Rapid Autonomous Complex-Environment Competing Ackermann-Steering Robot

Parth Parekh

July 2016

Academy of Engineering and Design Technology

Hackensack, NJ 07601

Beaver Works Summer Program

MIT-Lincoln Laboratories Beaver Works

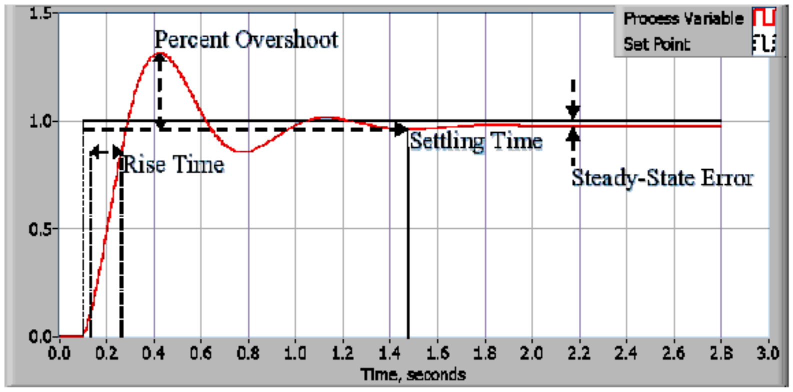
Computer Science and Artificial Intelligence Laboratory

Department of Electrical Engineering and Computer Science

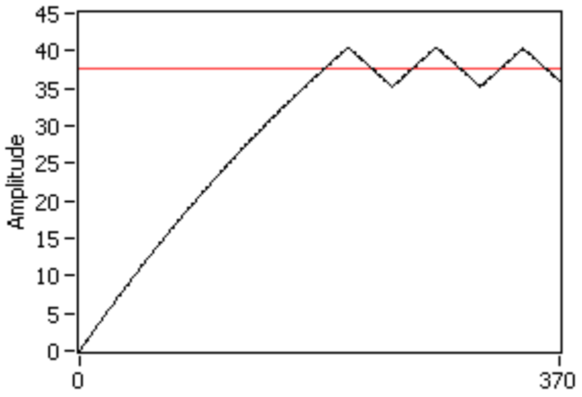
Department of Aeronautics and Astronautics

A. Introduction to Ubuntu, ROS, and Control Systems

The technical goals for the first week included properly installing VMWare Fusion, learning how to use basic ROS commands such as `roslaunch`, `catkin_make`, and `rostopic echo`, and using editors more varied than Gedit, such as Nano and Vim. The challenge for the end of the week was to have the robot follow a wall, first on the left side, then on the right side, and lastly, in the middle. To do this, most of the teams implemented a simple PID controller[1], with the only other option being a relatively primitive bang-bang controller[2]. The team chose a PID controller to reduce the steady-state error over time.



(a) The PID controller eventually converges to a steady-state error due to it's ability to calculate new error values, which the bang-bang controller to the right lacks.



(b) The bang-bang controller can only switch abruptly between two states, so it does not calculate an error value.

Figure 1

For this specific Friday challenge, only a PD controller was used because the integral part was not deemed necessary. The integration of the error would also be the most difficult step and the one which required the most calibration, and as everyone was already inexperiences and having difficult adapting to the Ubuntu environment, many chose to forgo it as well. Thus, rather than the standard PID equation, we used the following:

$$u(\mathrm{t}) = K_p e(\mathrm{t}) + K_d \mathrm{d}' e(\mathrm{t}) \qquad \text{Eq. 1}$$

The values we obtained from the LIDAR scan were those between 30˚ and 45˚ and 135˚ and 150˚, assuming the 0˚axis to be where the scan begins. We took the average of the distance from the nearest object between those 30 points and set a distance from the nearer of the two. That way, we could adapt to both the right and left wall without any problems. However, the PD controller worked well, requiring only about 1.8 seconds to reach its settling time, and running with a small steady-state error from there on out. This was true for both the left- and right-wall following tests. During the center test, the percent overshoot was much larger than in either of the other scenarios, but it still righted itself and had a settling time of 2.0 seconds, and having the same steady-state error as the two previous runs. Overall, the goal of this first week was to learn how to use the basics of ROS and Python and to start implementing basic algorithms. The PID controller, as it turned out, was a relatively basic control system; when we moved into the third week and began looking at motion planning systems, we learned how to implement potential fields to improve upon the wall-follower program.

B. Implementation of ZED Camera

This week began with learning how to take `.rosbag` files from the ZED Camera on board the car. A monitor running native Ubuntu was also given to all the teams to run updates on the car and to provide a larger screen to view of a playing video or a node map. In addition, OpenCV2 was downloaded on every student's virtual machine, and with it, basic image detection labs were run. These included identifying a red blob hanging beside four blobs of other color and then drawing a box around it while printing its color or size to the file. After going through this basic lab and learning some other simple commands in OpenCV2, our next Friday challenge was given: to identify a blob sitting at an intersection as either red or green, and taking a right turn if it was green or a left if it was red. Then, the wall would curve around and the robot would have to run a wall-following code to follow it. This task was made difficult by several factors; these included the fact that the vehicle has a large turning radius and has to begin making the turn several feet away from the blob, and also that the wall-follower would need to be able to follow walls and different curves on both sides. In addition, the robot would have to be running a simple safety controller that would automatically stop the robot if it got too close with colliding with a wall. Based on these guidelines, a list of requirements was made. The final product would need to include a PID controller, an image detector, a color detector, and a safety controller. Nodes including the Ackermann Drive, Ackermann Safety, LIDAR, ZED camera, and joystick teleoperator would need to be imported in. Based on that, the following node map was created:

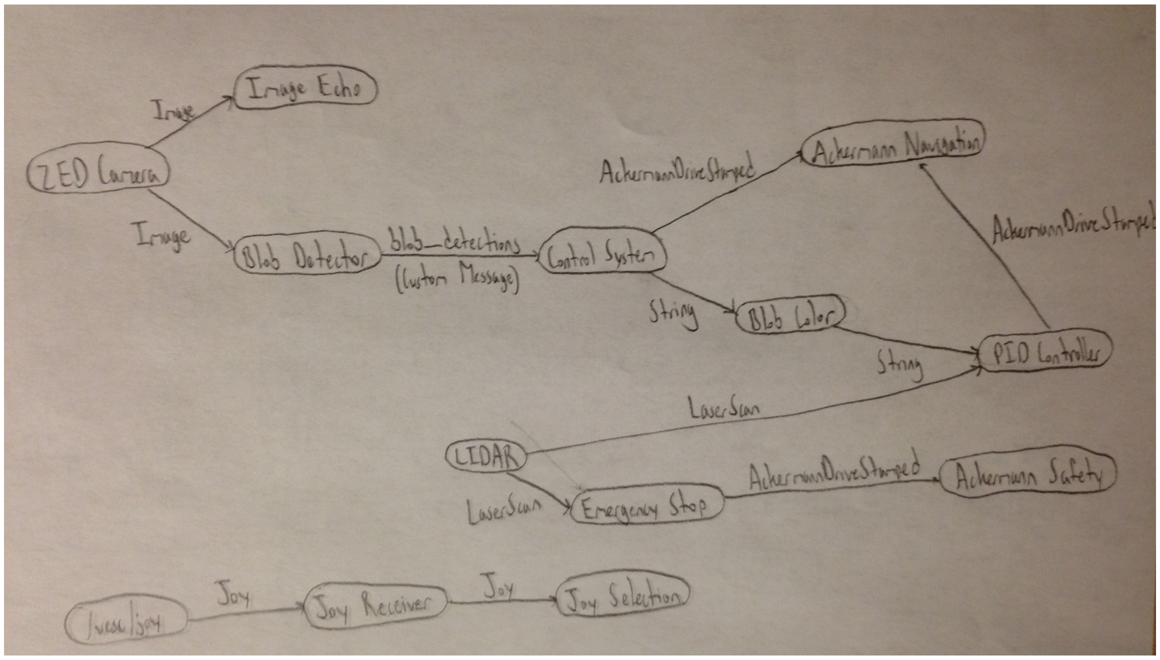


Fig. 2: Our final implementation consisted of the following 13 nodes, with the PID controller, Blob Detector, and Control System nodes being the most significant.

Again, a PD controller was used instead of the full PID-controller, as most of the time needed to be dedicated to finding a blob, deciding its color, passing it to a control system to make the correct turn, and then letting the PID, which could already be trusted, do the rest. Deciding the color proved to be the most difficult, as we used the red-green-blue color scale (RGB) to determine the color of the obstacle. In different lighting situations, this value would change, so defining a set of upper and lower bounds broad enough to cover most of them while simultaneously being narrow enough to discount other possible sources of green color, such as a light blue shirt someone in the audience is wearing, or a yellow piece of tape on the ground. Our team determined the following values would be sufficient:

Table 1

	RGB
Red	0, 190, 100
Green	35, 100, 200
Yellow (Light Green)	20, 200, 150

As the chart shows, the yellow color was also added to this program. This was because two different shades of green could be the possible blob, so one shade, which was closer to yellow, was named as such.

In the end, the goal was very nearly met. The robot saw the blob, observed the correct color in all three instances (red, green, and light green), and made the correct turn; however, the PD controller would fail when trying to follow the curving wall. That test proved that all the

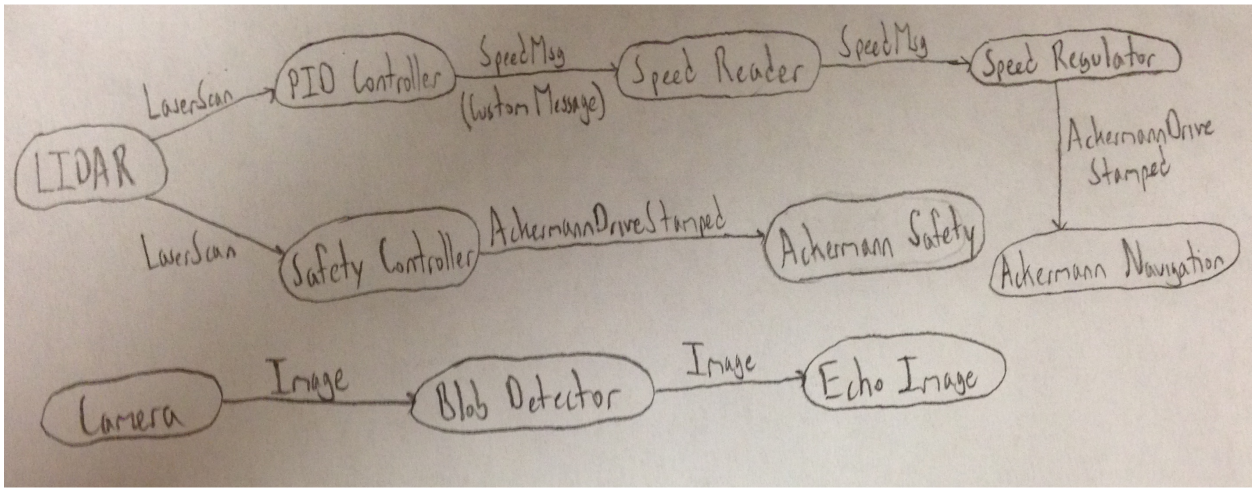


Fig. 3: This launch file, despite looking smaller, has only 3 nodes fewer than the last week's.

programs running need time dedicated to them to make sure they work and to properly calibrate them. In addition, it was learned that it was difficult to keep track of so many nodes at the same time. In addition, calibrating them all properly would take up even more time, and as many of them has similar variables, they could have been condensed into fewer nodes. It became apparent that the simplest solution would usually be the best one.

C. Utilization of Motion Planning Algorithms and Linear Algebra

The following week, motion planning algorithms such as potential fields were taught and implemented in place of the PID and PD controllers. These were capable of much more than simply wall following, so they were ideal for challenges such as exploring space or moving through an area with walls spread far apart from one another. While more complicated motion planning algorithms were taught and explained, such as A*, D* Lite, and RRT, they were deemed to be much too advanced to learn in a short period of time, and easily unneeded for the simple tasks our robots would be required to perform. As such, the algorithm that most teams turned to was the potential fields program. It was easily implementable, as it could fit within 12 lines of code, and certainly more stable and trustworthy than a PD controller. Thus, our challenge for the final week was to explore space while making as few collisions as possible, while detecting blobs and identifying their color at the same time. The blobs were placed at different angles and different points in the course, such as on the corner of a turn or to parallel to a wall. The team split into three different teams to explore this: the first team would work on the image recognition and identification; the second would work on modifying a PD controller to be able to explore space instead of follow walls; and the third, working solo, would try to implement the potential fields algorithm. While the image recognition was a program that would be needed no matter what the control system, the system itself was flexible. In the end, the potential fields did not work as expected, and the PD controller was finalized as the node on the second-to-last day. The node map for the code was then drawn up:

The Friday challenge for this week did not end well at all. A trapezoidal control for the speed was implemented on the last day, as a lot of gear slipping could often be heard when the code was running. This controller, not properly calibrated, ended up sabotaging the otherwise efficient PID control, and resulted in many more crashes and not much less gear slipping. While the image recognition worked fairly well, the trapezoidal control effectively increased our crashing percentage by 340%, as we had an average of 5 crashes before it and ended with 17 afterwards. The lesson learned from here was that a program should never be created and implemented in the last several hours without proper testing, as it has a high chance of ruining the positive effects of the previous code and is less likely to actually fix the entire problem. However, as stated before, the imaging worked well and detected four of the six blobs that passed by. It also correctly identified the color in each instance, as per the following RGB chart:

Table 2

	RGB (Lower Bounds)	RGB (Upper Bounds)
Red	0, 160, 130	15, 255, 255
Green	54, 30, 60	72, 255, 255
Yellow	40, 85, 50	55, 200, 200
Blue	200, 30, 30	220, 200, 200
Pink	300, 40, 80	310, 200, 200

As a result of the PD controller not working well at all, I looked the potential fields algorithm over the weekend and ran it with several different instances in Gazebo. Thus, when the team came in the following week, it was decided that the ruinous PD controller would be replaced by the potential fields, which was known in theory to be more reliable and which also worked very well on Gazebo, a world of perfect physics.

D. Potential Fields and Calibration of the Color Detection

In the final week, there were two technical challenges. The first one would be the same as the challenge of the last week, where the car would have to explore space and detect blobs while avoiding collisions. The second one would be the same as the challenge of the second week, where the car would have to turn right if it detected a red blob in front of it or go straight if it detected a green blob. However, there were certain caveats which essentially made them more complex challenges than at first blush. The image recognition problem with the first challenge would be that the robot would have to detect the colors in a different lighting environment, as the building for the final day was different. The same problem existed for the second challenge, but the effect was multiplied because the paper was hanging off a bar and not flush on a flat surface. Some of the light thus passed through it and gave it a completely different RGB value that needed to be found out and implemented in the code. Thus, new upper and lower bounds were defined:

These values were confirmed to work for the red and green blobs that were being used, but only in certain lighting situations. As the room had large windows and large amounts of natural light, and very little artificial light, the brightness was conditioned on the position of the sun throughout the day. The values above were decided between 10 and 11 P.M., as that was when the final race was scheduled to occur. These values would hold mostly constant between 10 and 4 P.M., but it would most likely fail to work before or after that time range because the sun is in a significantly different position and the light provided by the bulbs is not even close to the luminosity of natural sunlight.

Table 3

	RGB (Lower Bounds)	RGB (Upper Bounds)
Red	1, 120, 80	15, 255, 254
Green	48, 150, 75	85, 255, 255
Yellow	35, 140, 75	90, 255, 255
Blue	100, 50, 75	135, 255, 255
Pink	160, 20, 75	185, 255, 255

In addition, the final challenge for Friday involved the car needing to move around the racetrack shown below.

Thus, the robot had to be able to travel in between the two walls and would have to maintain that throughout turns in order to achieve the most efficient time. In addition, a shortcut

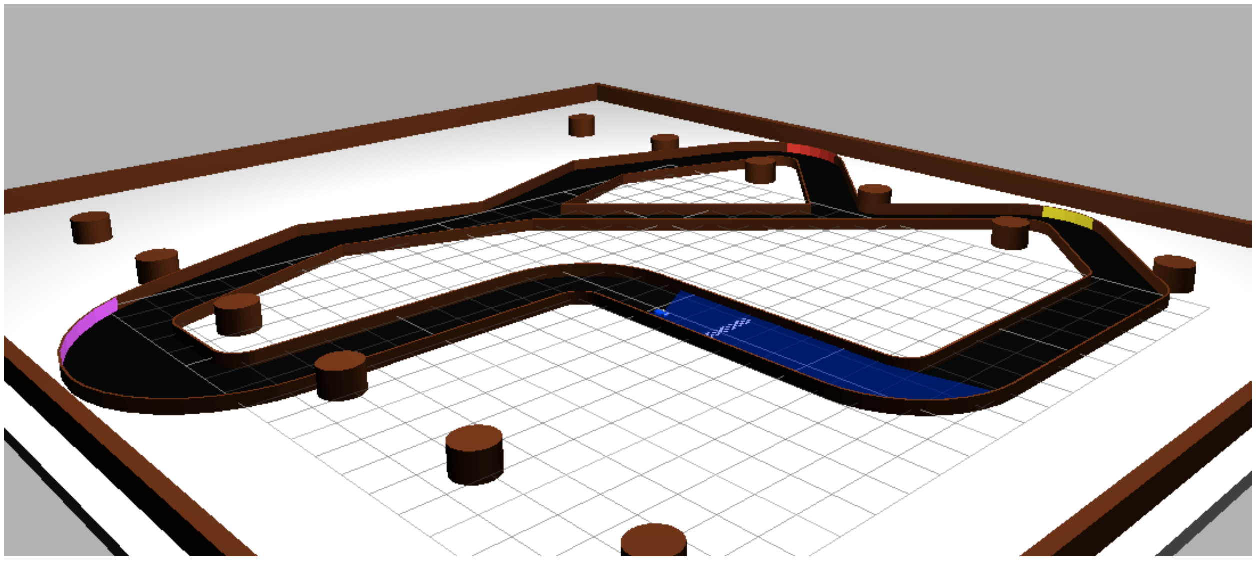


Fig. 4: The final racetrack involved walls made of cardboard, weaving through the pillars.

would be provided during one of the three runs, where a green blob hanging from a bar at the intersection would let the car now if the path was clear. If it was red, the path not immediately ahead would be blocked, effectively ending the robot's run if it could make a u-turn. If it detected a red blob, it would need to turn right to take a longer approach to the end mark. Thus, color detection would be of utmost importance here, as only our quickest time would be the one needed to decided the order of heats and winning. Detecting the green blob during one of the runs was of maximum importance to us.

Besides that, the team decided to remain with a purely potential fields approach to this racetrack. The sensitivity to walls was increased, so it would stay closer to the middle; in addition, the speed and steering angle being published the Ackermann Drive were increased, as the halls were longer and wider than the ones used in past challenges. Finally, the positive charge of the walls was decreased to reduce an oscillating motion in the robot, as it would be forced to turn away from a wall if it got too close. This originally did not work well, as it would still oscillate too much. However, eventually the problem was solved by using a derivative control in it similar to PD controller, where the error could be reduced over time, and our proportional error being the speed and angle message itself, instead of a constant value K_p multiplied by some proportion of distance.

Finally, an intermediary node was written to make the turn if a color was detected. The color would be detected by a five-point system, where if the program detected either a red or green color in five consecutive frames, it would assume that whatever contained the color was the blob and it could execute a turn. The potential fields algorithm would then publish to this intermediary rather than directly to Ackermann Drive. If the intermediary did not detect a color, if would just publish the message to Ackermann Drive, but if a color was detected, it would begin executing the respective turn.

The linear algebra behind the final potential fields controller is as follows:

```
scan_x = (self.charge_particle * scan_x_vectors) / np.square(msg.ranges) Eq.2
scan_y = (self.charge_particle * scan_y_vectors) / np.square(msg.ranges)

kick_x = (np.ones(1) * self.charge_forward / self.boost_distance**2.0)
kick_y_component = np.zeros(1)

total_x = (np.sum(scan_x) + kick_x) * 0.12
total_y = (np.sum(scan_y) + kick_y) * 0.12

self.last_total_y = total_y
self.error = self.last_total_y - total_y
total_y += self.kd * self.error

msg.angle = (self.steering * np.sign(total_x) * math.atan2(total_y, total_x))
msg.speed = (self.speed * np.sign(total_x) * math.sqrt(total_x**2 + total_y**2))
```

The `scan_x_vectors` and `scan_y_vectors` are the vectors taken of all 1080 LIDAR points and converted to Cartesian coordinates with the use of the `numpy.sin()` and `numpy.cos()` functions. These unit vectors are multiplied using variables declared in the main class, and divided by the square of `msg.ranges`, from the LIDAR scan. Then, a 'kick' for the car, by using all the x-unit vectors to provide a charge behind the car to propel it forwards. This charge is also determined by variables declared in the main class. Also, the kick's y-coordinates do not change, because it has to be directly behind the car always. Afterwards, variables `total_x` and `total_y` are declared, both of which determine the speed and steering angle commands sent to the Ackermann drive. Both of them are approximately an eighth of the sum of the kickback and the scan components of each vector.

The last five lines include the derivative error calculation and the actual messages sent to Ackermann. The derivative error is very simple; it is just the last y-component stored minus the current one, to reduce the oscillations the car used to make. This error is then multiplied by a low, negative K_d value to replace the `total_y` value. This is the y value that is then used to calculate the steering angle and speed. The angle is determined by taking the arctan of the `total_x` and `total_y` components and then multiplying it by a steering angle variable declared in the main class and the sign of the `total_x` value. This is to enable the car to almost completely, on a dime, change its steering angle from a positive to a negative number; this can enable it to make a turn out of a dead end or make a drastic turn to avoid steering into a wall. The speed uses the Pythagorean theorem to take the value lying between the x and y components, which is then also multiplied by a speed variable declared in the main class and the sign of the x-component of the vector. This allows it to have 'negative' speeds, or simply to speed up in the opposite direction. This also helps it back out of walls.

Our car ended by reaching success in two of the three challenges. In the first technical challenge, the car did not crash into a single wall while exploring the space. However, most of the blobs it detected and took screenshots of were of the reflection of the sunlight off the floor, leading the code to draw a box around it and declare it as yellow. In the second technical challenge, the car did not succeed in any of the runs, mostly which was thought to be the sunlight making the cardboard around it also look red. The light streaming through the windows at this time was much more than the light coming in when the RGB values were calibrated, at 2:00 P.M.; luckily, the car would be racing around this time, so hopefully it would not suffer the same fate. In the actual race, it turns out that the car did not successfully detect green in any of the runs, but it detected red in two of them. It seems like it went into the area with the most free space, which was the longer lane. The car finished fifth overall, last in its heat, and completed the race once in the final race. Overall, this suggests mostly successful results, and had there been more time to calibrate values and test the derivative function (which was added two hours before the competition), the race might have gone differently.

References

1. Goodwin, G. C., Graebe, S. F., Salgado, M. E. (6 Oct. 2000). *Control System Design*. Published by Prentice Hall PTR.
2. Artstein, Z. (1980). "Discrete and Continuous Bang-Bang and Facial Spaces, or: Look for the Extreme Points". *SIAM Review* 22-(2). p. 172–185.