Aido 3 - LF challenge

Pure Pursuit (PP)

By

Tobi Carvalho

and

Philippe Marcotte

Contents

1 Abstract

In the context of the AIDO-3 we had to compete in the Lane Following challenge (LF). The controller we used was a pure pursuit controller with a few tweaks, which seems to be working really well, ranking 11 in the LF test-simulation challenge, first in the LF real-validation challenge (as of December 15th), first at the class competition and second at the NeurIPS 2019 live competition.

2 Methodology

During the HW2 we had to implement a Pure Pursuit (PP) algorithm and make it work on our robot. To do so, we implemented the PP algorithm using a few assumptions to make it more stable and effective. In this section we will cover the list of assumptions.

2.1 Follow the yellow line!

First assumption made was for helping the robustness and stability of the PP algorithm by basing it on criterion's that are more constant. Therefore, we decided to base our pursuit point primarily on the yellow lines and use the white ones only if they are the only ones visible. This is different from what could be seen as the more intuitive solution which is to look at the middle between the yellow and white lines. As can be seen in figure 1, the road used during the competition was delimited by two white lines and a dashed yellow one in the middle. Hence, the center of the forward lane is always to the right of the yellow line, while it could be either on the right or the left of a white one.

Using the center between the yellow and the white line can then lead to confusions since it is unclear relative to which of the white lines we are going to be looking. One of these possibilities leading directly to going in the middle of the incoming lane. We did tried to use some conditions to differentiates the whites lines, but we found out that all our attempts were confusing and unreliable, specially when turning a corners.

Using the, more intuitive, approach to use both yellow and white lines to determine the pursuit point, there is a total of 4 combinations of line that have to be taken into account. Depending on its orientation and position, the robot can either see both yellow and white lines, yellow lines, white lines or no line at all. While focusing primarily on the yellow lines reduce the number of possibilities that have to be

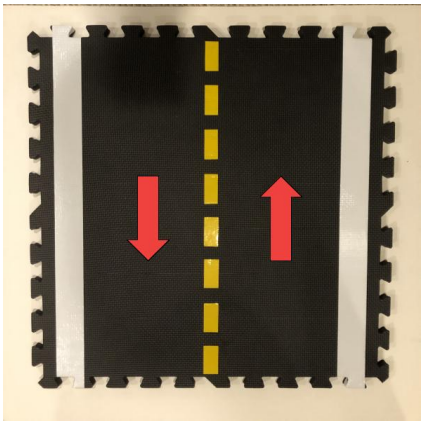

Figure 1: Road example

cover to only 3 (is there yellow lines, is there only white line and is there no line at all), making it more stable since the change of the lookup point isn't smooth while shifting from one situation to the other.

Finally, an offset in the y direction was added to the lookup point since we don't want to be on the yellow line, but in the middle of the lane.

Our strategy was then to first look if there was any yellow line and, if it was the case, to add an offset to the right corresponding to half the size of the lane. If no yellow line were detected but white ones were, we would focus on those and add the same offset but to the left. If no line were seen (which should not happen), we would simply continue going forward but at a slower pace hoping we would end up seeing lines again.

2.2 Don't look too far, but don't look too close!

The second assumption made was to not give the same importance to every segments. It is well know that the behaviour of PP algorithm is deeply dependent to the look ahead distance. Indeed, if it is too big, we risk detecting and focusing on object that are outside of the track or to turn corner too sharply, both resulting to going off road. On the opposite end, if is too small, we risk to oscillate from one point to the other, resulting in a poor driving. Even worse if we are subject to latency, it could lead to driving past the pursuit point which would also result in falling off the road. For this reason, we usually define a fixed look ahead distance. But in our case, this could be detrimental to the robot since we don't have any guarantee for any line to exist at this distance. To address this, our strategy was to define a look ahead

distance and to weight the segments relative to it. As in equation 1, the weighting is proportional to the inverse of the square of the difference between the segment's distance and the look ahead distance plus one (in order to prevent the coefficient to explode and to smooth out the importance given to only one segment).

$$w_i = \frac{1}{1 + (d_i - L)^2} \tag{1}$$

This way, we give more importance to the segments that are close to the look ahead distance without loosing the capacity to work even if there is no segment at this distance.

2.3 Going faster by slowing down

Under the assumption that one of the factors preventing us to go faster, was the quick change of the layout relatively to the belief array and the location of the lookup point when the robot is taking a turn. We tried to allow our robot to slow down when it is taking a turn. In order to do this, we scaled the speed of the robot to the look ahead distance with a lower cutoff threshold. This mean that we allowed the robot to slow down as the resulting look ahead distance got smaller, but only to a certain point. Assuming that the look ahead distance got smaller when it is in a turn, this would allow the robots to go slower when taking one, making it more stable by letting the belief array get corrected fast enough by the image to compensate for the variation of the layout.

In practice though, we realized that the robot was going most of the time at it's minimal allowed speed, suggesting that the look ahead distance might have been a little bit too big. Otherwise, we also observed that whenever we tried to increase the default speed over certain threshold, the robot went from driving well to driving terribly bad right away. We are uncertain of the exact cause, but we suspect it could be related to the update of the belief between new images. While doing the homework on the Particle filter, we realized that the calculation of the update of the belief array was overshooting the rate at which the robot was turning and driving. This problem was probably caused by the algorithm using the time of the computer instead of the time of the simulation, which shouldn't be a problem when it is executed on the robot. However, the fact that we never calibrated the gain on our robots, only the trim, could most likely lead to a similar problem. If the gain isn't calibrated, there is going to be a mismatch between the speed we send and the actual speed of the robot, leading to a bad update of the belief array between the images. We think that this could explain why we couldn't go beyond a certain speed

since there could be a threshold over which the image update frequency is not high enough to compensate for the bad updates. We didn't look too much into it and it's all speculative, but it seems logical and might be worth investigating.