car race setting? There are many good reasons for this. First, in an autonomous race setting, there will be moments where cars want to pass each other. The passing problem can be broken down into a few components. The car behind should first know where the car in front is relative to itself. Then the passing car should try to predict where the car in front will move to in the near future. Next, the passing car should generate a trajectory which takes into account the future trajectory of the leading car such that in the near future the passing car will have passed the other car. Second, near-future pose estimation is important in a race setting because it can be used by the leading car (with a camera looking backwards) to predict how the car behind will try to pass it. For instance, if the leading car sees the car behind inching towards the right, then it may predict that the trailing car will try to pass it on the right side. Third, near-future pose estimation has benefits in life-sized real-world cars as well. On the roads, in highways and urban environments, it will be helpful to predict where other cars will be. This will be especially useful when cars cannot yet communicate with each other, so that the autonomous car can predict where another autonomous car or a human driver will be in the near-future. Fourth, near-future pose estimation is useful not just in self-driving cars, but also in other types of vehicles, such as autonomous boats, drones, and planes. Imagine a future where drones are delivering objects between drones. It would be helpful to know where the other drone will be so that the delivering drone can execute such a delivery mid-air to the other drone.

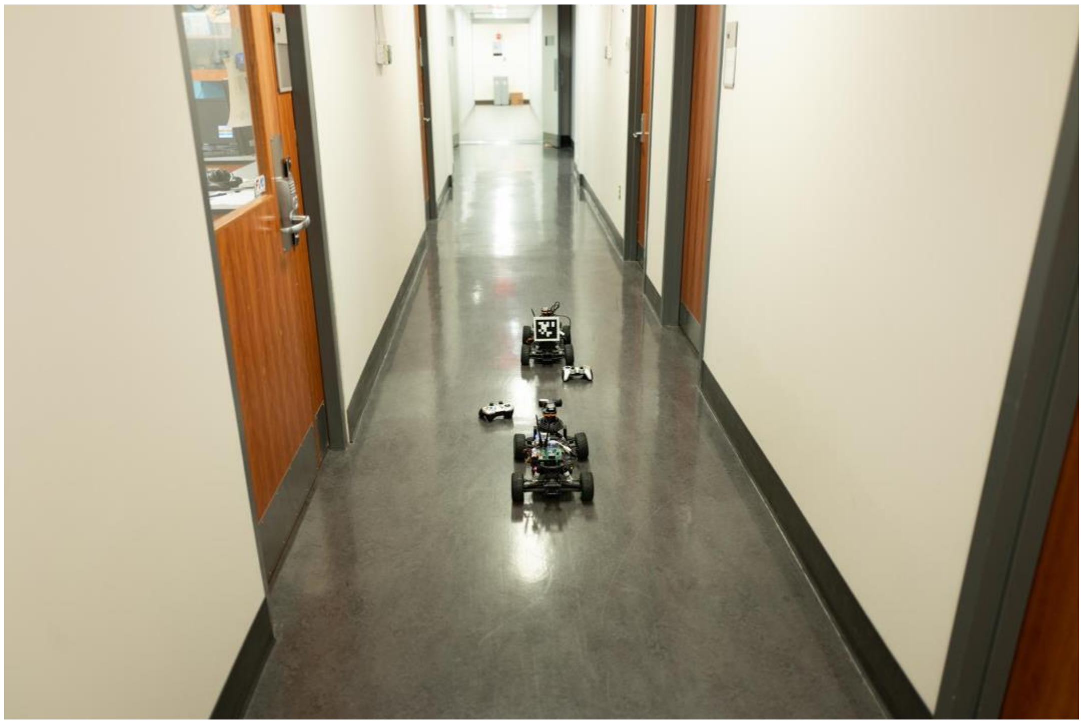


Figure 1. High-level overview of 2 cars. The car behind has an RGB camera and the car in front has an AprilTag mounted on its back.

Figure 1 shows a high-level overview of 2 F1/10th cars positioned in the 2nd floor of Levine Hall in the University of Pennsylvania School of Engineering and Applied Sciences quadrangle. Here the cars go around the rectangular shaped hallway, around the mLab, in a counter-clockwise fashion. The car behind (the passing car) has a Logitech C910 RGB camera mounted on top of its lidar sensor. The car in front has an April Tag mounted on its rear, with a known size.

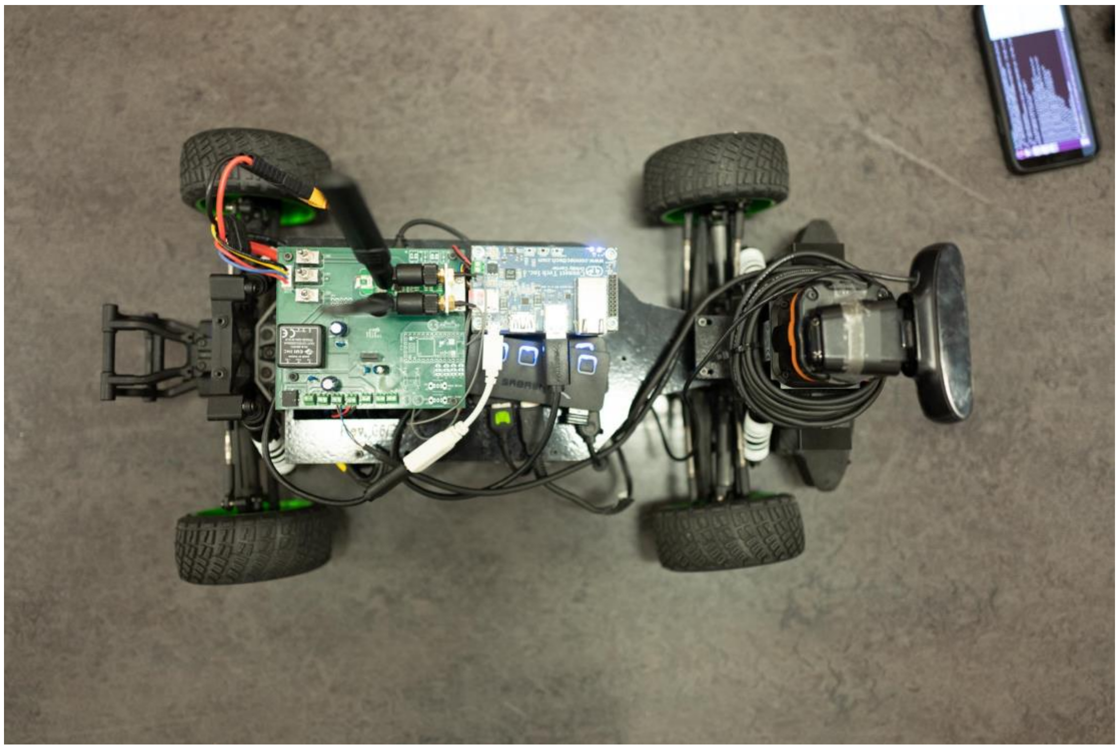


Figure 2. A top-down view of the electronics mounted on the F1/10th car.

Figure 2 shows a top-level view of the hardware and electronics on board the F1/10th autonomous car. Note that the RGB USB camera has been mounted on top of the lidar sensor.

Description of Approach

We use the AprilTag system, developed by the APRIL Robotics Laboratory at the University of Michigan, to get pose estimation: xyz position and 3D orientation. Since the F1/10th cars are assumed to be racing in a 2D plane, we only need the yaw orientation information. We do not need the roll nor pitch information since cars are not going up/down slopes, nor are they banking on angled turns. AprilTag is a type of visual fiducial – an "artificial landmark designed to be easy to recognize and distinguish from one another" (Olson 2010). The AprilTag system has a set of QR-code-looking tags which can be printed at various sizes. The tags are in black and white, and each tag has a unique id. Since we only need one tag to be detected for purposes of this paper, we use the id 0 tag from the 36h11 family, which is the standard, most commonly used AprilTag. I chose to use the AprilTag system, as opposed to alternatives such as ARToolkit and ARTag, because AprilTag has a well-written, well-documented ROS package (apriltags2_ros) and because I have seen the AprilTag library being used in the GRASP Lab and in the mLab. Hence I would have people to ask for advice if I got stuck on using the AprilTag library. AprilTag has also been documented to have decent accuracy, around 5 – 10% errors, with xyz pose estimation errors increasing as distance increases. During my preliminary testing, I verified that the AprilTag estimated poses were within around 5% for a tag on letter-sized paper up to 3 meters away, and around 10% error up to 8 meters away.

Figure 3 shows a closer look at how the car behind uses an RGB camera to detect the AprilTag mounted on the back of the leading car. Figure 4 shows how the AprilTag is mounted on the leading car. We had to mount the AprilTag higher so that there would be no occlusion. If even part of the black square is occluded by part of the car, then most of the time the AprilTag will not be detected and hence we will not have pose or orientation information.

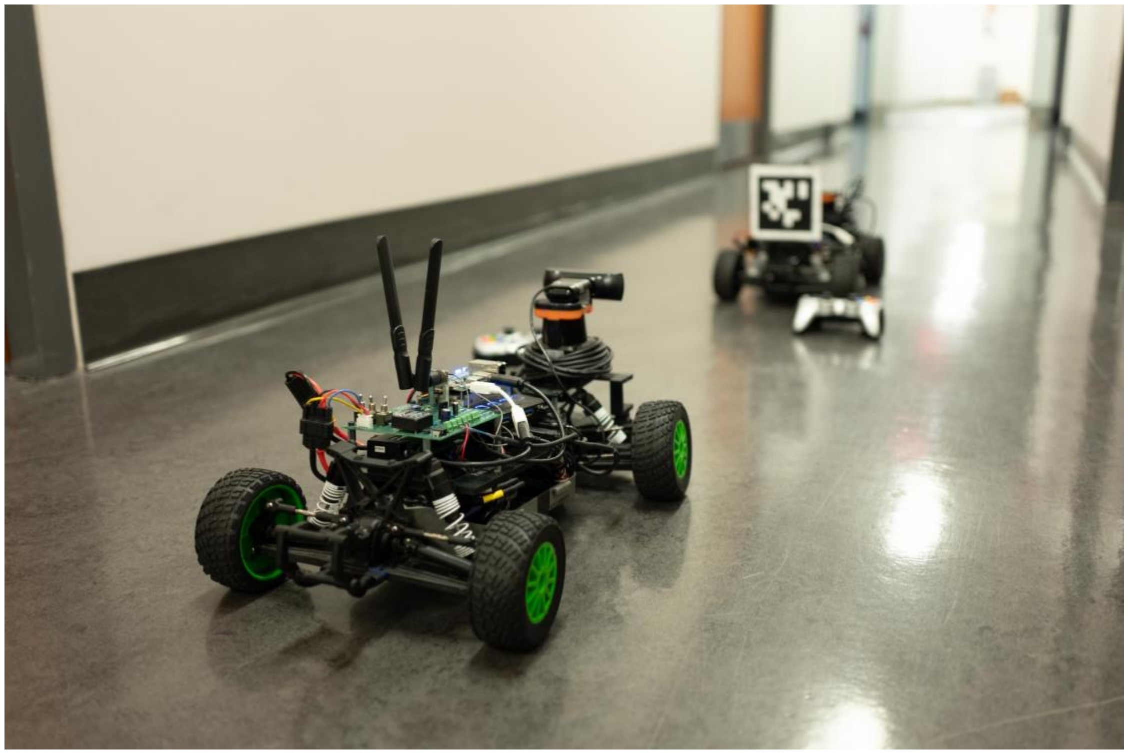


Figure 3. A closer look at how the car behind uses an RGB camera to detect the AprilTag mounted on the back of the car in front.

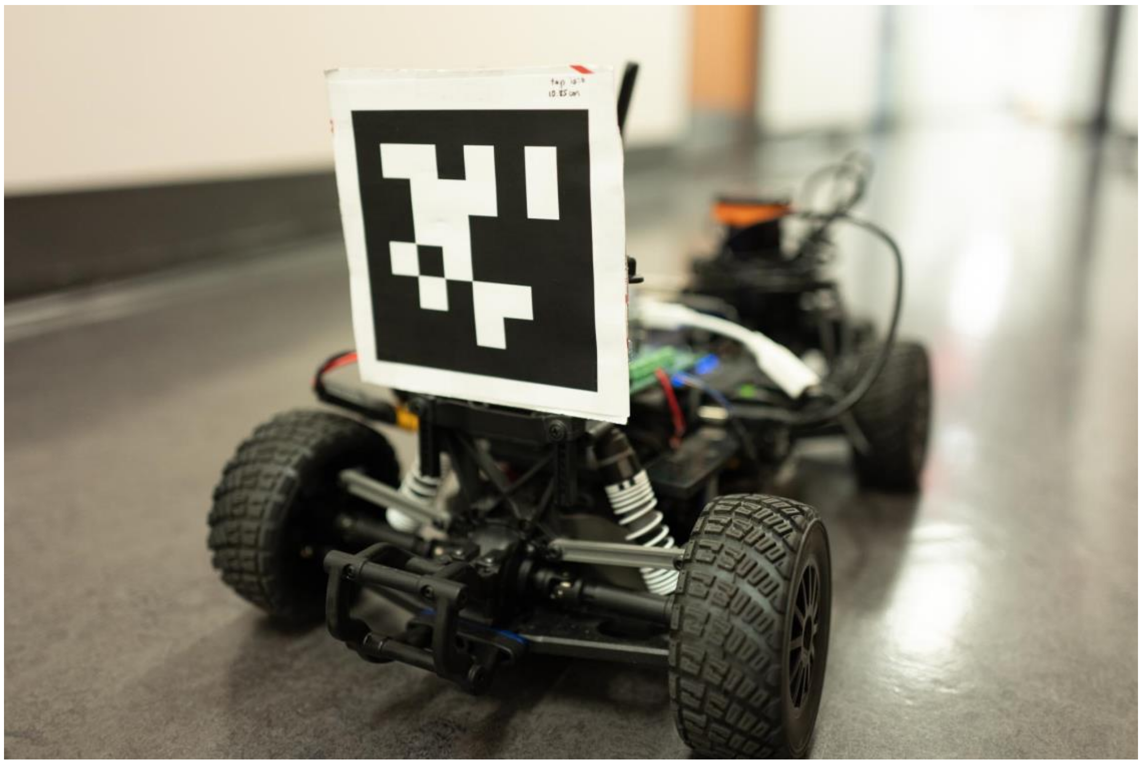


Figure 4. A close-up look at the mounting of the AprilTag on the leading car. Originally, the AprilTag was mounted lower, but any type of occlusion that covers the black square causes the AprilTag to not be detected, hence it is mounted higher for now occlusion.

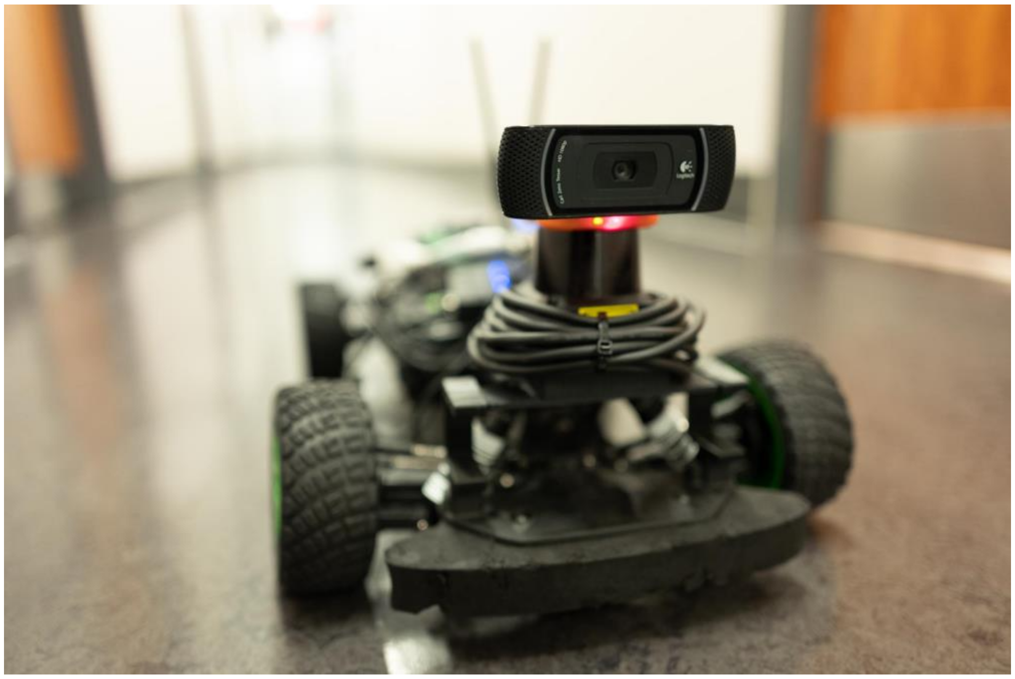


Figure 5. A close-up frontal look at the car with a lidar sensor and a RGB USB webcam. I am using the Logitech C910 USB webcam which records at up to 1080p at 30Hz. The camera is mounted carefully on top of the lidar sensor such that it does not obstruct the lidar sensor's planar measurements.

Figure 5 shows a close-up on how I mounted the Logitech C910 camera on top of the Hokuyo lidar sensor. I was careful to make sure that the camera would not occlude the Hokuyo lidar sensor 270 degree frame of view. The camera does not occlude the lidar sensor because the Hokuyo is a 2D plane lidar sensor, not a 3D lidar sensor. Hence as long as the Logitech C910 is not within the 2D plane it is not blocking the lidar sensor. The Logitech C910 camera can record at up to 1920 x 1080 resolution at 30fps (frames per second), but when feeding this image into ROS using the usb_cam package, I found that 1920 x 1080 images were processed very slowly by the apriltags2_ros package, at only 1 – 2 Hz. Below is a chart of the speeds at which images of various resolutions were processed by the apriltags2_ros library.

Resolution	Hz processed by apriltags2_ros	Aspect Ratio
1920 x 1080	1 – 2 Hz	16:9 (Wide)
1280 x 720	3.5 – 4 Hz	16:9 (Wide)
640 x 480	11 – 12 Hz	4:3 (Standard)

Table 1. Speeds at which images of various resolutions are processed by apriltags2_ros package

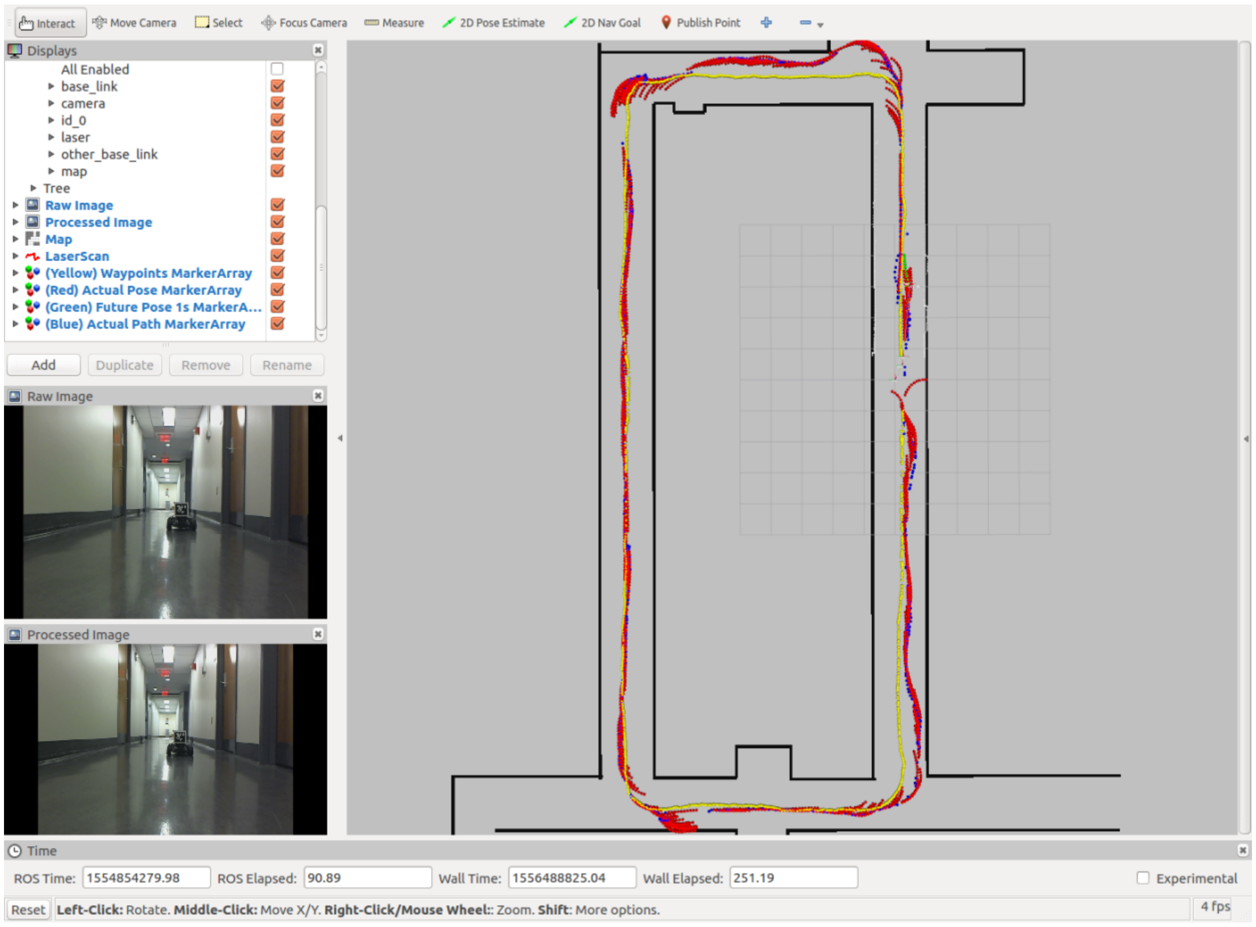
For an autonomous car racing setting where cars will be racing at around 5 meters per second, a refresh rate of 1 – 2 Hz is way too slow. By the time we fetch the next image, the car will have already traveled 2.5 meters! Upon consultation with Matt O'Kelly, a PhD candidate in the mLab and an expert and pioneer on the F1/10th movement, we agreed that a speed of over 10 Hz would be necessary to make meaningful near-future pose estimates.

Hence I decided, after weeks of deliberation and experimentation, to use the 640 x 480 resolution. There were some tradeoffs made here. The advantage of 640 x 480 is that it allows for a 11 – 12 Hz processing by apriltags2_rso, which satisfied the requirement that Matt and I put on for having over 10Hz AprilTag pose estimate speeds. Another positive thing is that 640 x 480 is sufficiently high resolution to detect the relatively small sized AprilTag which is 10.85 centimeters (.1085 meters) wide, from up to 5 meters away. I figured that it would not be necessary to detect tags more than 5 meters away because at that far of a distance, our car will not be making any passing maneuvers anyways. One negative aspect of the 640 x 480 resolution is that the image is not wide. The aspect ratio is 4:3, which translates to 1.33:1, as opposed to 16:9, or 1.77:1. Having a wide frame of view is important for pose estimation of a car because as the passing car encroaches closer on the leading car, the leading car takes up more of the frame of view, and hence parts of the tag may fall out of the frame of view with a normal (non-wide) aspect ratio. I tried setting the resolution to 480p, or 858 x 480, but the usb_cam returned a raw image which was entirely distorted and basically just looked like a bunch of random horizontal lines streaking across the screen. Upon further investigation, it turns out that the culprit is rooted in the hardware of the Logitech C910 camera, where only specific fixed resolutions can be shown. The lowest resolution 16:9 wide aspect ratio image that can be shown is the 720p, or 1280 x 720 resolution. As an alternative, I bought the Logitech BRIO 4K webcam, but also had no success with it. In fact, the image would not even show up at any resolution with the usb_cam interface, so I had no choice but to stick with the Logitech C910 webcam.

Using the usb_cam library also required calibrating the camera. To do this, I used the ROS camera_calibration package, where I printed out a 8 x 10 square checkerboard, mounted it on a hard flat surface, and held it up to the RGB camera at various angles, distances, and positions in order to calibrate the camera.

I used ROS (Robotic Operating System) for this project, using code that was started by the mit-racecar project, then refined by mLab's F1/10th movement, and including some code I wrote in summer 2018 while I was working in the mLab to develop the ESE 680 autonomous car racing course which I then assistant taught in the fall of 2018.

I wrote a ROS node in Python called future_pose_estimation.py, which subscribes to the /tag_detections topic and /tf topics and returns a set of 10 waypoints which represents where the leading car is predicted to be up to 1 second in the future. The 10 waypoints represent 0.10 seconds in the future, 0.20 seconds, …, up to 1.0 seconds in the future. Below are some screenshots of the outputs in Rviz (which is a visualization component of ROS).

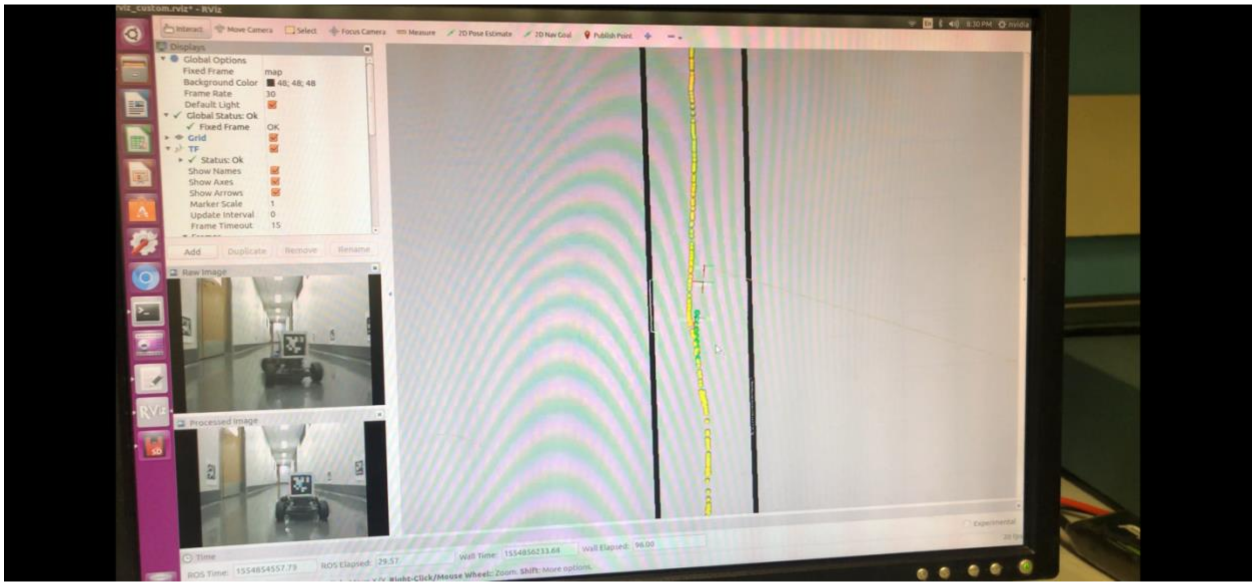


ROS Time: 1554854279.98 ROS Elapsed: 90.89 Wall Time: 1556488825.04 Wall Elapsed: 251.19

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options.

Figure 6. Screenshot of Rviz workspace

In Figure 6 above, on the bottom left corner is the raw image that comes out of the usb_cam package. The bottom image on the left is the image after processing by the apriltags2_ros package which adds on a box (difficult to see here) over the image with the outline and pose estimation. On the main screen on the right, in yellow, is the set of manually driven waypoints that approximates the general direction and path that a car is expected to follow around the world. The 10 points in green represent the predicted future pose estimate of the leading car. Details on how these 10 points are computed will be given later in the paper, under the Algorithms section.



```
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
  Default Light        ✓
✓ Global Status: Ok
  ✓ Fixed Frame        OK
Grid                   ✓
TF                     ✓
  ✓ Status: Ok
  Show Names           ✓
  Show Axes            ✓
  Show Arrows          ✓
  Marker Scale         1
  Update Interval      0
  Frame Timeout        15
```

Figure 7. Here is another test case where the car is generally going straight. The green dots illustrate where the car is predicted to be in the future.

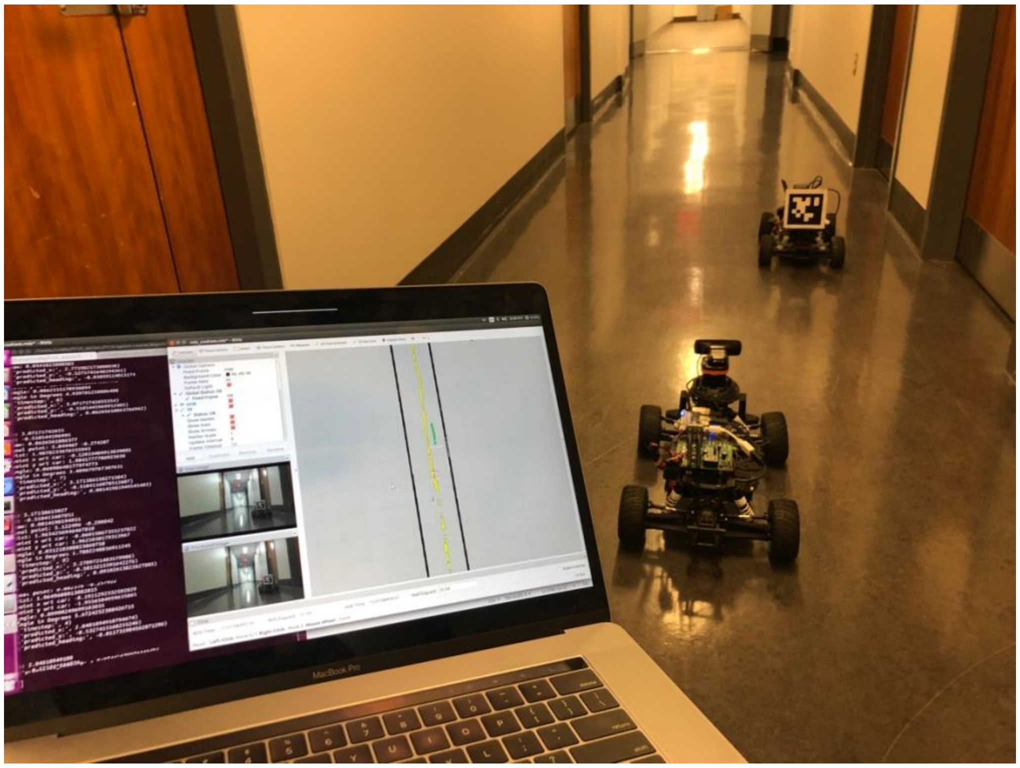


Figure 8. Real-life image of the cars, along with predicted near-future pose.

In figure 8, I have a picture with the two cars in the Levine Hall second floor hallways, just outside the mLab. On the left is my laptop screen, which is streaming the view from the Jetson TX2 over wifi using VNC Viewer. On the laptop screen you can see green dots which represent where the car in front currently is and where it is predicted to be 1 second into the future.

Part of my near-future pose estimation algorithm, which will be described in detail later, depends on knowing a general set of waypoints that the cars are expected to follow in a race. Here, I manually drove the car as close to the center of the hallways as possible, in a counter-clockwise direction to complete a loop. I used the MIT racecar particle filter library (https://github.com/mit-racecar/particle_filter) in order to localize the car using the lidar scans that are matched with a 2D map of Levine Hall which I created manually. I created this map manually (in figure 10, the black lines) using an architectural floor plan which I obtained from the School of Engineering administration, which is accurate and represented in meters. Particle filter will match the laser scans with the given map, and also taking in an initial pose estimation, track the car as it moves through the map. I wrote a custom node in Python, based on initial code provided by Matt O'Kelly, in order to generate this smooth set of waypoints for the car.

Approaches Which I Did Not Use

Equally important to discussing the approach I used are the approaches which I did not use. While trying to solve this problem of near-future pose estimation, there were many other approaches that came to mind but I chose not to pursue for various reasons.

One approach for pose estimation of a car is using blinking LED lights as a type of fiducial tag. Researchers [cite the article I read] have demonstrated that one can use an RGB camera to detect patterns in flashing LED lights in order to detect points. We should know the location of the points relative to one another in the actual world, in terms of meters apart from each other, and then be able to derive the pose of the object in the 3D world. In practice, this would look like 4 flashing LED's mounted on the leading car. The LED's could not be mounted uniformly, such as one on each corner, but rather would have to be spaced apart irregularly. That way, there would not be multiple solutions to the optimization problem of deriving the pose of the 4 points. I ultimately decided not to pursue this approach because flashing LED's can be occluded, especially in a race, by the car itself. As the car yaws from side to side while making turns, the front left or front right LED lights may fall out of view from the camera.

Another similar approach is to use infrared LED's and an RGB camera which has been modified with an Infrared pass filter (Lengenfelder et al. 2018). This way we would not need to blink the LED's in order to notice them, as infrared light in the infrared spectrum would be detected without the need to blink the infrared lights. A research paper from ETH Zurich shows that this approach has been effected when mounted on drones, but I decided not to do this approach because some infrared led's may become occluded. Moreover, it is a lot more simple to use an AprilTag which only requires mounting a piece of paper, as opposed to having to wire electronics and find a power source for those electronics.

There have also been many neural network / machine learning approaches to this problem. [cite some papers which I read but did not understand]. These involve providing training images which are images from the USB webcam, of the leading car. I would also need to provide ground truth, actual measurements for xyz position and orientation of the leading car. This would be a huge challenge to measure since it would be tedious to measure so hundreds or even thousands of distances. Moreover, training would need to be in different environments – indoors, indoor hallways, indoor non-rigid race tracks, outdoors, dark lighting conditions, bright lighting conditions, and much more. Because I also do not understand machine learning nor neural networks, this was not the approach for me. An idea for future work though, is that if April Tags provide somewhat of a good ground truth position and orientation measurement, one could use mount the AprilTag and use its estimated pose as the pose which is fed into the neural network alongside the image.

Another promising approach I read about involved trying to match up images from an RGB camera with a known 3D model of that object. In this case, imagine if we had a 3D model of the F1/10th car, and an image from the USB webcam perspective of the car from behind. The algorithm would then try to match up the image with a set of poses to predict the most accurate pose of the car. Behind the scenes, this approach [reference this paper] requires a neural network, which seems overkill for this problem when an AprilTag can be used for now.

Algorithm

Approach #1. Change in XY Position

The initial approach to the near-future pose estimation algorithm was a simple change in xyz type of model. I chose to start with the most simple approach first. The general idea is to take 2 consecutive estimates of xy position over some period of time. The yaw orientation is not taken into account here. Over that period of time, compute the change in xy position. The change in xy divided by the change in time gives a sense of velocity at a certain direction. Then multiply that velocity by 1 second to compute the future change in xy position. Add that change in xy position to the current position, and that derives a simple estimate of where the car will be in the future.

The results of this approach were not that great. The algorithm could predict where the car would be in the future for simple cases like moving straight forward or straight back, but it was unrealistic on turns. For example, if a car in the hallway were moving towards the walls of the hallway, it would project forward that the car would go through the hallway. Obviously, this is not possible. Moreover, since the yaw orientation of the car wasn't taken into account, this simple algorithm was missing out on an important point of information provided by the AprilTag library. Since XY position estimates could occasionally be wrong, having an additional point of information like yaw orientation would provide a mechanism to double-check the computations.

Approach #2. Linear Model, using XY Position and Yaw Orientation

The next approach was to use both the XY position and yaw orientation information. Below is a figure of what this looks like.

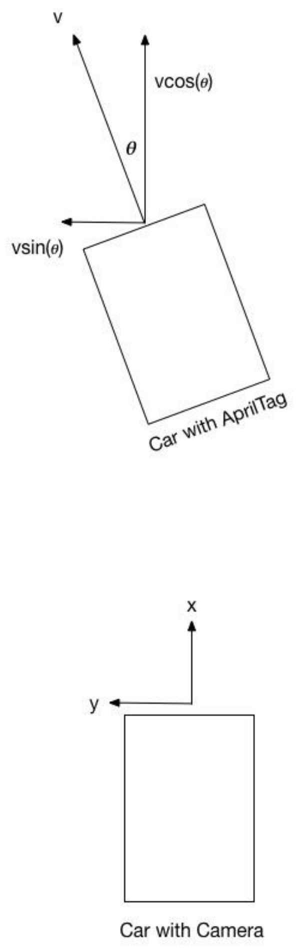


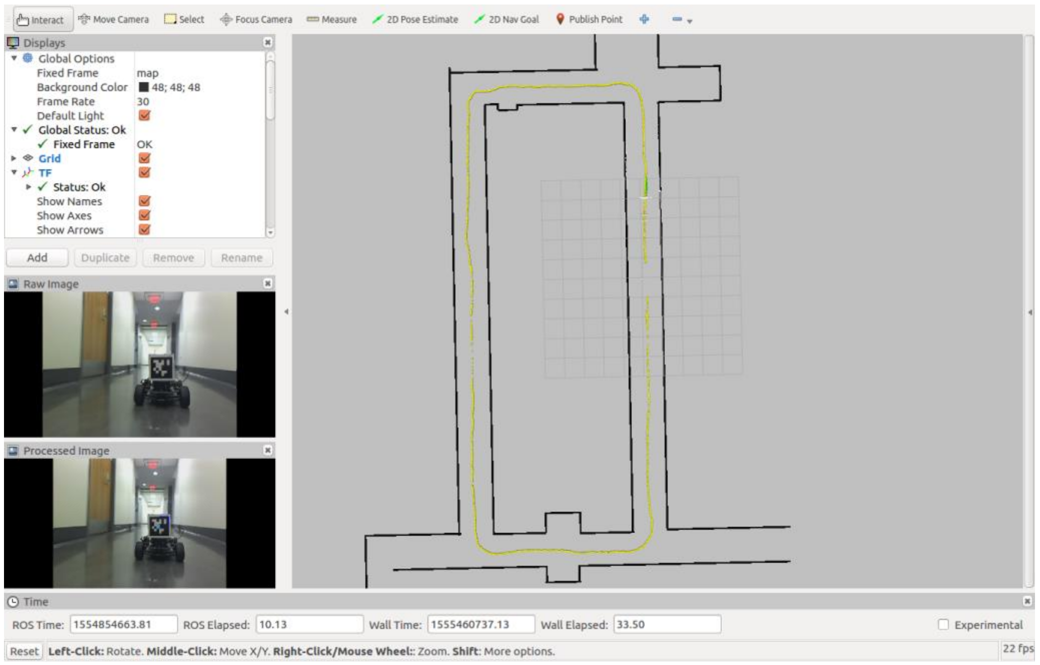
Figure 9. Linear velocity model. Yaw angle $\boldsymbol{\theta}$ and velocity are measured relative to the car with the camera. Once we know the x and y components of velocity of the leading car, then we can multiply the velocity by duration of time to estimate where the car will be in the near future.

The linear velocity model is an improvement on the displacement-only model from Approach #1 described above. Here we know the yaw orientation of the leading car relative to the car behind it – the car with the camera. Since the AprilTag is attached to the base link, or the back of the leading car, we get a good estimate of where the car will be in the near future. Since there is some type of delay for servo actuations for the steering wheels, knowing the orientation of the car is a pretty accurate measurement since it will probably take around a quarter of a second for the car to change its steering angle drastically.

What are the downsides of this linear model velocity model? For one, it suffers the same problem as Approach #1 where the world map is not taken into account. With this model, the leading car can be oriented towards a wall and the linear velocity model will predict that the car will continue going straight through the wall. In reality, in a F1/10th race, we know the race course ahead of time and hence we need to factor in the race course map into the predictions of where the car will move. The intuition here, as discussed with Professor Rahul Mangharam, is that if the leading car is approaching a left turn, but the car is moving straight, we should factor in the weight that we *expect* the car to veer towards the left turn because otherwise it will crash. This observation leads to the intuition which is Approach #3, next.

Approach #3. Pure Pursuit following a set of waypoints

Pure pursuit is an algorithm I implemented where the car follows a set of waypoints around the map. Obtaining the set of waypoints is described earlier in the paper, and involves manually driving the car while running a localization algorithm within the known map. Here, I manually created this set of waypoints by driving the car in a counter-clockwise loop around the Levine Hall 2nd floor hallways. I purposely drove the car as centered as possible in the hallways, to give a neutral estimation of where a car might be at any point in time during the race. Remember that here the overall idea is to estimate where the car is going to be in the future by knowing its location in the map, then finding the next closest waypoint that the car attempts to reach. The pure pursuit algorithm outputs a steering angle for the car to get to that waypoint, and I project this forward 0.10 seconds, update the car's location, call pure pursuit again. I take that new pure pursuit output steering angle and project the car's location forward another 0.10 seconds, and continue until doing this 10 times and reaching 1 second into the future.



Global Options
Fixed Frame: map
Background Color: 48; 48; 48
Frame Rate: 30
Default Light: ✓
Global Status: Ok
Fixed Frame: OK
Grid: ✓
TF: ✓
Status: Ok
Show Names: ✓
Show Axes: ✓
Show Arrows: ✓

ROS Time: 1554854663.81 ROS Elapsed: 10.13 Wall Time: 1555460737.13 Wall Elapsed: 33.50

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.

Figure 10. The black lines represent the map around Levine 2nd floor. The yellow points represent the waypoints in a counter-clockwise loop around the hallways.

Here is an image of the map around the Levine 2nd floor hallways, The yellow dots represent the set of waypoints plotted around the hallway, which are represented as (x, y) coordinates. Remember that since the world is assumed to be a 2D flat plane in the F1/10th racing environment, we can omit the z axis. Note further that since we only care about (x, y) coordinates, we do not need to include in the waypoints the orientation that the car should be at at each position.

What is the pure pursuit algorithm? According to the original 1992 Carnegie Mellon paper, "Implementation of the Pure Pursuit Path Tracking Algorithm," "pure pursuit is a tracking algorithm that works by calculating the curvature that will move a vehicle from its current position to some goal position." The algorithm is composed of a few steps:

1. Determine the current location of the vehicle. That is, know where the vehicle is in the map.
2. Find the path point closest to the vehicle: this is called the goal point. That is, look through all the waypoints and find the one that is closest to the vehicle but just farther than its lookahead distance.
3. Transform the goal point to vehicle coordinates. That is, get the coordinates of the goal point in the vehicle's frame of view.
4. Calculate the curvature. That is, compute the steering angle that the vehicle should make in order to reach the goal point.
5. Update the vehicle's position. This can be done by projecting the steering angle with v$\cos(\boldsymbol{\theta})$ and v$\sin(\boldsymbol{\theta})$.

The lookahead distance is a parameter that is often tuned. A small lookahead distance, such as 0.5 meters, will cause the vehicle to follow the path points, or waypoints, very closely. However, the vehicle will oscillate more. On the other hand, a larger lookahead distance, such as 2 – 3 meters, will allow for smooth linear motions with little oscillation. However, on turns the vehicle may cut corners and end up hitting walls. Generally, a smaller lookahead distance is more effective at low speeds and a larger lookahead distance is more effective at high speeds. For purposes of this research, I set the lookahead distance to a constant 2 meters, which is a pretty neutral value for a car that is going around 1 – 2 meters per second.

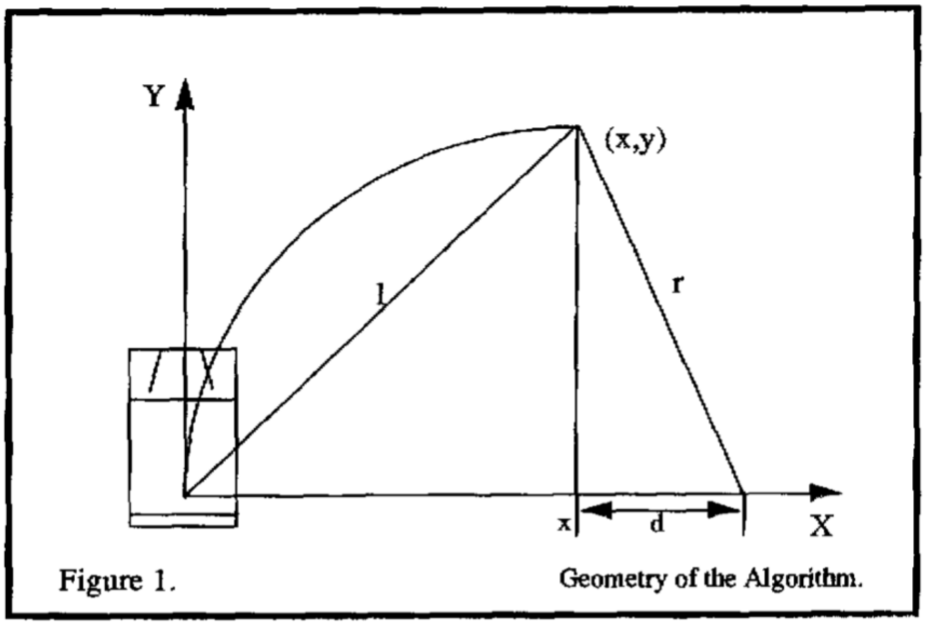


Figure 11. A figure borrowed from the Pure Pursuit paper. The pure pursuit algorithm calculates the steering angle in order to get the car to the goal point (x, y).

$$d = r - x$$

$$(r-x)^2 + y^2 = r^2$$

$$r^2 - 2rx + x^2 + y^2 = r^2$$

$$2rx = l^2$$

$$r = \frac{l^2}{2x}$$

$$\gamma = \frac{2x}{l^2}$$

Figure 12. This figure is a derivation of the curvature γ of the car, which is related to 2 times the x coordinate in the car's frame divided by the Euclidean distance squared.

Pure pursuit alone is a nice algorithm which can predict the general car movement. For instance, if the car is approaching a left turn, we can expect the car to be turning left soon. Or if the car is going straight down a long hallway, we can expect the car to follow the general waypoints and continue moving straight down the hallway. Pure pursuit alone, however, is also not realistic. For instance, if the car is going straight down a hallway but hugging the right while, and if the waypoints were created in a way such that waypoints are in the center of the hallway, then the pure pursuit algorithm would predict that the car would turn its steering angle to the left to get back to the middle of the hallway. In reality, an autonomous car might continue hugging the right wall in preparation for a nice apex on an upcoming left turn.

Consider another example of where pure pursuit alone has limitations in predicting near-future pose estimation. Imagine a car is currently making a left turn, and is in the center of the hall. Also suppose the car is moving very fast. The pure pursuit algorithm does not take velocity into account. So pure pursuit would posit that the car's goal point some lookahead distance in front is still in the center of the hallway and assume that the car can reasonably get there while making a turn at high speeds. In reality, however, the car will inevitably drift; the car will probably end up on the right side of the hallway after making the left turn at full speed.

Approach #4. Weighted average between linear model and pure pursuit

We have seen that the linear model is limited in that it does not take into account the race map. We have also seen that the pure pursuit model is limited in that it does not take into account the current trajectory of the car given its orientation. Hence I propose combining these two methods via a weighted averaging. The intuition is that in the very near future, like 0.1 seconds to 0.4 seconds into the future, we should expect the linear model to hold more weight. But in the later near future, so 0.4 to 1.0 seconds into the future, we should expect the pure pursuit model to hold more weight because a car will generally follow the map in the longer run.

Discount = X = 0.7

Time in Future (seconds)	Weight on Linear Model, y	Weight on Pure Pursuit Model, 1 - y
0.1s	X^1 = 0.70	0.30
0.2s	X^2 = 0.49	0.51
0.3s	X^3 = 0.34	0.66
0.4s	X^4 = 0.24	0.76
0.5s	X^5 = 0.17	0.83
0.6s	X^6 = 0.12	0.88
0.7s	X^7 = 0.08	0.92
0.8s	X^8 = 0.06	0.94
0.9s	X^9 = 0.04	0.96
1.0s	X^10 = 0.03	0.97

Table 2. As the projected future time increases, more weight is placed on the pure pursuit pose estimation and less weight is placed on the linear model.

Notice how in the table above, as the projected future time increases, more weight is placed on the pure pursuit pose estimation and less weight is placed on the linear model. For 0.1s into the future, linear model is weighted 0.7 and pure pursuit model is weighted 0.3. By the time we have reached 1.0s into the future, linear model is only weighted 0.03 while pure pursuit model is weighted 0.97.

We recompute the predicted pose of the car at each 0.1 second interval. Hence to predict where the car will be in 1 second in the future, there would have been 10 calls to pure pursuit and linear model, each step taking in as input the predicted pose computed by the previous step.

$$X_N = \sum_{i=0}^{N} \alpha^2 * (\text{linear model prediction}) + (1-\alpha^2) * (\text{pure pursuit model prediction})$$

$\alpha = 0.7 = \text{discount}$

X_1, for example, would represent the position of the car 0.1 seconds into the future.
X_2 would represent the position of the car 0.2 seconds into the future. It would sum up the relative movement of the car from 0 to 0.1 seconds with the next relative movement of the car from the 0.1 to 0.2 seconds.

Experimental Results

This section will report experimental results from the Approach #4, which is the weighted average between linear model pose predictions and pure pursuit pose predictions.

Experiment 1. Accuracy of Raw AprilTag Measurements.

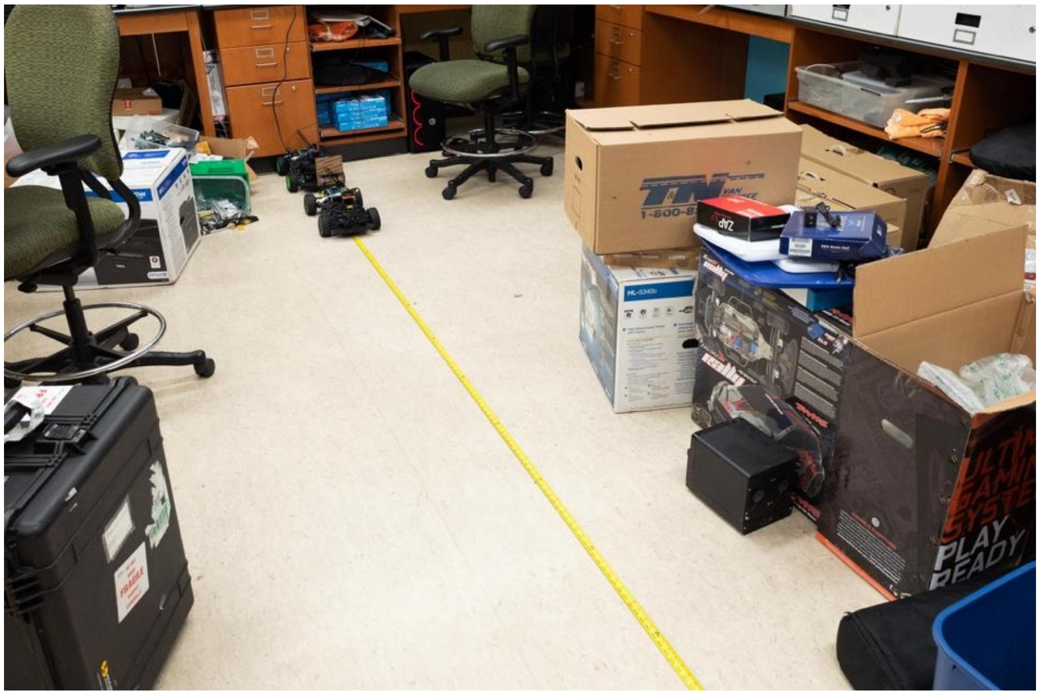
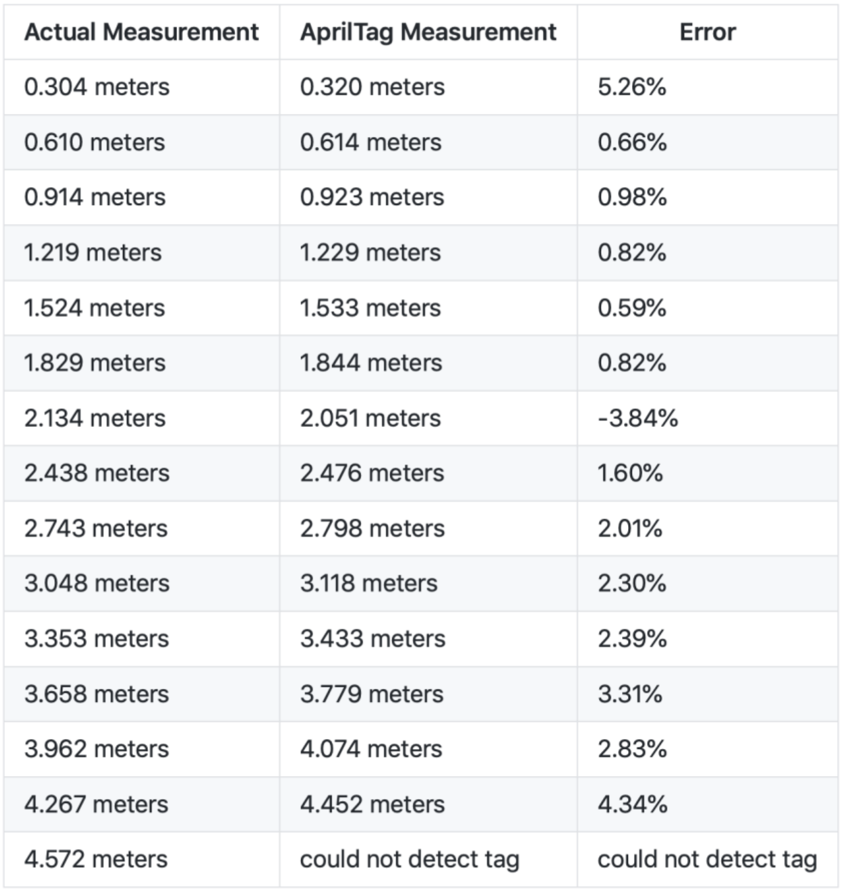


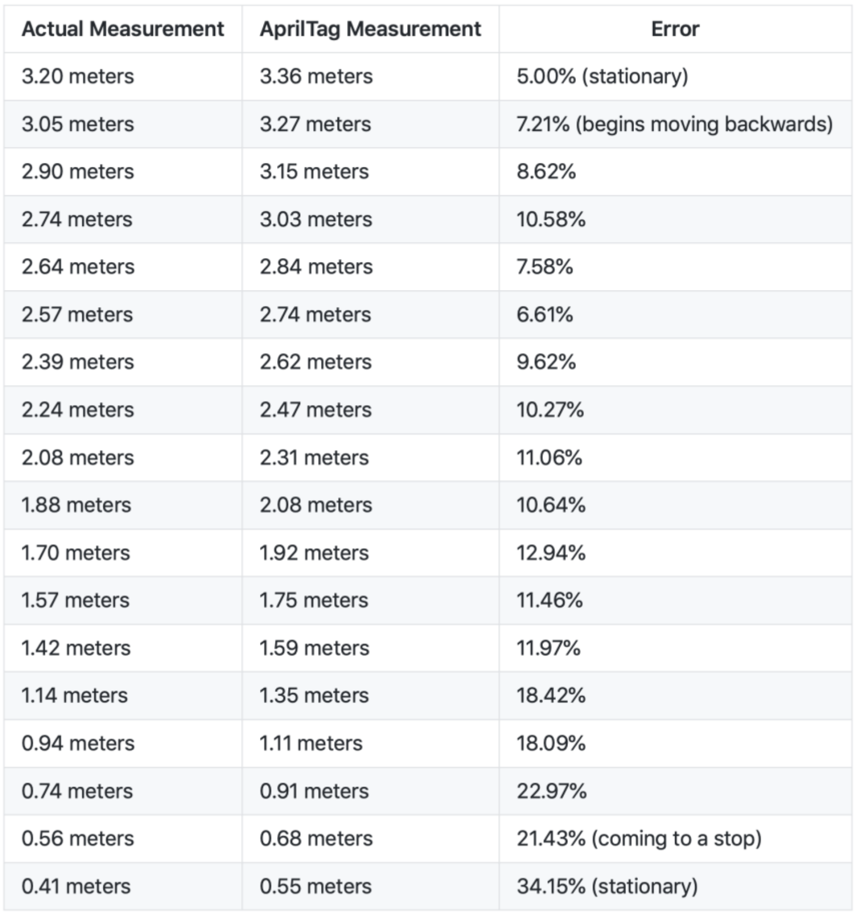
Figure 13. Experiment setup to measure distances at 1 foot, 2 feet, …, 15 feet.



Actual Measurement	AprilTag Measurement	Error
0.304 meters	0.320 meters	5.26%
0.610 meters	0.614 meters	0.66%
0.914 meters	0.923 meters	0.98%
1.219 meters	1.229 meters	0.82%
1.524 meters	1.533 meters	0.59%
1.829 meters	1.844 meters	0.82%
2.134 meters	2.051 meters	-3.84%
2.438 meters	2.476 meters	1.60%
2.743 meters	2.798 meters	2.01%
3.048 meters	3.118 meters	2.30%
3.353 meters	3.433 meters	2.39%
3.658 meters	3.779 meters	3.31%
3.962 meters	4.074 meters	2.83%
4.267 meters	4.452 meters	4.34%
4.572 meters	could not detect tag	could not detect tag

Table 2. Actual measurement vs AprilTag Measurement, along with percentage error.

Experiment 2. Accuracy of Raw AprilTag Measurements While Car is Moving Backwards
Video Link: https://www.youtube.com/watch?v=Qp6suG0r_1M

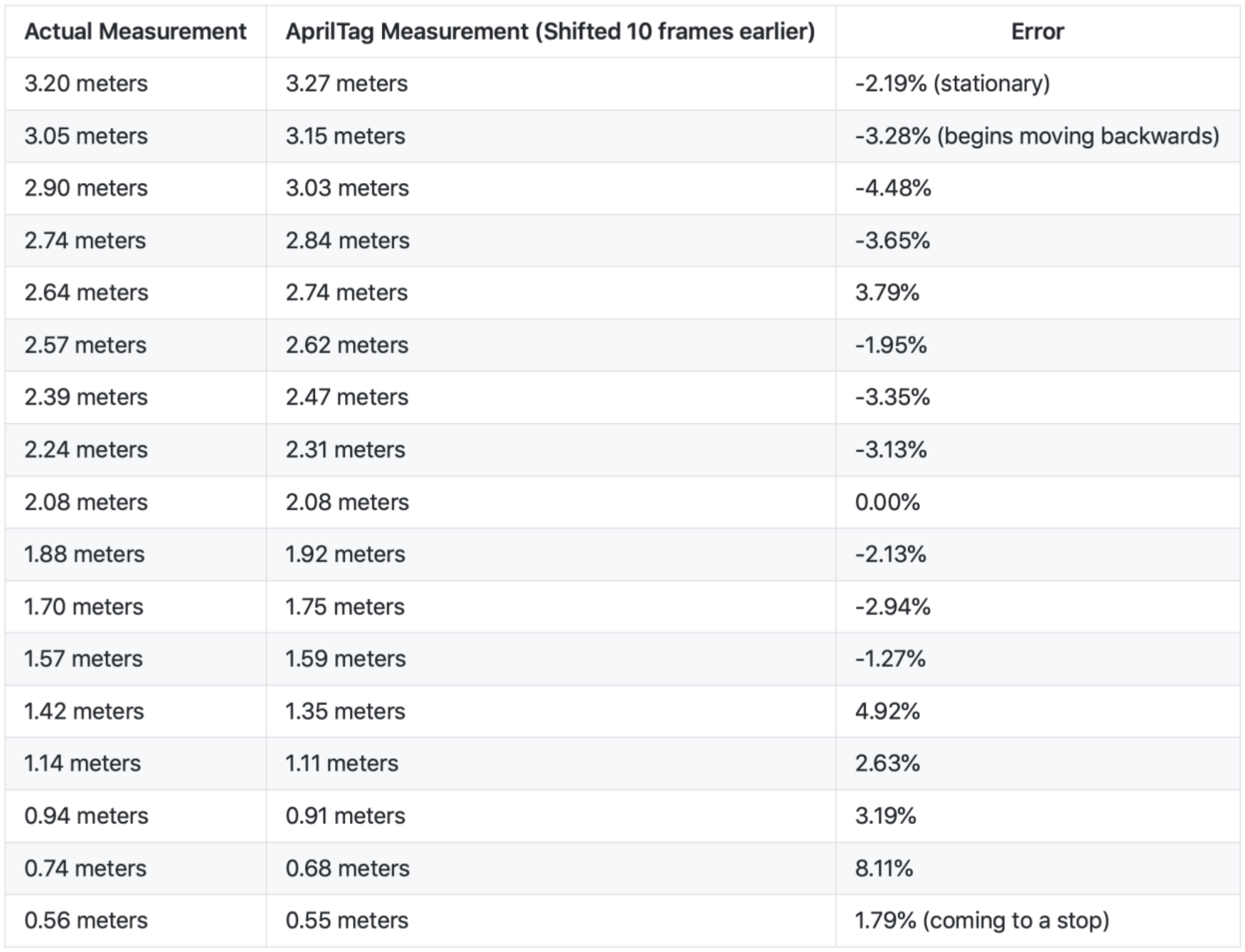


Actual Measurement	AprilTag Measurement	Error
3.20 meters	3.36 meters	5.00% (stationary)
3.05 meters	3.27 meters	7.21% (begins moving backwards)
2.90 meters	3.15 meters	8.62%
2.74 meters	3.03 meters	10.58%
2.64 meters	2.84 meters	7.58%
2.57 meters	2.74 meters	6.61%
2.39 meters	2.62 meters	9.62%
2.24 meters	2.47 meters	10.27%
2.08 meters	2.31 meters	11.06%
1.88 meters	2.08 meters	10.64%
1.70 meters	1.92 meters	12.94%
1.57 meters	1.75 meters	11.46%
1.42 meters	1.59 meters	11.97%
1.14 meters	1.35 meters	18.42%
0.94 meters	1.11 meters	18.09%
0.74 meters	0.91 meters	22.97%
0.56 meters	0.68 meters	21.43% (coming to a stop)
0.41 meters	0.55 meters	34.15% (stationary)

Table 3. Actual vs AprilTag measurements while car is moving backwards. Values are taken at equidistant steps over time, 10 frames for 24fps video is roughly 0.42 second increments.

The results here look a lot worse than I thought they would be. The error seems to grow larger as the car moves backwards closer to the camera. The reason is likely because of some type of delay in computing the AprilTag measurement, which results in a pretty constant difference in actual vs AprilTag measurement of around 0.15-0.20 meters. Here it seems like that the AprilTag is taking around 10 frames (at 24 frames per second) to process, hence if we were to move all the AprilTag measurements earlier by 10 frames, or 0.42 seconds, the measurements would line up. This graph is insightful because it shows us an AprilTag processing delay of around 0.42 seconds. I don't think there are any confounding variables with Experiment 1 because Experiments 1 and 2 share the same setup, and were recorded only minutes after each other. Another reason for the roughly 0.42 second delay is that the AprilTag is also processing at around 10Hz, around 0.1 seconds, so it may also take 0.1 seconds before getting the next AprilTag measurement.

Notice that if we shift the AprilTag measurements such that they are 10 frames earlier (0.42 seconds earlier), the error is in line with the results of Experiment 1 when the AprilTag was stationary.

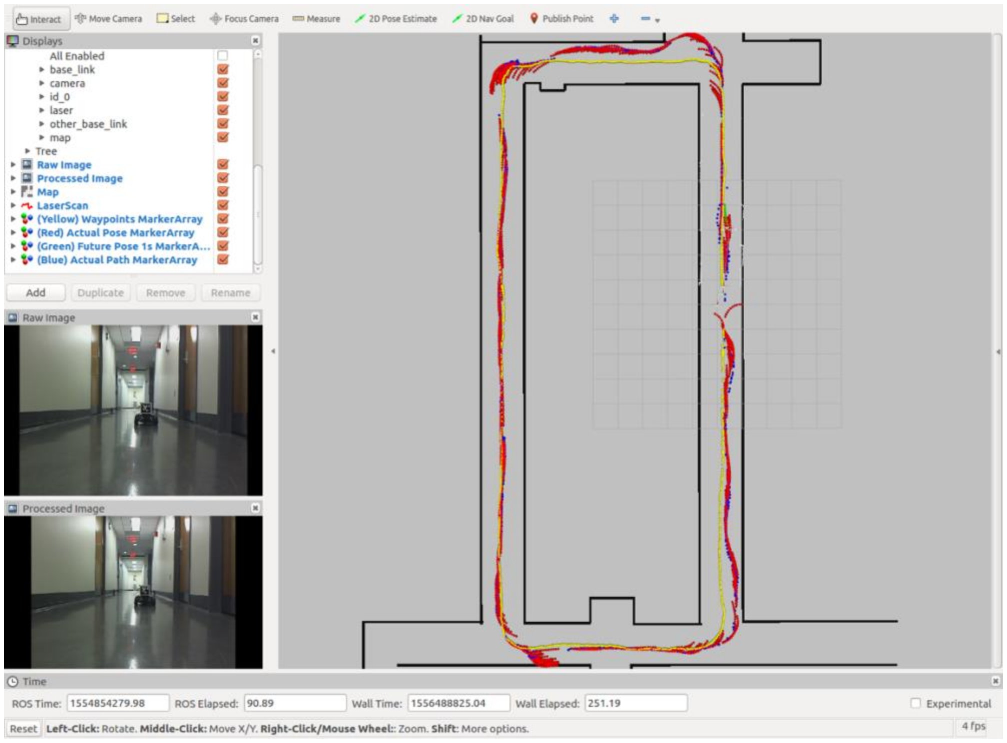


Actual Measurement	AprilTag Measurement (Shifted 10 frames earlier)	Error
3.20 meters	3.27 meters	-2.19% (stationary)
3.05 meters	3.15 meters	-3.28% (begins moving backwards)
2.90 meters	3.03 meters	-4.48%
2.74 meters	2.84 meters	-3.65%
2.64 meters	2.74 meters	3.79%
2.57 meters	2.62 meters	-1.95%
2.39 meters	2.47 meters	-3.35%
2.24 meters	2.31 meters	-3.13%
2.08 meters	2.08 meters	0.00%
1.88 meters	1.92 meters	-2.13%
1.70 meters	1.75 meters	-2.94%
1.57 meters	1.59 meters	-1.27%
1.42 meters	1.35 meters	4.92%
1.14 meters	1.11 meters	2.63%
0.94 meters	0.91 meters	3.19%
0.74 meters	0.68 meters	8.11%
0.56 meters	0.55 meters	1.79% (coming to a stop)

Table 4. Actual vs AprilTag measurements, with AprilTag measurements shifted earlier by 10 frames, or 0.42 seconds. AprilTag measurements shifted one row up essentially.

Notice that here the error measurements are much more in line with the results of Experiment 1 when the AprilTag was stationary. This is evidence that the AprilTag measurements are indeed delayed by around 10 frames, or around 0.4 seconds. This delay could result from printing the AprilTag measurements in the terminal window, since it is known that Python print statements - especially when printing a lot of statements - slows down the code.

Experiment 3. Accuracy of Near Future Pose Predictions



All Enabled ☐
- base_link ✓
- camera ✓
- id_0 ✓
- laser ✓
- other_base_link ✓
- map ✓
- Tree
- Raw Image ✓
- Processed Image ✓
- Map ✓
- LaserScan ✓
- (Yellow) Waypoints MarkerArray ✓
- (Red) Actual Pose MarkerArray ✓
- (Green) Future Pose 1s MarkerA... ✓
- (Blue) Actual Path MarkerArray ✓

ROS Time: 1554854279.98 ROS Elapsed: 90.89 Wall Time: 1556488825.04 Wall Elapsed: 251.19 ☐ Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options.

Figure 13. This is a color coded image comparing the actual versus predicted path.

RED: Actual Pose of the car in front (saved waypoints around the entire loop)

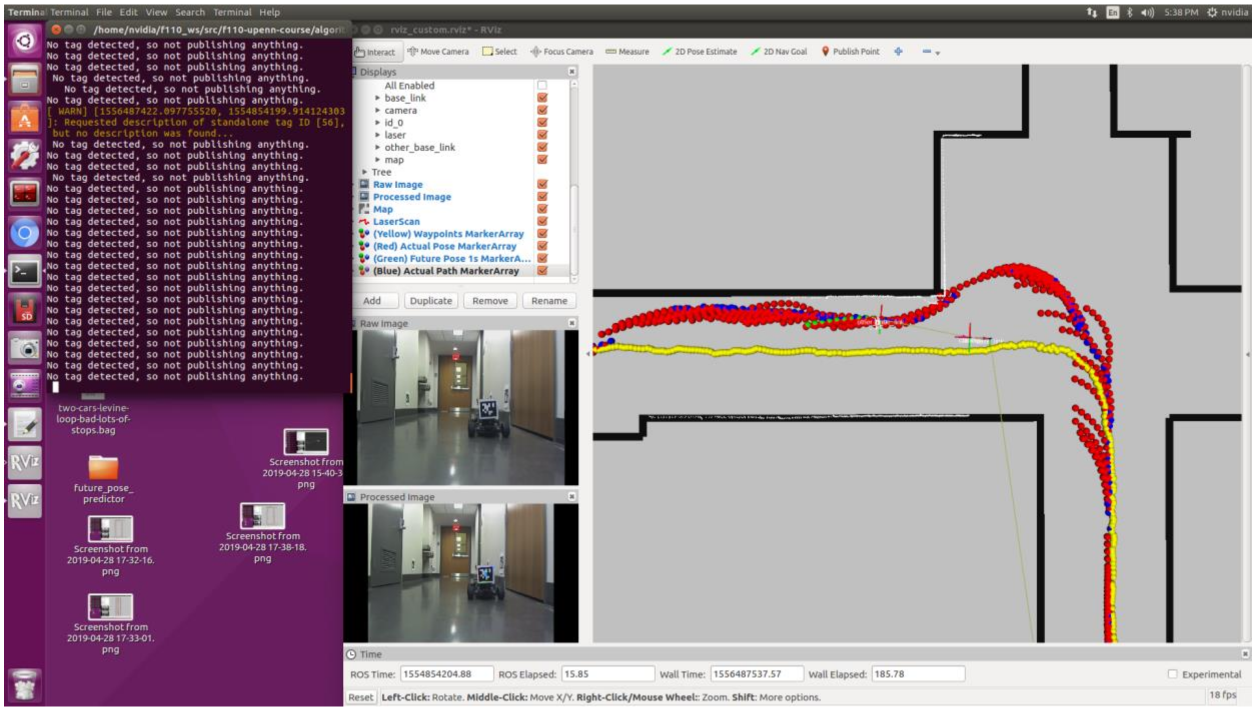
GREEN: Near future pose prediction algorithm's current prediction for where car will be 1 second in the future

BLUE: Actual Pose of the car (generated by saving all of the GREEN pose estimates over an entire loop). Note that even though blue in this case represents the actual pose of the car, it isn't the actual pose because it is still relying on AprilTag measurements, which as we showed in Experiments 1 and 2, has around 5-10% error. Moreover, the blue dots also relies on the particle filter measurement for the location of the car with the camera, based on lidar measurements, which also has some error. = YELLOW: This is just a set of waypoints that is used by the near future pose prediction algorithm to know the general direction that the car should be going in the map.

Notice that in general, it is good if RED and BLUE overlaps (because it means that predicted path was similar to actual path), and bad if RED and BLUE do not overlap. On straightaways there is a lot of overlap. On turns, there is less overlap. Images below will analyze the results.

Here is a video of the car going around the loop while predicting near future pose estimates of the car in front: https://www.youtube.com/watch?v=uQ1Bp-Bjlzk.

Case 1. Going straight while near the right wall


/home/nvidia/f110_ws/src/f110-upenn-course/algori
rviz_custom.rviz* - RViz
Displays
All Enabled
base_link
camera
id_0
laser
other_base_link
map
Tree
Raw Image
Processed Image
Map
LaserScan
(Yellow) Waypoints MarkerArray
(Red) Actual Pose MarkerArray
(Green) Future Pose 1s MarkerA...
(Blue) Actual Path MarkerArray
Raw Image
Processed Image
Time

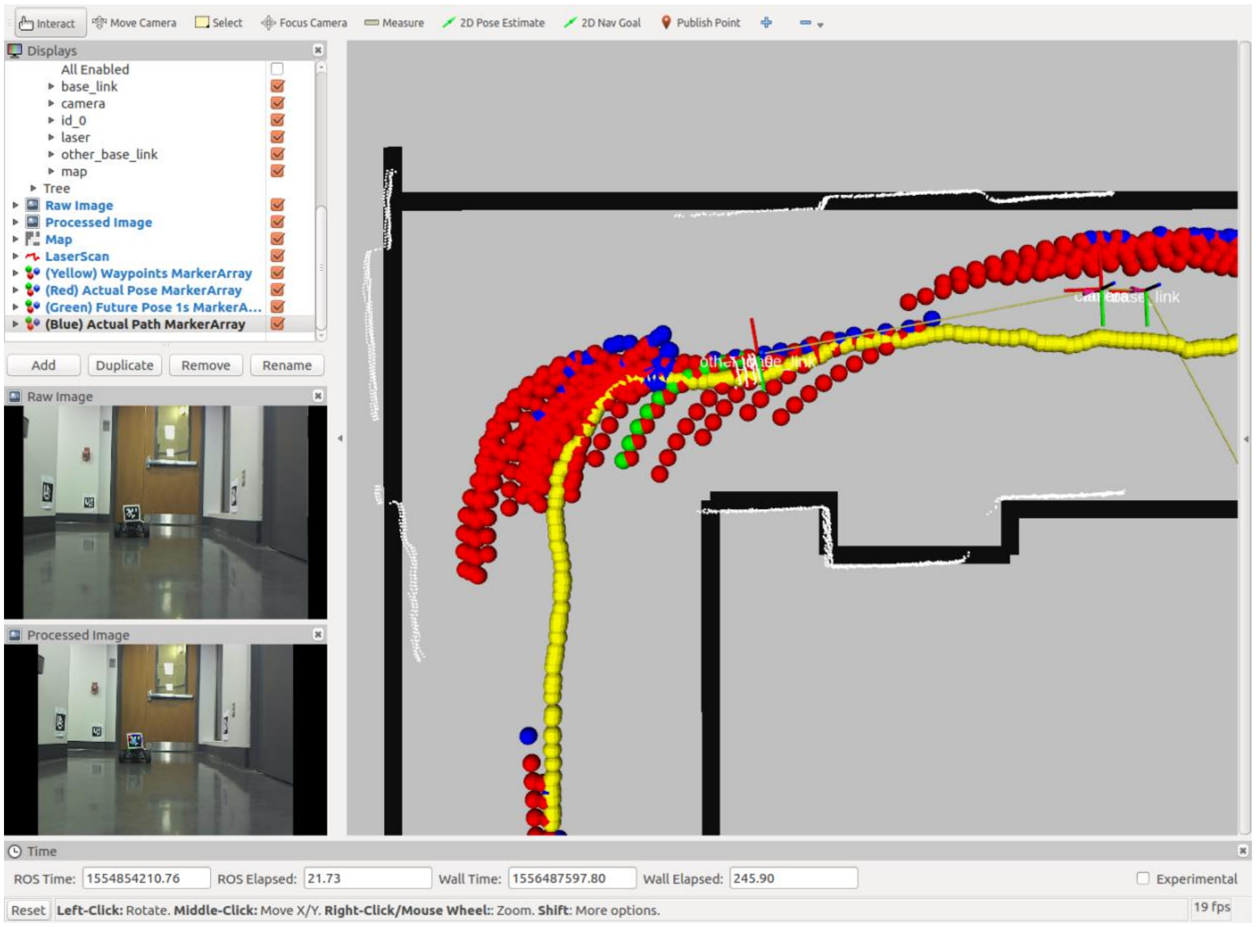
```
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
 No tag detected, so not publishing anything.
   No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
[ WARN] [1556487422.097755520, 1554854199.914124303
]: Requested description of standalone tag ID [56],
 but no description was found...
 No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
 No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
No tag detected, so not publishing anything.
```

ROS Time: 1554854204.88 ROS Elapsed: 15.85 Wall Time: 1556487537.57 Wall Elapsed: 185.78

Figure 14. Car going straight while near the right wall

This is a case where the car is going straight but it is located closer to the right wall. Notice that both cars here are moving from right to left, in a counter-clockwise fashion around the map. other_base_link represents the actual location of the car in front. This location of other_base_link is a transform between the other_base_link and map frames. Notice that the green dots represent where at this point of time, where the algorithm predicts the car will move. This prediction is clearly wrong because while the green dots bank left relative to the car, the blue dots (which represent the actual pose of the car) bank slightly towards the right. The predicted pose banks towards the center waypoints (represented in yellow). As we follow the blue dots which gradually get closer to the right wall, notice that at every time step the predicted pose is towards the left to get to the center of the path, which is constantly off until finally the car banks towards the left to initiate a left turn.

Case 2. Car preparing for a left turn



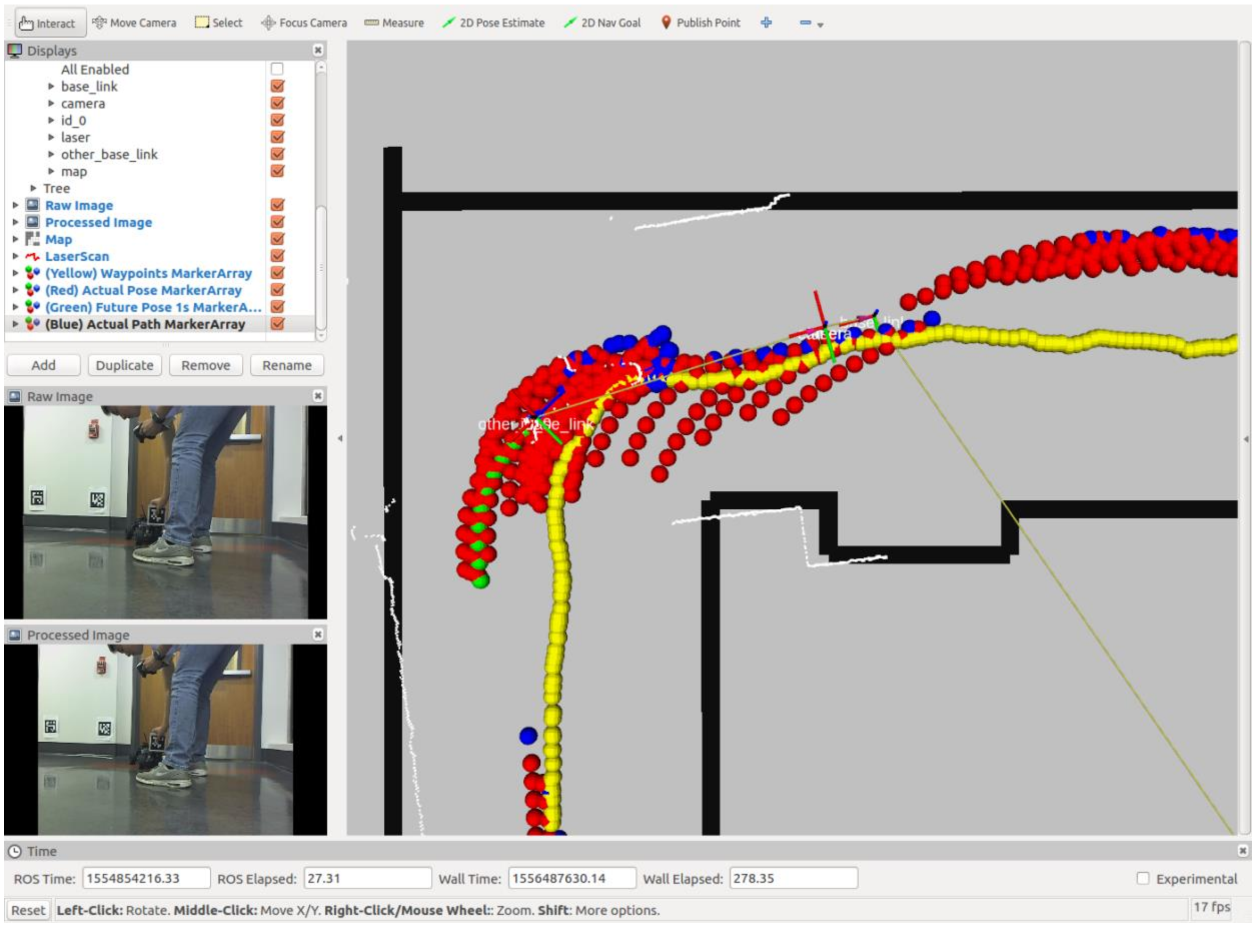
ROS Time: 1554854210.76 ROS Elapsed: 21.73 Wall Time: 1556487597.80 Wall Elapsed: 245.90

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options.

Figure 15. Car preparing for a left turn

Here, the car is approaching a left turn. If you watch the video linked above, you'll notice that the car stops because I had to adjust the AprilTag on the back. I decided to leave this in because it could simulate a car which may suddenly stop, and offers a test case for if someone walks onto the track. Notice that for as far as a meter before the car actually makes a left turn, my algorithm predicts that the car will begin turning left. All of these are, for the most part, incorrect. If you look at the blue dots, notice that the car actually turns later than it is predicted to. This could be due to the farther lookahead distance of 2 meters set in the pure pursuit prediction aspect of the algorithm. Setting a shorter lookahead distance may partially alleviate this problem for turns.

Case 3. Car making left turn later than expected



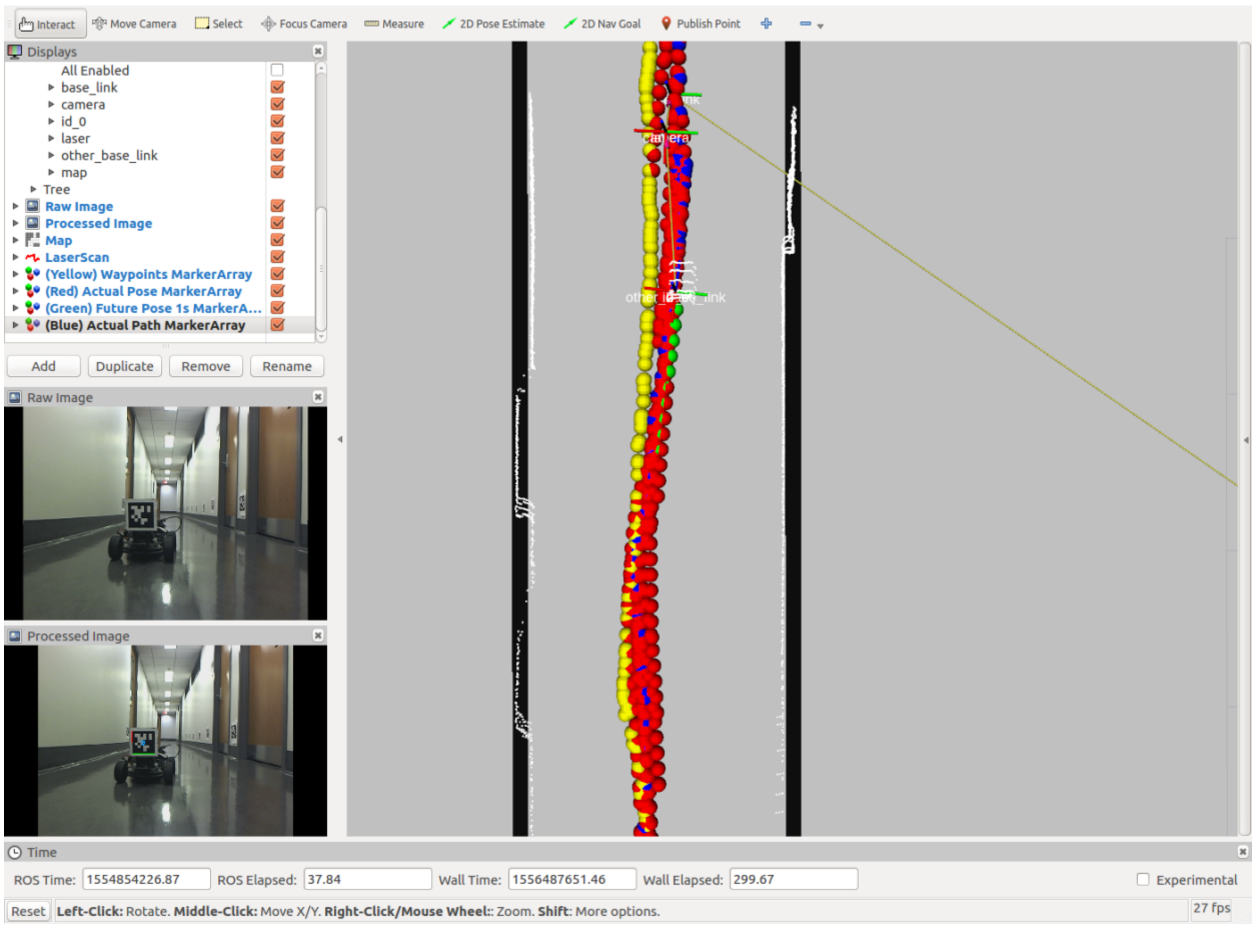
- All Enabled ☐
- base_link ✓
- camera ✓
- id_0 ✓
- laser ✓
- other_base_link ✓
- map ✓
- Tree
- Raw Image ✓
- Processed Image ✓
- Map ✓
- LaserScan ✓
- (Yellow) Waypoints MarkerArray ✓
- (Red) Actual Pose MarkerArray ✓
- (Green) Future Pose 1s MarkerA... ✓
- (Blue) Actual Path MarkerArray ✓

ROS Time: 1554854216.33 ROS Elapsed: 27.31 Wall Time: 1556487630.14 Wall Elapsed: 278.35 ☐ Experimental

Figure 16. Car making left turn later than expected

Here, the car finally makes a left turn, later than expected. Notice the trail of red paths from earlier predicting that the car would make a left, when instead the car continued moving straight. When the car finally does make the left turn, the AprilTag later falls out of frame, which is why there is a discontinuity in the blue dots, which represents where the car actual went.

Case 4. Car going down straightaway



ROS Time: 1554854226.87 ROS Elapsed: 37.84 Wall Time: 1556487651.46 Wall Elapsed: 299.67

Figure 17. Car going down straightaway

Here the car is moving from top to bottom of the image. other_base_link represents the car that we are predicting the future pose of. Notice that the green dots overlap the blue dots pretty well. This is a good thing. Because the car is generally in the center of the track, the algorithm predicts that it will follow the yellow dots (the waypoints which I generated to track the center of the race track). Notice also that as the blue dots oscillate slightly from right of center to left of center, the red dots also track this general motion. This is due to my algorithm taking into account the current yaw of the car and using a linear model that projects forward the car's current motion.

Case 5. Car initiating a left turn

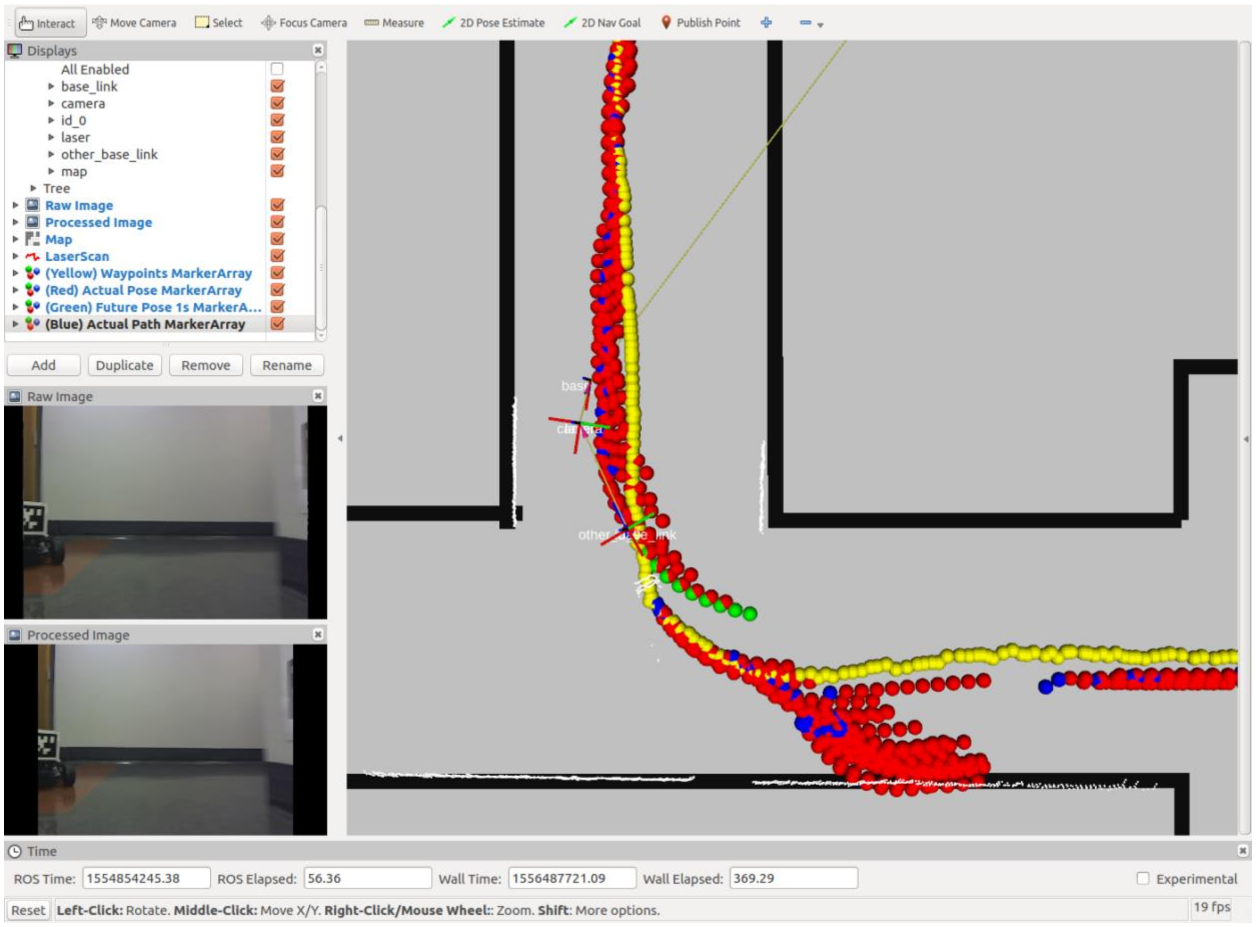
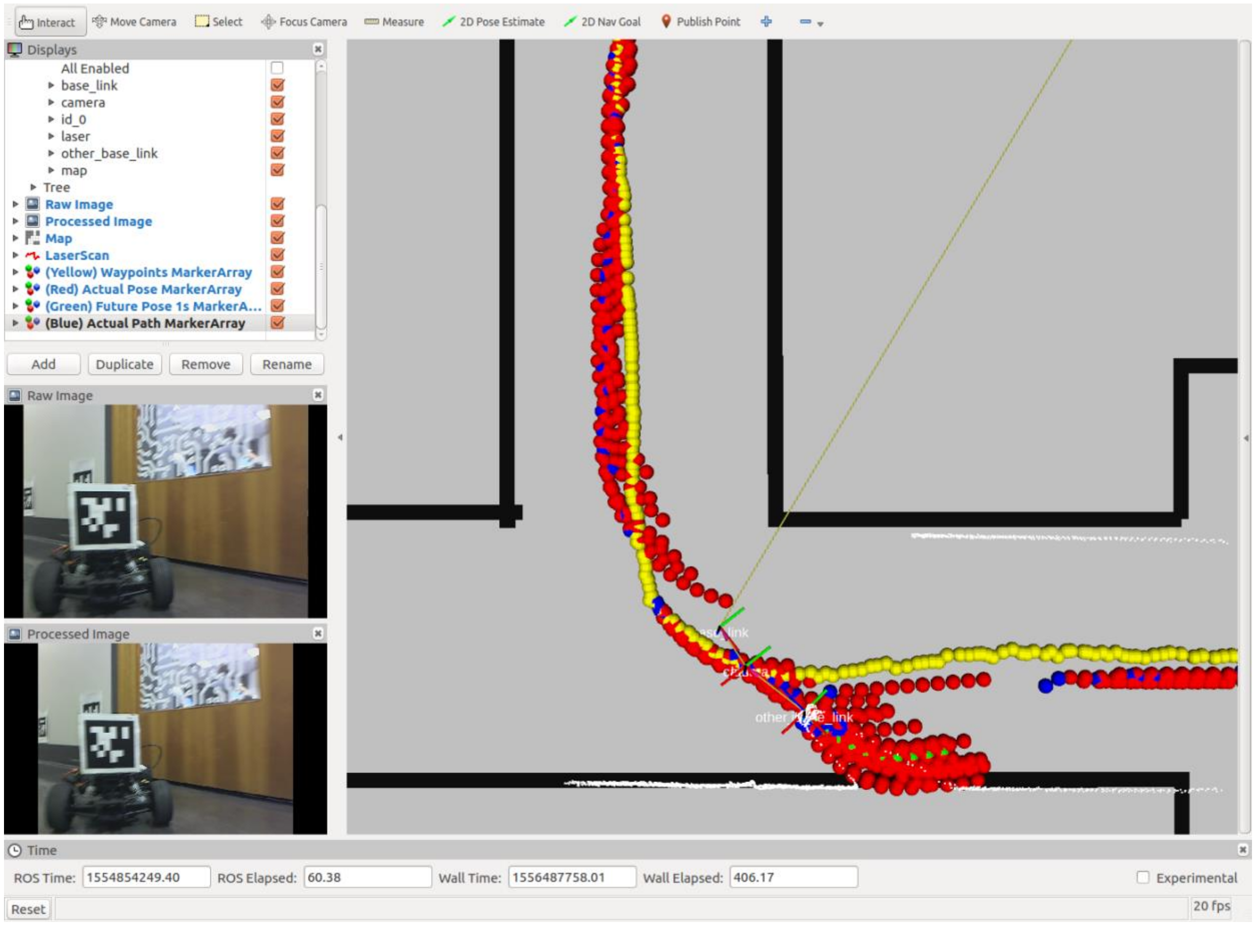


Figure 18. Car initiating a left turn

Here the car is initiating a left turn. The predicted trajectory at this point (in green) does not overlap with the actual future path of the car (in blue). Instead, the car continues straight for a bit more and makes a wider left turn. With future time steps, however, the algorithm adjusts and eventually predicts the correct trajectory. There is a chance that the car in front may have been trying to make a left earlier, but because of dynamics of the car such as drift, the car ended up making a wider turn. It is also possible that due to the roughly 0.4 second delay of the AprilTag, as was quantitatively measured in Experiment 2, perhaps the car is actually further ahead, and hence it is tracking the blue dots. I tried to analyze the videos, but because it is a 2D image with the car far away, it is difficult to judge the depth of the car. More analysis can be done on this in future work, for those who continue this project.

Case 6. Car makes left turn late

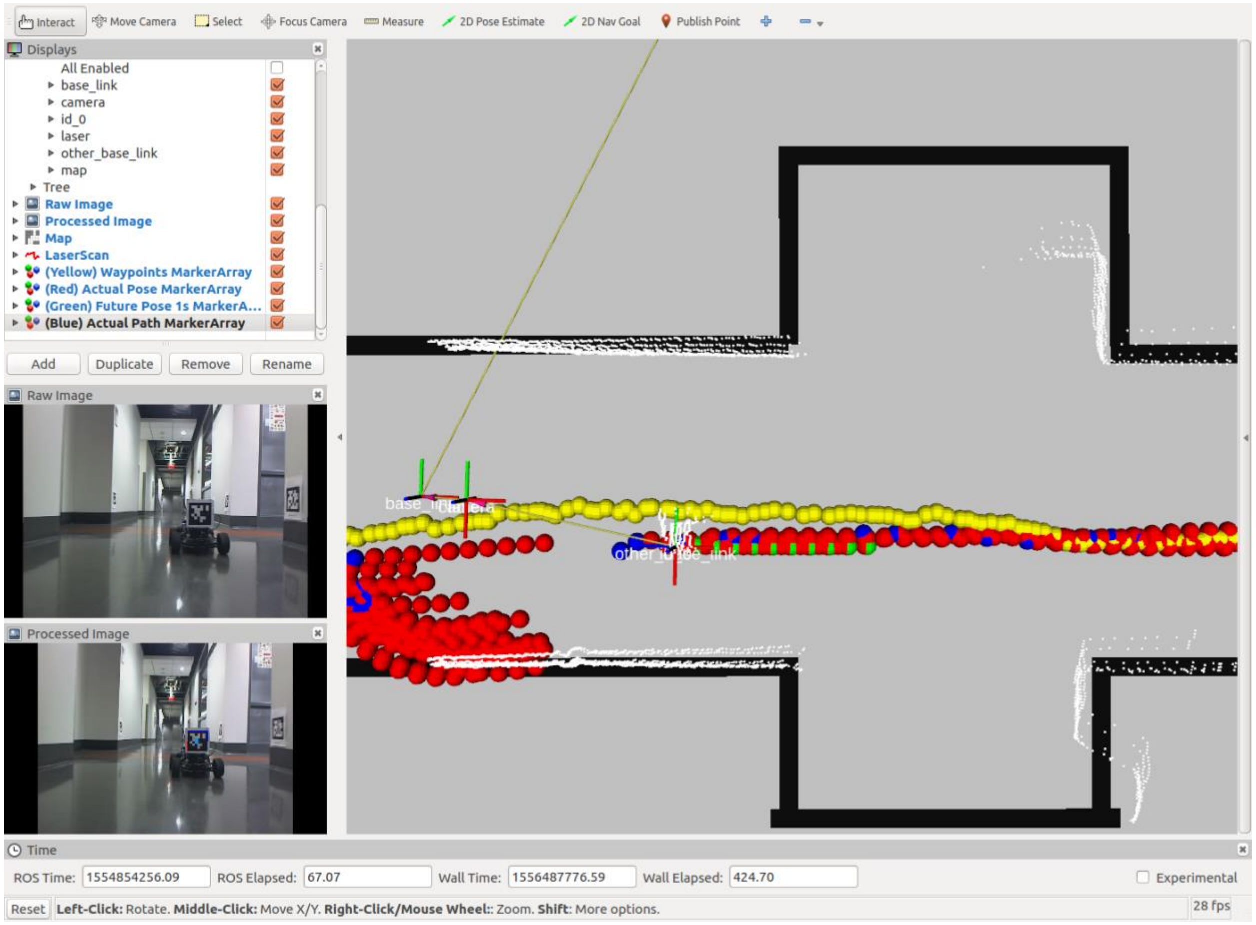


ROS Time: 1554854249.40 ROS Elapsed: 60.38 Wall Time: 1556487758.01 Wall Elapsed: 406.17

Figure 19. Car makes left turn late

It turns out that the car makes the left turn pretty late. Here, the prediction is accurate. The green dots follow the blue dots. Although we can't see the blue dots continuously here (because the car's AprilTag quickly falls out of frame of the camera), it's clear that the green predicted pose tracks the blue actual pose. Note that if we were using a simpler algorithm in this case, such as a linear model which just projects forward the car's current orientation, the resulting predicted pose would have been a straight line through the right wall.

Case 7. Car going straight

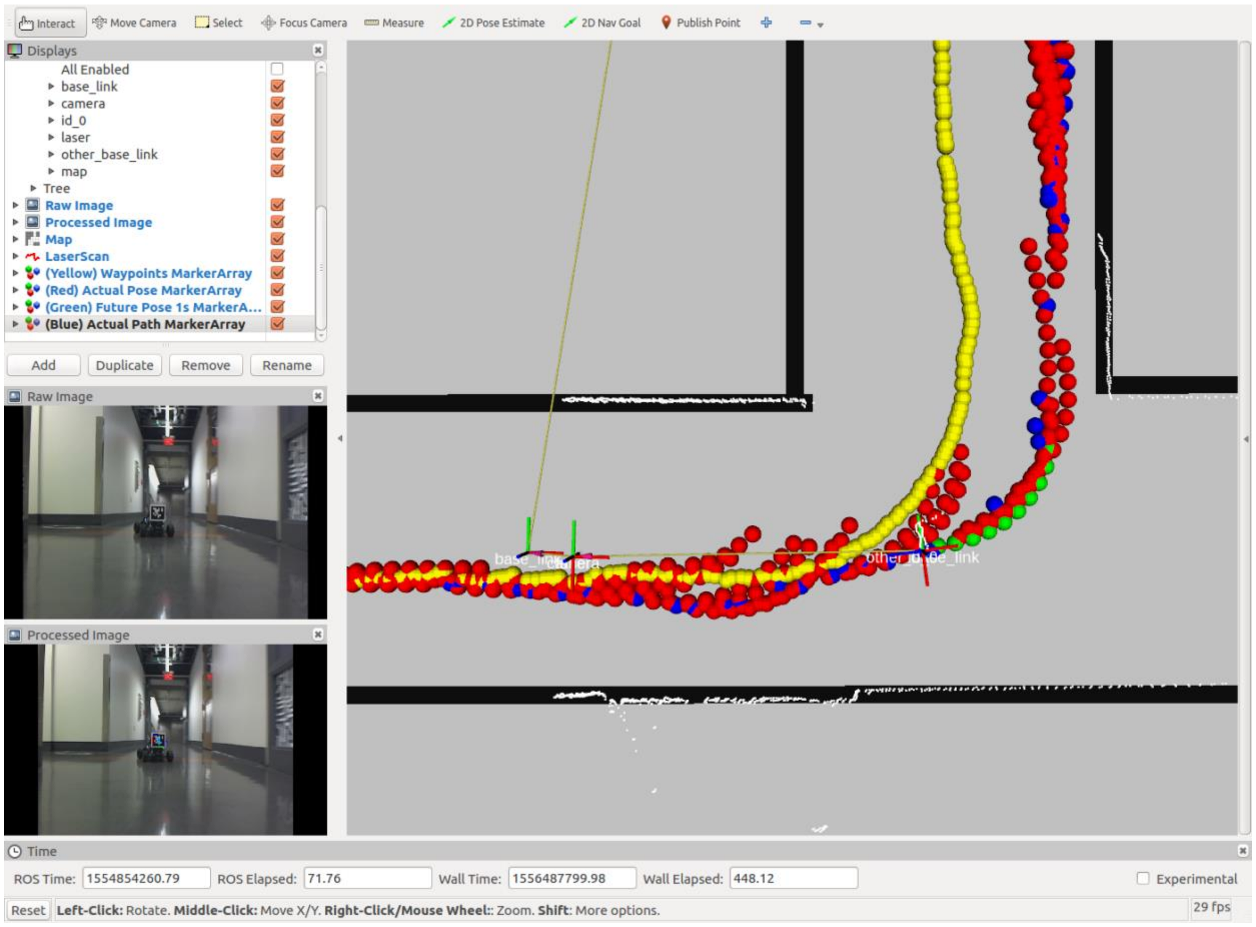


ROS Time: 1554854256.09 ROS Elapsed: 67.07 Wall Time: 1556487776.59 Wall Elapsed: 424.70

Figure 20. Car going straight

This looks pretty accurate. Blue dots overlap with red dots.

Case 8. Car making third left in the map



ROS Time: 1554854260.79 ROS Elapsed: 71.76 Wall Time: 1556487799.98 Wall Elapsed: 448.12

Figure 21. Car making third left in the map

This left turn looks pretty accurate. The blue dots track the red dots. Notice that there are elements earlier though where red dots predicts that the car should turn earlier, but the car did not turn that early.

Case 9. Car veering back towards center from right

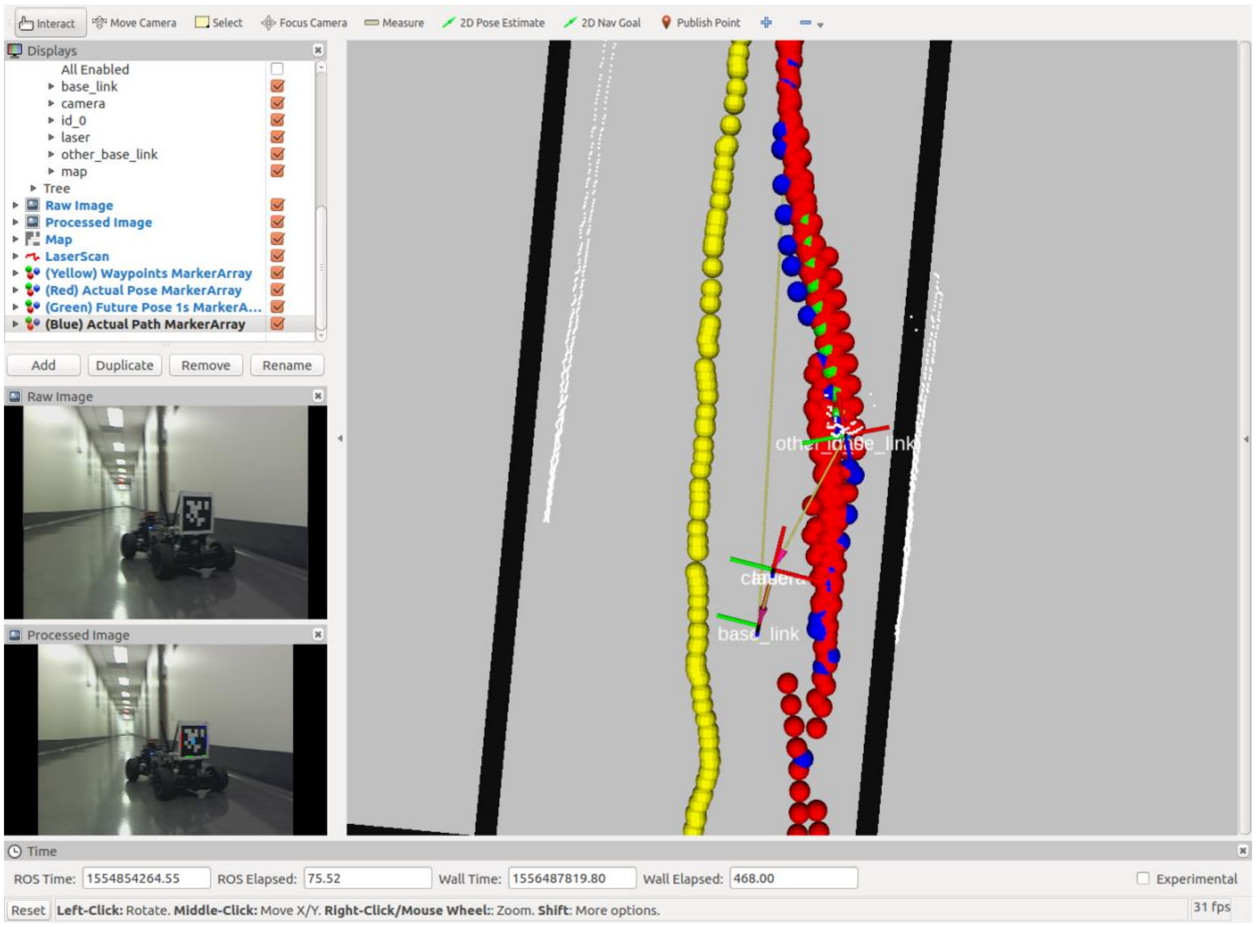
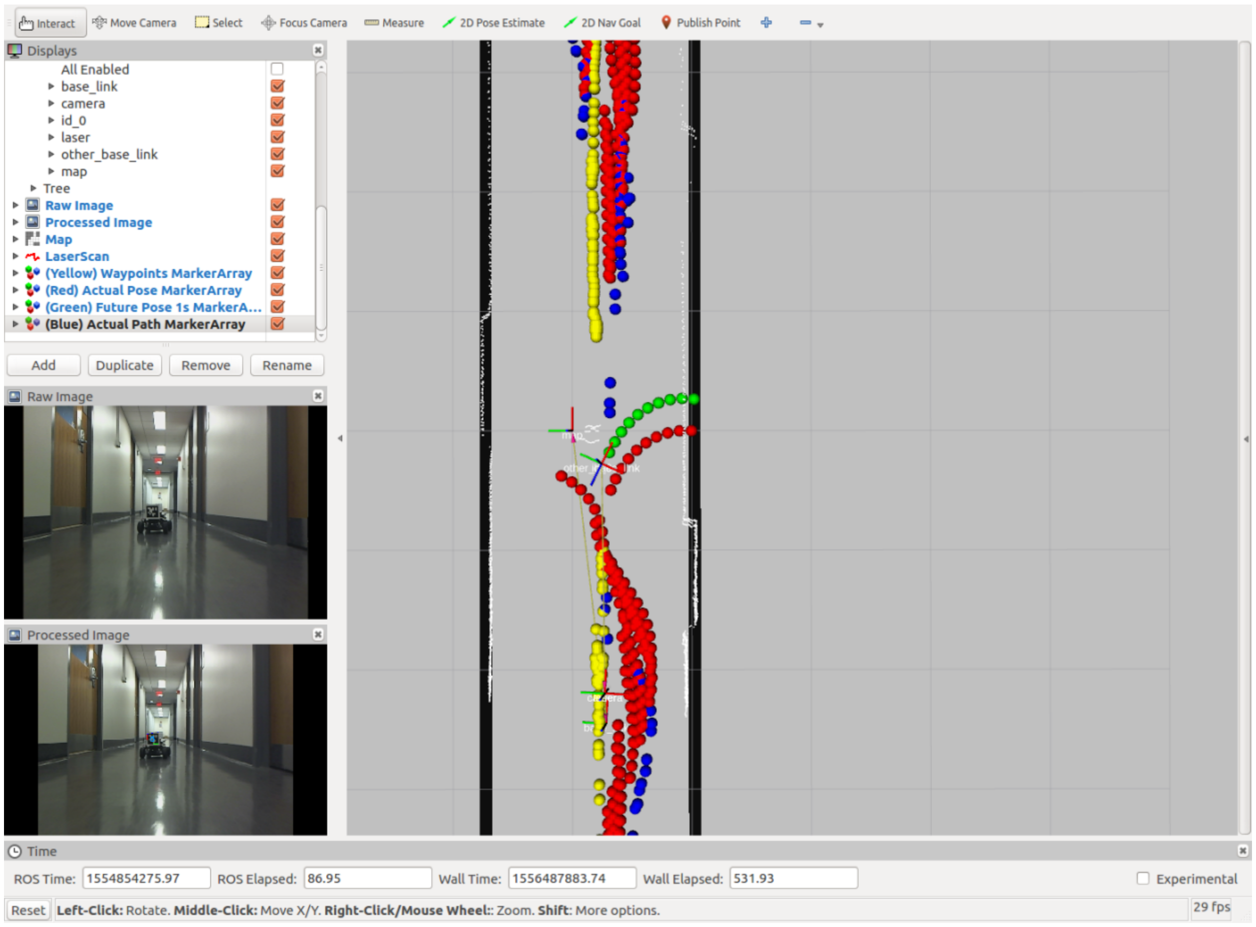


Figure 22. Car veering back towards center from right

Here the car is accurately predicting that it will veer back towards the center.

Case 10. Edge case where car wants to loop around



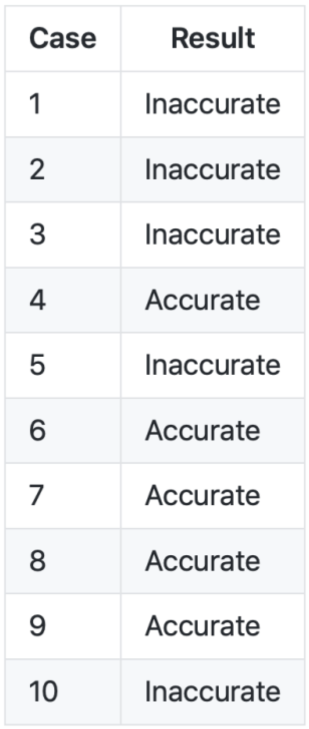
ROS Time: 1554854275.97 ROS Elapsed: 86.95 Wall Time: 1556487883.74 Wall Elapsed: 531.93

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options.

Figure 23. Edge case where car wants to loop around

The green dots here indicate the algorithm predicts that the car will go into the wall? You must be wondering what is going on here. This is an edge case. Because my algorithm uses pure pursuit as part of its weighting process, and because pure pursuit thinks that the closest yellow point is the one behind the car (as opposed to the one in front of the car), the car attempts a hard right turn in order to loop back to the point behind it. This is only a momentary problem, because as soon as the car advances forward another few centimeters, it picks up the point in front and makes an accurate prediction. I thought it would be important to explain this edge case here for the accute readers who picked this up.

To summarize Experiment 3, here is a table of which results were good (predicted path tracked actual path) and which were bad (predicted path did not track actual path).



Case	Result
1	Inaccurate
2	Inaccurate
3	Inaccurate
4	Accurate
5	Inaccurate
6	Accurate
7	Accurate
8	Accurate
9	Accurate
10	Inaccurate

Table 5. Summary of which cases were accurate or inaccurate. Roughly half the cases are accurate.

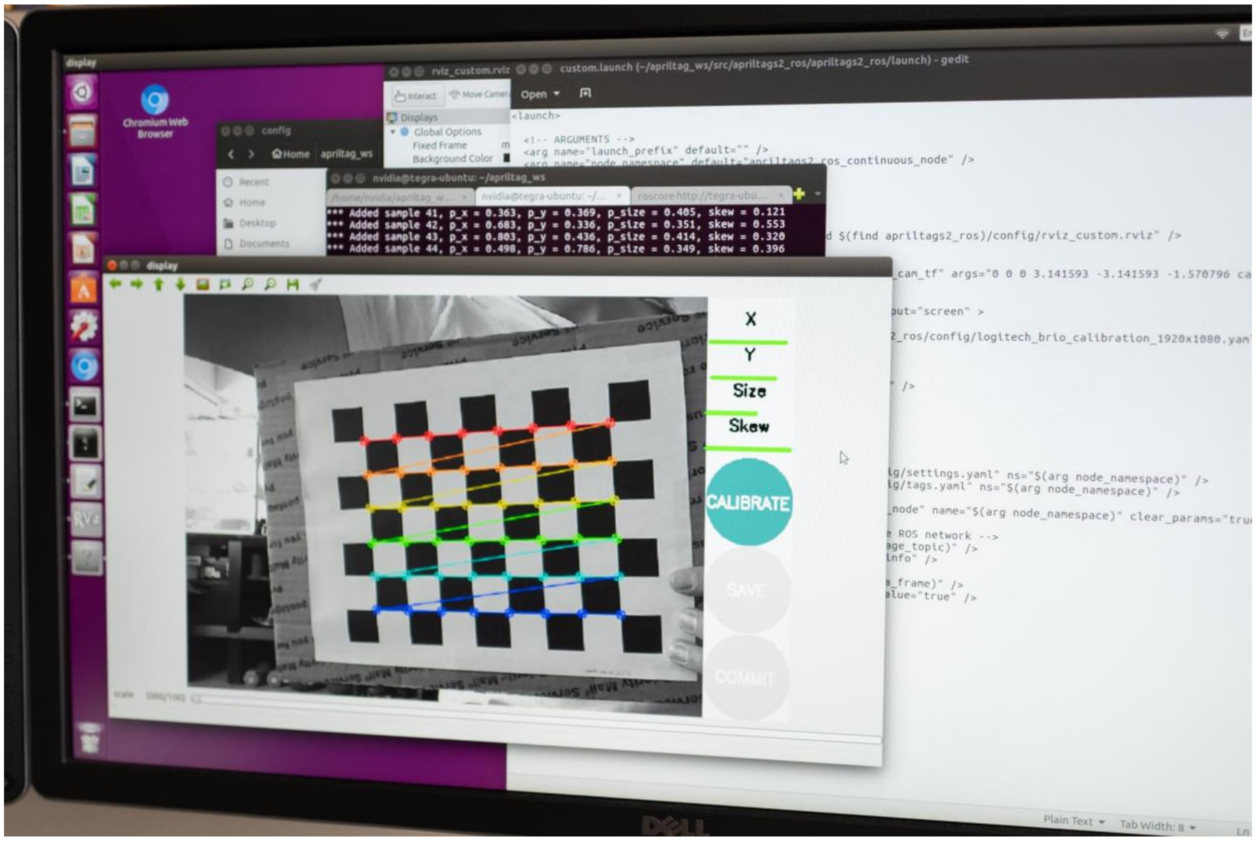
How to Test on Different Tracks/Maps
Chances are that if you are running this code, you won't be in the same location and will have a different track to run on. If you want to run the code as is with minimal configuration, you can run from the bag file by doing "roslaunch future_pose_prediction bag_future_pose_estimation.launch". If you would like to use your own set of waypoints and map, here is where you would change them.

1. In the /maps folder, add your own map.
2. In the /waypoints folder, add in your set of waypoints. You can record a set of waypoints while driving your car around your map and using particle filter localization using the waypoint_saver package inside of the f110-upenn-course repository which is linked near the top of this README.
3. In the /launch folder, modify the launch file so that you specify the updated map name and the initial pose for the car within the map.

How to Set Up a Different USB Webcam
Chances are that you do not have the exact same USB webcam I have which is the Logitech C910. (If you do, hey cheers to you!) Follow instructions from the ROS camera_calibration package: http://wiki.ros.org/camera_calibration.

1. You will need to print out a checkerboard pattern, which is linked on the page.
2. Mount the checkerboard pattern on a piece of cardboard or equivalently flat, hard surface.

3. Run the command described on the website that starts with "rosrun camera_calibration cameracalibrator.py ..." and do the calibration process.
4. You will see a screen that looks like this:



```
*** Added sample 41, p_x = 0.363, p_y = 0.369, p_size = 0.405, skew = 0.121
*** Added sample 42, p_x = 0.683, p_y = 0.336, p_size = 0.351, skew = 0.553
*** Added sample 43, p_x = 0.803, p_y = 0.436, p_size = 0.414, skew = 0.320
*** Added sample 44, p_x = 0.498, p_y = 0.786, p_size = 0.349, skew = 0.396
```

5. Once calibration is complete, find the .yaml file for the calibration and copy it into the /config folder from this repository.
6. You will need to modify the launch files in the /launch folder such that they load your calibration file and also so that the resolution matches your camera.

Opportunities for Improvement

1. As you can see with the results above, in around half the cases the algorithm inaccurately predicts the near future pose of the car. The case I see with most inaccurate predictions is when a car goes into a left turn, but ends up turning later. The algorithm predicts that the car begins turning a bit earlier than if it were staying in the center of the track. How might we be able to have an algorithm that predicts that the car makes wider turns? Perhaps playing with the lookahead distance, specifically by setting a shorter lookahead distance for the pure pursuit component of the algorithm, may address this problem. Or you may need to just rethink the algorithm, so that it considers what the car did in the past. Currently the algorithm does not consider past behavior of the car.
2. Often times the AprilTag falls out of the frame of view of the camera. Currently my Logitech C910 is set to record at 640x480, which is not a widescreen resolution. This is a

1.33 aspect ratio, whereas typical HD formats like 720p and 1080p are 1.77 aspect ratios. Hence, with the current 640x480 resolution, the AprilTag will fall out of frame when the AprilTag moves too far to the left or right. I did try using 720p, but the problem was that the AprilTag rate was only 4Hz which was unusable, compared to 12Hz from 640x480. I did also try other 1.77 aspect ratios of lower resolution such as 850x480, but any resolution besides 640x480 led to entirely incomprehensible images that were a bunch of random horizontal lines. I think that the resolution is locked at the physical level. When looking at the Logitech C910 website specs, it does list 640x480 as one of the native resolutions. It would be helpful to get some type of webcam that can support lower resolution at wide aspect ratio.

3. What happens if our car begins to pass the other car from the side? Then clearly the AprilTag will fall out of field of view. This is arguably the most important aspect of passing where we need to know the other car is. How might this be solved? Could we mount 2 different USB webcams? Do we mount 3 AprilTags on the car in front? One on its rear, and one on each side? Each AprilTag would have a different id and we would hard code the locations of each id relative to that car. Is it even possible to use a 180 degree or 360 degree camera? And process the image in equirectangular format?

Special Thanks
Thank you to Professor Mangharam and Professor Taylor for advising me on my senior thesis. Also thank you to Matt O'Kelly for providing insight into the weighted averaging algorithm used, and for his many calls and Slack messages and in-person meetings since last summer with answering my F1/10 questions.

I spent summer of 2018 learning about F1/10 cars for the first time, while developing a new ESE 680 course on autonomous racing which I assistant taught in fall of 2018. Then in spring of 2019 I worked on this project.

References

Althoff, Matthias. “CommonRoad: Vehicle Models.” 2018. https://commonroad.in.tum.de/documentation/vehicle_model_doc/

Coulter, R. Craig. “Implementation of the Pure Pursuit Path Tracking Algorithm.” 1992.

Grabner, Alexander et al. “3D Pose Estimation and 3D Model Retrieval for Objects in the Wild.” 2018.

Lengenfelder, Christian et al. “Low-cost and retrofittable pose estimation of rigid objects using infrared markers.” 2018.

Nahangi, Mohammed et al. “Automated Localization of UAVs in GPS-Denied Indoor Construction Environments Using Fiducial Markers.” https://www.iaarc.org/publications/fulltext/ISARC2018-Paper023.pdf

Olson, Edwin. “AprilTag: A robust and flexible visual fiducial system.” 2010.

“Real Time pose estimation of a textured object.” OpenCV Website. https://docs.opencv.org/3.1.0/dc/d2c/tutorial_real_time_pose.html

Savarese, Silvio & Fei-Fei, Li. “3D generic object categorization, localization and pose estimation.” 2007.

Wang, John & Olson, Edwin. “AprilTag2: Efficient and robust fiducial detection.” 2016.