RRBJ (Rahul, Russell, Bryan, James)

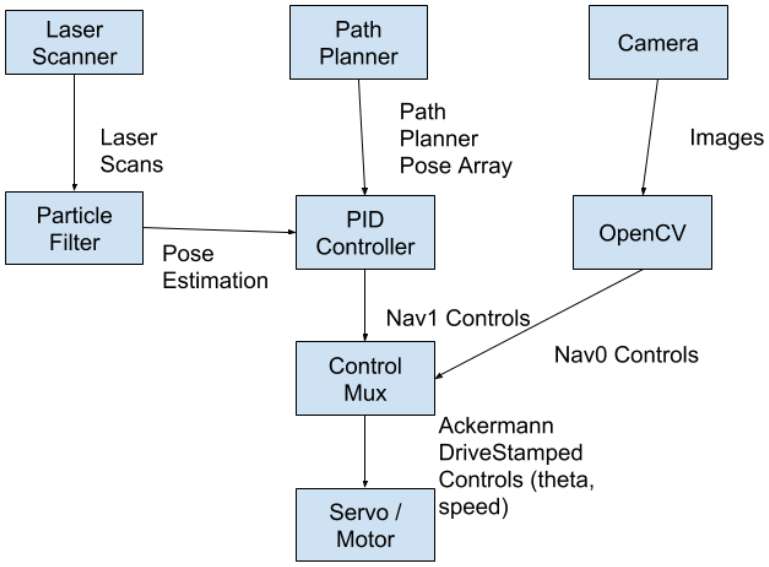
Code Repository: https://github.com/russelldeguzman/ee545_robot_car

EE 545 Final Report

Abstract

For our EE545 final project, our group attempted to program a robot to autonomously navigate the EEB basement. The robot had to drive over various “targets (blue pieces of paper)” scattered throughout the basement as well as avoid “traps (red pieces of paper)”. We implemented three of the algorithms taught in this class to control the robot, namely - MPPI, MPC and PID. We then came up with an architecture using path planning, OpenCV, and PID control to navigate to waypoints and avoid traps, and a particle filter to estimate the robot’s pose within the map.

Architecture



Robot’s System Architecture

Our high-level system diagram (above) shows the way we organized our control flow. Our main components consist of a position-estimator, a path planner, a PID controller, and a CV controller. We utilized the laser scanner to estimate the pose of the car in the map, as we did in the second lab so the PID controller has an idea of its location. The path planner we leveraged from Patrick’s provided functions in Lab 0 and it provides an array of poses which represents the path the PID should take. The PID then publishes its controls based on its inferred distance from the plan. In parallel, the CV node takes image input from the camera and observes any blue/red targets on the ground. If it sees a blue target, it publishes higher priority controls to steer towards it and conversely for the red target.

Path-Planning

To save time, our group leveraged the path planner provided by Patrick in the earlier labs. We also wanted to design a path that strategically hit our targets in an order that could minimize our time as well as avoid traps. We created a script that created a bagfile of pose arrays that the robot used as ideal goal positions. The robot, while in action, would pop positions off of a python “deque” object get a new position as it moved forward. We found that the path planner would not always compute the most optimal path given just the provided waypoints, thus we added additional waypoints throughout the path to provide more control authority to the generated path. After some tweaking we were able to generate a path that minimized turning.

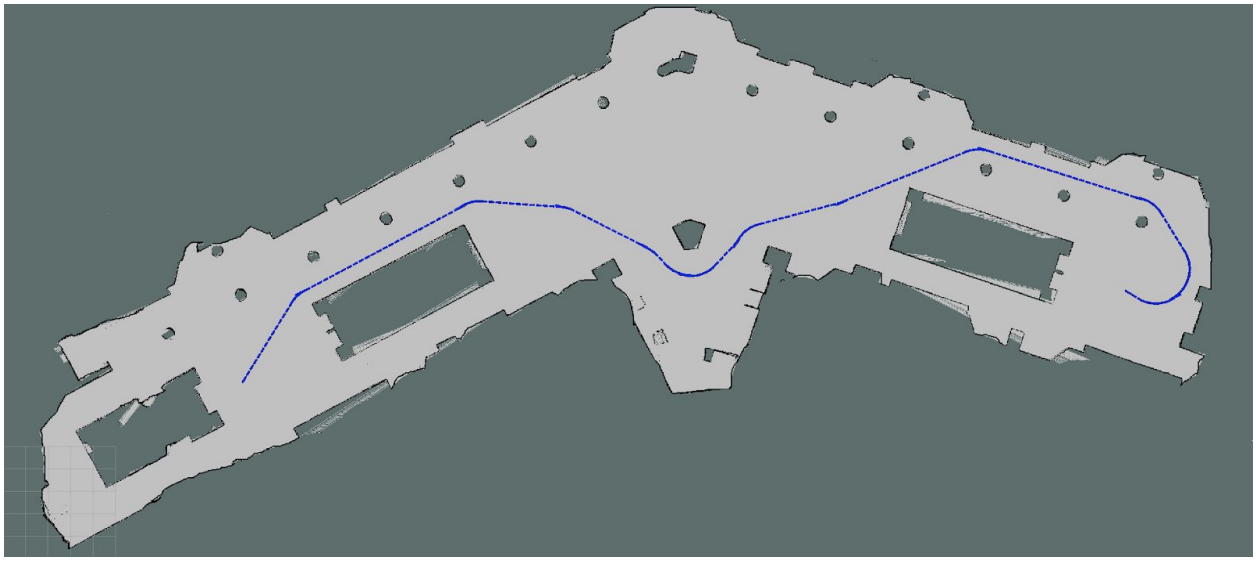


Image of the final path

```
# Start Pose
start_pose = np.array([[2500, 640, 5.7]])

# Start and Blue targets from the CSV files
plan_targets = np.array([[2600, 660, .46],
                         [2538, 438, 2.83],
                         [2198, 331, 2.83],
                         [1880, 440, 3.55],
                         [1642, 523, 4.12],
                         [1533, 606, 3.14],
                         [1435, 545, 2.7],
                         [1256, 457, 2.9],
                         [1030, 450, 3.6],
                         [675, 639, 3.6],
                         [540, 835, 4.2 ]])
```

The tuned path was based off of these target points and pose orientations

MPC

The idea behind MPC as the overall control mechanism was that we could design a cost function and use the predicted rollouts to achieve close proximity to our targets. The rollouts are

based on the theoretical kinematic constraints of the car, so each predicted path, should correspond with car positions for a given steering input. Unfortunately, we opted not to use this control because of time/complexity tradeoff. Our first approach was a cost function that was the sum of the Euclidean distance from each of the goal poses for the rolled out steering angles. However, this did not yield any good results even in the simulation.

A second MPC-based approach was to precompute the costs for the entire map using Gaussian distributions to create a gradient of lower costs towards the precomputed path. This worked really well in a simulation but still not as well as our basic PID approach. Additionally, when testing it out on the robot, it had issues following the cost gradient, thus we had to abandon this approach due to time constraints. The cost gradient is composed of two gaussians centered around the path positions. The first gaussian with smaller sigma value had the effect of keeping the MPC on the path. The second gaussian with a larger sigma value had the effect of redirecting the robot back to the path if it wandered off too far. Lastly, the walls were all inflated to penalize rollouts that were too close to colliding with the walls.

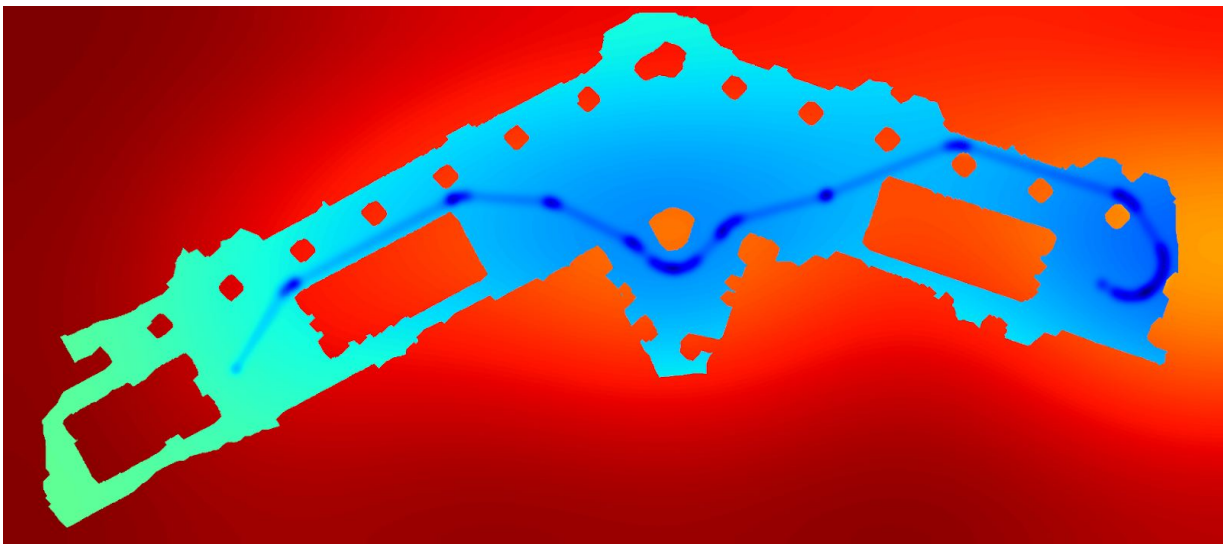


Image of the cost gradient

MPPI

We also considered and implemented an MPPI controller, using our kinematic motion model as the forward motion propagation algorithm (we did not implement a NN dynamics model). This performed reasonably well under certain circumstances, but was not as robust as PID control for the known-course task involved in the demo.

One of the main issues with implementing MPPI was striking a reasonable balance between robot movement speed and stability of the nominal control. Our implementation as of the time of the demo used fixed variance values for the distribution from which both speed and steering angle were sampled. In retrospect, because the speed noise didn't track the current speed of the car, this made the speed controller susceptible to wind-up issues, i.e. the noise signal was unable to change the nominal control rapidly enough to avoid overshooting a target. This primarily showed up while attempting to converge on the goal. When navigating short paths, the car would be able to decelerate and arrive at the goal, but when navigating over large

distances (where the nominal control had more time to accumulate velocity), the car would overshoot the goal and attempt to deal with this issue by circling around. Various tweaks to the cost function were tried to reduce this behavior, but it proved fairly resilient under a number of cost function implementations. Using a larger fixed variance value was unable to resolve this issue as it tended to introduce too much instability to the nominal control.

Lastly, a major issue in MPPI control was the inherent race condition between the physical speed of the car and the computational speed of the algorithm. As the maximum car speed was increased, the algorithm needed to iterate more rapidly to adapt to obstacles before the car would be at risk of colliding with them. Reducing the K (# of rollouts) or T (length of rollouts in time steps seemed to produce less stable paths, but it was difficult to strike a balance between stability and useable compute time. One other issue that we didn't have time to resolve was that our particle filter seemed to update faster than 10 Hz, so the time step, dt, used in computing the rollouts could vary between each iteration. Downsampling the update from the PF so guarantee fixed-time updates would likely have helped with stability of the nominal trajectory.

PID

For our path-following algorithm, we opted for the PID controller, as we got the best robot performance out of it. The PID is responsible to get the robot within close range of the target waypoints. It was also an easy candidate to tune and debug. The PID controller receives the robot coordinate position information from the particle filter's inferred position output. Furthermore, the controller receives its plan/goal information from the path planner node in the form of pose arrays. From these two inputs, the error offset can be calculated similarly to lab one by taking an error measure of translational and rotational offset between two poses. We published these controls to our "Nav 1" topic (a lower priority) so that the CV module could take over controls ("Nav 0") once a target is within close enough range to be detected by the camera.

OpenCV

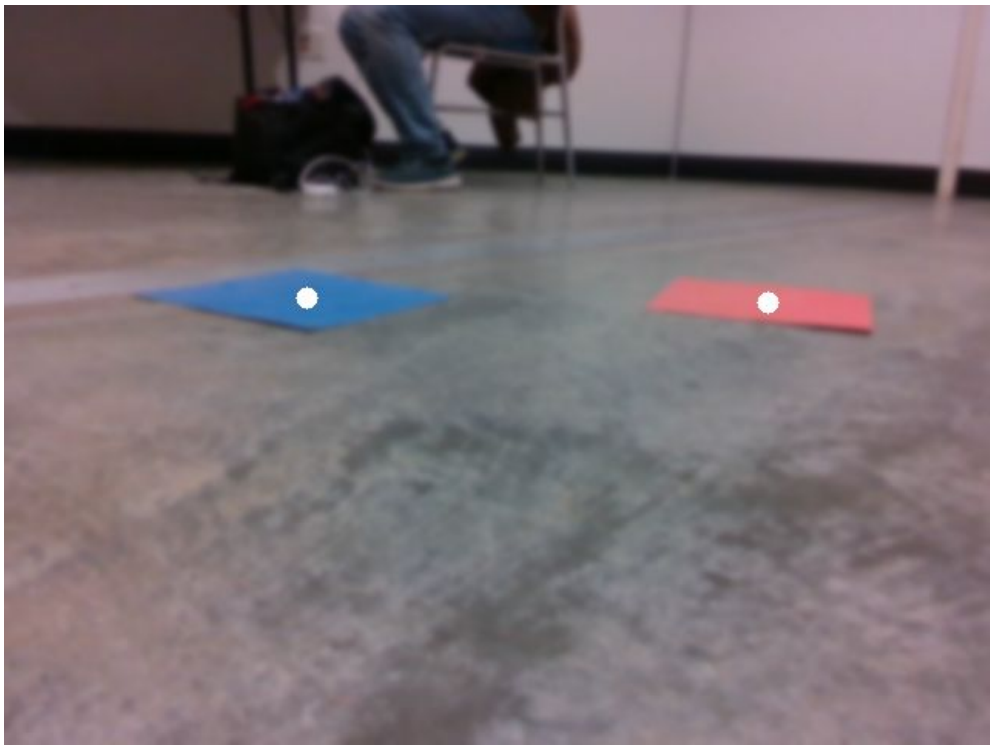
The computer vision module used colour based thresholding to detect the red and blue squares. Using the CV bridge, we converted the ROS image to a CV image and passed it to a function where the thresholding masks were applied. Initially, we used a simple RGB threshold range to segment the blue and red regions in the image separately. Since the RGB colour space is not invariant to lighting changes, we converted the image to the HSV colour space and had threshold ranges for blue and red to get the blue mask and red mask respectively. While these ranges worked well for the lighting in the lab, the illumination differences in the basement threw our CV module off. We recalibrated the vision system by collecting the values of the Hue, Saturation and Value at various points on red square, blue squares in the image and averaged them. We then set a tolerance limit around this mean value which gave us the upper and lower bounds of the threshold. Once we received the bit masks after using the calibrated bounds for the red and blue colours, we found the largest contour of the resulting binary image, which in most cases would be the contour corresponding to the red/blue squares. The centroid of the largest contour served as a good estimate of where the squares lay. The image space was divided into a fixed number of bins which corresponded to a steering angle and found which bin

the centroid was in. This method discretized the steering angles in steps of the number of bins and did not prove to be as effective - causing the car to oscillate when the centroid position moved from one bin to another. We then switched over to a PID control scheme which in theory would give us a smoother control.

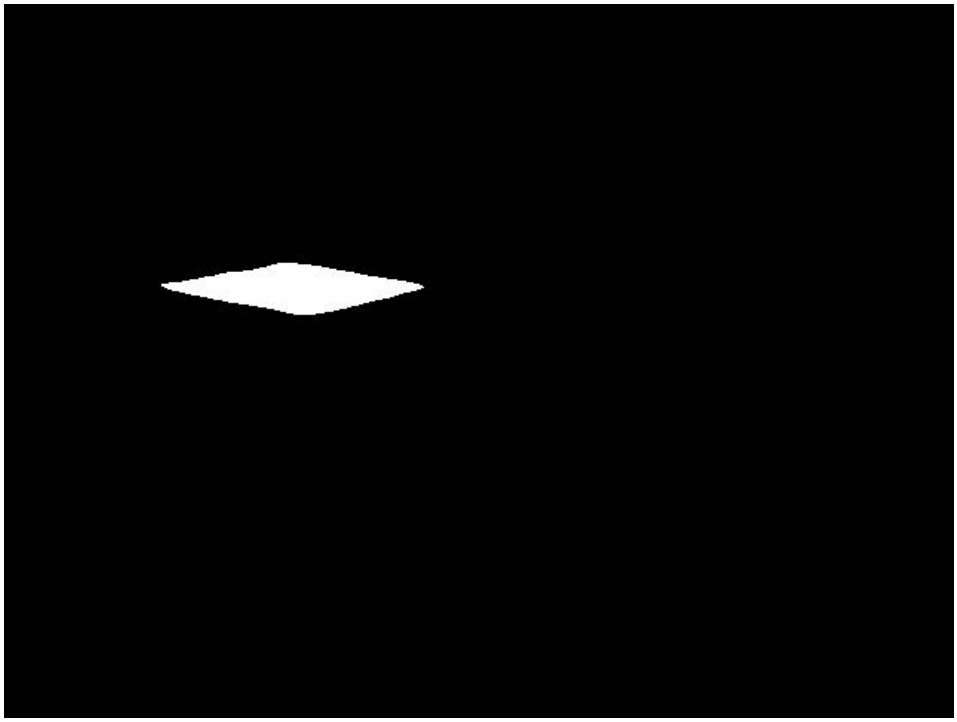
The difference between the x-coordinate of the centroid of the largest contour and the half the width of the image gives us a pixel offset of the head of the car from the centroid of squares. We then converted this pixel-offset error into angle error by translating the offset into a percentage of the width of the image and multiplying that with the max steering angle.

```
def compute_error(self, centroid_x_pos, img_width):
        x = (((img_width / 2) - centroid_x_pos) / float(img_width)) *
self.max_angle
```

We then fed this error into a PID controller which determined the angle to publish to our drive controls. We chose to publish to the high-level ackerman_cmd_mux - nav_0 topic which had a higher priority than the base PID controller. This allowed the robot to smoothly steer towards a target as it came into the range of the camera.



Target Centroid Detection



Blue HSV Mask

Red HSV Mask

Results

Overall, the PID algorithm with the CV control scheme worked out the best. We finished in 29.52 seconds and hit all of the targets. We encountered some difficulties during transferring our system to the robot. One of the issues was CV image processing time while the PID was running at higher speeds. If the speed produced by the PID was too high, the robot would scoot past the target and react to the CV controls a second later. We overcame this by shrinking the area size of the images by ½ using cv2.resize, and this drastically reduced the compute time on the car.

During our trial, we were able to incrementally speed up the car while also tuning the PID controllers to maintain stability. Our method to tune the PID was to first set all the terms to zero, then incrementally increase the proportional term until the car started oscillating. Then, we would back off the P slightly and increase the D term and perform the same iterative tuning. We opted not to use the integral term due to computing time.

We also spent our final hours tuning the Path plan so we could choose an optimal PID path. To make the path planner hit a certain plan, “dummy” objectives were also added to the planner to force the plan to, for instance, steer around a specific pillar on the map. All of these nodes with the exception of rviz were run on the car to avoid latency and connection issues.

Conclusion

We were all happy with the outcome of this final project due to all the algorithms we were able to learn and implement, even though some did not perform as well as we anticipated. Furthermore, we all learned to appreciate the simplicity and performance of the PID controller. We ended up using two in our design and they both were orders of magnitude easier to implement and debug than other schemas. To take this challenge further, an interesting future assignment would be to introduce moving obstacles to the course, so the car would have to implement some sort of object avoidance (and make use of MPC). It also would have been interesting to get MPPI working on the robot, as it seemed to have promise in the simulation. That being said, we are quite happy with these results. Thanks for the fun class!