Autonomous Navigation of a Structured Environment

Samuel Maynard (captain)	swm668	samwmaynard@gmail.com
Christopher Denny	cd28957	chris.denny@utexas.edu
Jack Ceverha	jwc3255	jack.ceverha@gmail.com
Eric Lee	ejl966	ericlee123@gmail.com

Sam Maynard, Christopher Denny, Jack Ceverha, Eric Lee
May 9th, 2016
swm668, cd28957, jwc3255, ejl966

Project Description -

The goal of this project is to have our car follow a wall and make turns around a place or angle where two or more sides or edges meet using PID feedback mechanisms. This project is a combination of the work we completed in the past, in terms of controlling the motion of the car and balancing an inverted pendulum. Our controller is built on the assumption of a wall of homogenous appearance (ie. smooth and uniform). This assumption proved problematic when presented with the problem of making an entire loop.

In terms of reaching our goals for this project, we began with having our car follow a wall. From there, we adapted our strategy to have the car also turn a corner, which in a way, is still the task of following a wall. With this implementation, we did not need to do anything extra for the car to follow a loop around a track. This was not exactly apparent in our demo, but the issue was not due to work on our end. The makeshift wall made out of trash cans caused our corner finding algorithm to fail; had the wall been solid, as is expected in a legitimate track, our car would have made it around the track perfectly. Going above and beyond, we also recorded some very enticing rosbags, full of really interesting data that shows how our car responds to certain situations with respect to wall. One rosbag in particular (around.bag) showed that our assumption of uniform course structure is inadequate, as the LIDAR did not interpret intuitively smooth obstructions as smooth.

Task Description -

A general purpose wall following algorithm would ideally be able to deal with a wide variety of courses and obstacles, including dynamic ones. However, given the limited scope of our project, we simplified the problem being solved. We assume that the car is always following a wall on its right hand side, we assume that the wall has consistent reflectivity (for our LIDAR to work), we assume that the wall is linear, and we assume that corners have at least a four foot wide mouth. This allows us to solve our problem in a simpler, more direct (but more error prone in complex situations) way instead of having to program a rudimentary artificial intelligence. To start we will describe our

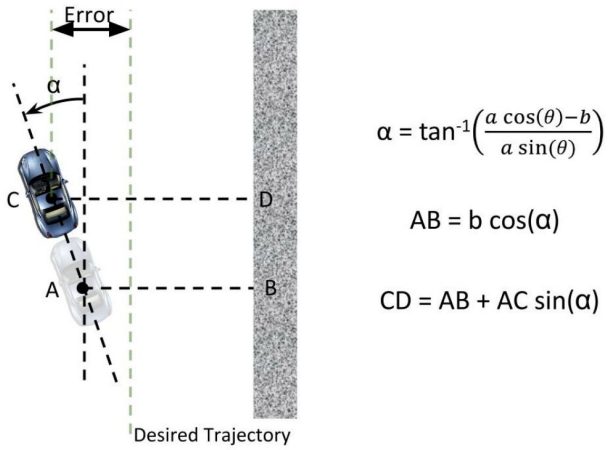

Figure 1

solution for following the wall, and then expand that to include our corner turning strategy.

Our wall following strategy involves using a PID algorithm operating on the car's distance from the wall. We calculate the car's distance from the wall using two fixed angle readings from the LIDAR sensor, one on the car's x-axis towards the right (towards the wall), which we call 0°, and one at 50°. We can then calculate the distance to the wall through trigonometric identities, as detailed in figure 1. From here, we can PID towards a target distance from the wall, or equivalently we can subtract our target distance from our current distance and PID towards 0. This is what we chose to do. The output of the PID algorithm is fed back to the car as the angle to turn the wheels to. The velocity of the wheels is kept constant. The PID constants we used were P = 28, I = 0, and D = 0.18.

This rudimentary control algorithm works for about one "cycle" of the PID loop - that is the car would turn away or towards the wall (if it was too near or far from the wall to begin with), but when it passed the target distance it would not re-correct fast enough, and then rapidly over correct when it was too late. This is because the PID algorithm calculates the objective angle that the car should continue at - that is if the PID algorithm outputs an angle of 0° then it wants the car to run parallel to the wall. However, the car interprets angles relative to its current frame of reference, that is 0° means "drive straight." This means that when the PID returns 0°, the car will instead drive at whatever angle it was at before. This leads to the car correcting too late, and then drastically over correcting as we witnessed during testing.

This could be fixed in one of two ways. The first is to transform the angle output from the PID algorithm from the global frame of reference into the car's frame of reference. This would be done by subtracting off the angle we calculated the car to be at from the wall (α in figure 1) from the PID output. In order to achieve this we would need to rework the structure of the software, as α was only used to calculate the absolute distance from the wall. This would mean modifying the message passing structure, which comes at a high cost. The other option, which we decided to

implement, was to project the car forward into the future and make a decision based on that position. This doesn't correct for the difference in the reference frames directly as the previous option did, but is significantly easier to implement. By projecting the car a constant time forward in the future (we used 1 second) the car realizes if its current trajectory will not be beneficial to it in the future, and turns the other direction if it will not be. This way, if the car reaches the target position with a non-zero angle, instead of continuing forward like in the previous algorithm (since the PID would output 0°) the car instead realizes that at a second in the future it will have a non-zero error, and will choose instead to output an angle that will correct the car onto a horizontal path with the wall. The projection process is detailed in figure 1.

Following corners involved extending our previous algorithm by adding a state machine to the car's logic. Instead of having the car try to PID around a corner, we employed a heuristic to try and detect corners, and then switch from the "wall following" state to the "corner" state. When a second heuristic was satisfied the state machine would transition back to the "wall following" state and the car would resume using the algorithm detailed above. The corner finding heuristic involved monitoring the distance at a 45° angle from the car for sudden jumps. Through an iterative testing procedure, we decided that if the distance jumped more than 1.5 meters to a distance that was greater than or equal to 3 meters that we had encountered a corner. We would continue to stay in the corner state until the second heuristic was satisfied, this one being much simpler: the distance at the 45° mark falling below one meter. The behavior of the "wall following" state is the behavior discussed above, and the behavior of the "corner" state is to output the max angle towards the right.

This state machine worked fabulously as long as our previous assumptions were held to. If the course was simple and flat, the car would never misidentify a corner, and would very quickly take the turn. However, in more complex situations like finicky obstacles, the car would sometimes misidentify a corner and go careening into an obstacle.

Future improvements to our strategy would be, in the short term, to improve the corner detection from the simple heuristic that it is. This could involve using a more complicated heuristic, for example checking at multiple different angles instead of just 45°, or by modifying the "wall following" PID behavior to work for corners as well. In the long term we would want to improve our strategy by developing a mapping algorithm like SLAM and then using path planning. This would allow us to navigate far more complex environments than what we detailed in our assumptions.

Software Architecture -

Our logic is divided into seven units, each of which is comprised of a ros program. Two of these are hardware interfaces and not created by us: the urg_node in package urg_node and the serial_ node node in package rosserial_python. The first receives the LIDAR data, and the second sends data to the drivetrain of the car. We will cover the five nodes that we implemented and that handle the higher level logic of the car: control, corner_finder, dist_finder, kill, and talker.

The progression of logic starts with dist_finder. dist_finder subscribes to the LIDAR data

being streamed from the urg_node and does the calculations as noted below. It takes the result of its projected position, subtracts a pre-defined desired trajectory distance, and publishes this value to the `error` topic. The other data processing node, corner_finder, also processes incoming LIDAR data. The corner_finder looks at every LIDAR scan and checks to see if the corner or wall heuristics are satisfied, and transitions to the appropriate state if so. This state would be published continuously to the `mode` topic. From here, the control node subscribes to both the `mode` and `error` topics. Depending on the mode published by corner_finder, the control node takes action according to error values (distance from wall). If in 'corner' mode, control publishes the max angle towards the right to the `drive_paramaters` topic. If in 'wall following' mode, control will do the calculations for the PID algorithm, cap the angle at 100 or -100, and publish the result to the `drive_parameters` topic. The `drive_parameters` topic is received by the talker node, which has the simple job of translating the arbitrarily scaled number published by control into an appropriate power to publish to the arduino on the `drive_pwm` topic. This number is consumed by the rosserial node, which talks directly to the hardware. All of this work is moderated by the kill node, which listens for keyboard input and publishes an emergency kill to `estop` if the delete key is pressed.

A detailed description of each logical node follows.

```
kill:
  usage: rosrun race kill.py
  description: Used as an emergency stop switch. Listens for the
spacebar key and the backslash key. Spacebar is emergency stop
and will kill the car, backslash is used to resume normal
operation. It is recommended to run the kill node before any
other nodes and put it in emergency stop mode, and then
triggering normal operation when the car is fully setup.

corner_finder:
  usage: rosrun race corner_finder.py
  description: Publishes a continuous stream of strings that
indicate the mode that the car is in. Functions as the
gatekeeper for the finite state machine logic. Subscribes to the
scan topic and on every scan received checks to see if a
heuristic is satisfied and a state transition can take place.

dist_finder:
  usage: rosrun race dist_finder.py
  description: Transforms incoming laser data into a useful
error metric. Takes scan data and performs finds the car's
```

```
distance from the wall at a projected one second into the
future. Subtracts the desired distance from the wall and
publishes the result to the error topic.

control:
  usage: rosrun race control.py
  description: Listens to both the mode and error topics. Makes
the final decision on the velocity and direction of the car. If
in the 'wall following' mode, computes the PID with the error
and outputs the result to the drive_parameters topic. If in the
'corner' mode outputs the maximum angle towards the right to the
drive_parameters topic.

talker:
  usage: rosrun race talker.py
  description: Processes the output from control. Control
assumes arbitrarily that angles go from -100 to 100. Talker
transforms this into the pwm scale that is processed by the
drive_pwm topic that it publishes to.
```

Performance Results -

Overall our software worked pretty well in solving the systems tasks set to us. It navigates right-hand turns confidently and efficiently, quickly returning to the equilibrium wall-following distance. Indeed our car would be able to successfully go around a track (barring the trashcan example described in additional notes). When following a line it works very well, gradually bringing itself to the equilibrium point. We made sure when setting our PID constants to not make them too aggressive. It wouldn't make sense for the PID to over correct even when close to equilibrium. As such, it gradually makes its way to the racing line, much like in actual racing. Our turning algorithm also works pretty well for most corners we tested on.

Here it doesn't actually follow a racing line like we'd prefer but it still comes pretty close. Ideally we'd like the car to know way ahead of time when the corner was coming so it would be able to slow down just enough and clip the apex, yielding the fastest time. For now we are content with it consistently finding the corner and managing to turn before continuing on the next straightaway.

When running timing statistics on the different python modules we find that for each LIDAR tick, the numerical analysis takes 7.5E-5 seconds. At a LIDAR tick rate of 60 times a second this leaves an enormous amount of time for further calculations that could be done when expanding upon our algorithm. Even assuming pessimistic estimates for ROS overhead, this will leave over 99% of the $1/60^{th}$ of a second we have free for further computation. The network lag is a much larger problem, as it could take longer than $1/60^{th}$ of a second to run from the on board tegra to the

wheels, but since we process every LIDAR tick in an appropriate amount of time the lag from the board to the wheels is fixed. This essentially means that we are acting slightly in the past, but our well tuned PID algorithm should account for this small inconsistency with no problems.

Additional Notes -

There were several bugs we ran into while developing for this project. Some of these were problems we caused and later were able to rectify, whilst others were problems inherent to the system and thus we were unable to fix.

There were a few issues we found with the car that made algorithm development difficult. One of these was a shortcoming with the LIDAR. We found that when using the car near metal surfaces, the LIDAR was incapable of accurately detecting distances. Any time there was a section of the wall it was following that contained a strip of metal, the values returned back in the message were wildly inaccurate. As you can imagine, this became rather difficult to deal with it as it caused our PID algorithm to veer way off of equilibrium. We still don't have any surefire way to improve our algorithm's resilience to this type of error. The other problem we came across was the life of the battery. Time and time again, we'd be ready to test a new set of constants or different PID mechanism only to find the batteries had gone flat. This really cut into our efficiency as we'd then have to wait a while for the batteries to recharge before we could find the efficacy of our latest changes. This often set us back in terms of how much we could accomplish in a single day.

Along with the above, there were some bugs we created, some of which we fixed, while others remained persistent no matter what we did. Before we could even begin developing PID algorithms, we ran into several problems with simply setting up our workspace. Indeed, from the beginning, we felt that ROS was very finicky in what we were or weren't allowed to do. Setting up our initial catkin workspace required a very specific set of steps and failed if we even deviated in the slightest way. Needless to say, by the end of the project, we became very skilled at setting up the initial boilerplate code. Following along the line of ROS being finicky, another problem we ran into at the beginning was subscriber and publisher setup. Similar to workspace setup, ROS was very strict about which modules and messages had to be started in which order. Eventually we managed to get this gist of how this worked. One other issue took considerably longer.

When first trying to get the car to accept keyboard control, we spent a long time unable to even send messages to the car. After a substantial amount of debugging we found that it was because the car didn't ever have the race module run on its own hardware so it didn't know how to interpret it when we sent that module from the laptop. We finally managed to circumvent this issue by precompiling the race module on the car, just so it would know what it was looking at. After this we were able to send messages through this module to the car and it was able to understand how to use these messages to move the car. Another issue we came across when trying to get the car to move was the order in which to start the hardware onboard the car. For the longest time we always turned on the motor after turning on the tegra. It was only later that we found out that the motors have to be turned on before everything else for the car to function.

Finally, once all of the above was working, there were a few hiccups we ran into with our sensing and PID algorithms. When first implementing methods for getting distances at different angle offsets and acting off of those readings we kept finding strange behavior. We soon discovered we had made the mistake of using global measurements rather than relative measurements. To put it another way, we were mixing up our calculations between values that were relative to the wall and other values that were relative to the centerline of the car. Once we changed it all to consistently use relative measurements, we got much more controllable behavior. One problem that we still haven't been able to fix deals with the corner detection algorithm. We found that when the car came close to two trashcans at least six inches apart it interpreted that as a corner and attempted to drive between them. We still haven't come up with a method to prevent this from happening while still achieving reliable corner detection in other cases.