EECS 106B Project 2 - Nonholonomic Control with Turtlebots

Prabhman Dhaliwal Matthew Hallac Jaiveer Singh

***Abstract*—In this paper, we investigate the performance of three path-planning strategies for nonholonomic control of a Turtlebot with simulated bicycle dynamics. These planners are subsequently tested against a series of elementary movements and obstacle-avoidance tasks to judge the effectiveness and generalizability of each technique.**

***Index Terms*—Turtlebot, Nonholonomic control, RRT, Nonlinear optimization, Sinusoidal steering, Bicycle model**

I. Methods

For the purposes of this investigation, we considered three separate path planners. First, we considered a Sinusoidal Steering planner; second, a Nonlinear Optimization planner; and finally, a Rapidly-exploring Random Tree (RRT) planner.

A. Sinusoidal Steering Planner

1) Theory: The sinusoidal planner utilizes out-of-phase sinusoids to control the nonholonomic system. To use this technique, we need the system dynamics to approximate a canonical model:

$$\dot{x}_1 = u_1$$
$$\dot{x}_2 = u_2$$
$$\dot{x}_3 = x_2 u_1$$
$$\dot{x}_4 = x_3 u_1$$

For our car model to match this form, we apply a nonlinear transformation

$$\dot{x} = v_1$$
$$\dot{\phi} = v_2$$
$$\dot{\alpha} = \frac{1}{l}\tan(\phi) v_1$$
$$\dot{y} = \frac{\alpha}{\sqrt{1-\alpha^2}} v_1$$

$$v_1 = \cos(\theta) u_1$$
$$v_2 = u_2$$
$$\alpha = \sin(\theta)$$

However, in the model above, when $\theta \in \{90°, -90°\}$, $\cos\theta$ goes to infinity, and we hit a singularity. To account for this, need to switch to an alternate model as described below. This alternate model is formulated as:

$$\dot{y} = v_1$$
$$\dot{\phi} = v_2$$
$$\dot{\alpha} = -\frac{1}{l}\tan(\phi) v_1$$
$$\dot{x} = \frac{\alpha}{\sqrt{1-\alpha^2}} v_1$$

$$v_1 = \sin(\theta) u_1$$
$$v_2 = u_2$$
$$\alpha = \cos(\theta)$$

For open loop control with the transformed canonical model, we steer x, ϕ, α, and y separately before combining them. This algorithm is described in detail in *Nonholonomic Motion Planning: Steering Using Sinusoids* by Murray and Sastry [2].

2) Input Constraints, State Bounds, and Singularity: The state bounds on the system were regulated similarly across all the values. There was a maximum value for the steering angles that are passed into our sinusoidal planner that can be set within the configuration space, and a simple check will allows us to confirm whether or not our trajectory violates this bound. This is similar for the x,y state bounds, and the bounds on state theta is directly tied to our actions for resolving singularities, since theta can only cause state problems at angles that violate the model. To quote the spec regarding the standard canonical car model, "When the robot is turned 90 degrees, we know that all the input velocity u_1 goes in the y direction, and everything works normally. However, when that occurs in the model our model breaks".

The bounds on the inputs constraints for u_1 and u_2 are also passed into our functions in the form of maximal u values for each constraint. These constraints are important for our overall sinusoidal model because they are tunable parameters that define the total amplitudes a_1 and a_2 of our sinusoids. For example, in order for the point turning and simple motion paths trajectories to be complete, we set our u_1 to be 1 initially, and for the parallel parking it worked much better with a u_1 of 3.5. This gives our amplitude more slack to work with, and allows us a wider range of values that we can binary search for the values a1. Regarding these constraints, since the initial algorithm was designed with unconstrained systems in mind, we have to implement these constraints manually on our system, which forced our robot to take more motions on each state individually to compensate for the constrained system.

While a_1 was initially set to some value that was in the range of 0 to max(u_1), this value had to be searched properly for the 4th state value (y in the case of the standard canonical car model and x in the case of the alternate model). This was because, the β_1 value of our fourier series also relied upon this a1 value as well as a2. This search was done with a standard recursive approach, with an error threshold set once we got close enough to the true value of that state at the end time. Once a1 and a2 were

figured out through our recursive approach, we were able to get our final value for the 4th state value, and compare with the goal value, and get its error. Once we were within some threshold, we accept that value and use it for our planner.

The pseudocode for the singularity avoidance algorithm was based off a handoff method that iteratively broke up the regions that would give us singularities and use the proper model (standard or alternative) on those regions. The code utilizes a lot of for loops for sifting through the regions and properly breaking them up, but the core concept was describing handoff points (-45, 45, 135, -135 degrees), where we would transition from one model to the next. If the start state and goal states are in different quadrants based on those 4 handoff points, we have to turn on Singularity Avoidance, meaning we would have to figure out which regions we change models. These work in pairs, meaning that the first region would use one model, the next would use the other, and so on. This pattern continues until the final handoff theta near some threshold on the goal theta. This method was the one we settled on, where our intial tries were based of having a "soft region" around known singularities (like 90 degrees in the standard model would have a soft region of 85-95 degrees), that we would transition from one model to the next. This was not as dynamic as the handoff approach, and would often fail to change models quick enough.

B. Nonlinear Optimization Planner

1) Theory: The optimization-based planner takes the problem, considers constraints such as bounds and obstacles, and solves for an optimal path based on some metrics in the closed-form equation. This problem is a nonlinear optimization model because the bicycle model with steering is nonlinear.

2) Parameter Optimization: In this model, N represents the number of waypoints and dt is the time step. When choosing these parameters, it is important to consider safety between waypoints. If obstacles are small enough and the gap between points is large enough, there is potential for collision between waypoints. dt should be small enough and N should be large enough such that the distance between each jump should be smaller than the smallest obstacle. For an arbitrary goal state, given the size of the obstacles, one could set the number of waypoints to be the minimum to not have obstacle sized gaps plus a buffer to ensure there are no collisions, or set the obstacle sizes to be larger.

C. Rapidly-exploring Random Tree (RRT) Planner

1) Theory: RRTs are a commonly-used motion planning technique that broadly involves sampling possible configurations in the state space, building short paths between these configurations, and ultimately connecting many of these small paths as edges of a tree to span from the start to goal states.

More precisely, the basic RRT algorithm is as follows:

1) Begin with an empty graph G, such that nodes of G represent possible configuration spaces and such that the edges of G represent short motion plans that can be followed by the robot's controller.
2) Insert the starting configuration c_0 into the graph as the start node of the tree.
3) Using an appropriate heuristic, sample a randomized, valid configuration out of the configuration space as a new configuration c_{rand} to add into G.
4) Using an appropriate distance function, identify c_{near}, the configuration in G that is closest to c_{rand}.
5) Construct a small local plan p between the configurations c_{near} and c_{rand}, discarding this iteration if a valid, collision-free path cannot be found.
6) Generate a small prefix of plan p called p_{prefix}.
7) Identify the configuration achieved at the end of following p_{prefix} as c_{new}, and insert c_{new} into G, connected to c_{near} via p_{prefix}.
8) If c_{new} is sufficiently close to the goal configuration c_{goal}, then generate a final plan p_{final} from c_{new} to c_{goal}.

D. Local Planner & Heuristics Design

While the basic RRT algorithm is powerful, several choices and changes can be made to improve the performance of the planner in terms of speed and convergence. First is the question of distance function. Since the robot's configuration space is appropriately represented by four variables (x, y, θ, ϕ), and since θ has a periodic property, it is necessary to use a special distance function instead of the simple Euclidean norm in 4 dimensions. We found that the best implementation involved converting the heading angle θ into a complex number representation $a+bi$, which then naturally handles the fact that 0 and 2π are the same angle. Then, we can simply take the Euclidean norm of the coordinates (x, y, a, b) to get a uniform distance metric.

Another main consideration is the procedure used to sample new configurations. We determined that a goal-biased sampling strategy was most effective. With probability $\epsilon = 0.05$, the new random configuration c_{rand} was set equal to the goal configuration itself. This ensures that the planner adequately explores in the neighborhood of the goal, and increases the likelihood of finding a solution within the allotted iteration count.

Finally, and perhaps most importantly, is the question of appropriately constructing a local plan at each step. Given the complex nature of the robot car system, making a local plan efficiently is computationally difficult. However, since we know that RRT considers only a short prefix of the local plan, we can save computation by only preparing incomplete local plans in the first place. These so-called 'motion primitives' need to be varied enough to accomodate all possible motions, but also few enough to be computed rapidly and frequently during the iterative solving process. Taking inspiration from the Dubins curves solutions for a classic Dubins car, we provided three main classes of motion primitives: straight, in

which $\phi = 0$; right, in which ϕ is its maximum value; and left, in which ϕ is its minimum value. In each case, we also vary the velocity, producing 20 discrete velocity values for each of the three directions. The subsequent 60 motion primitives are curves and straight lines of various lengths. To complete the motion plan, we simply return the motion primitive that brings us closest to the target configuration.

II. Experimental Results

A. Summarized Results

Overall, for these environments, the optimization planner provided direct paths to the goal and performed well at avoiding obstacles. In the simulator, the optimization planned paths ran smoothly, however solving for the paths took a relatively long time. The sinusoidal planner solves for each constraint separately. This is clear in the figures for the sinusoidal planner, where each path appears to be a piecewise function of sinusoids, where after each motion one of the state variables is set.

B. Manipulation Tasks

1) Sinusoidal Planner:

a) Simple Motion: Figure 1 This diagram uses the simple motion with a tuned u of 1. This planner worked well along the bounds of our input constraints and state constraints, however would sometimes go our of x,y bounds. A tuned u of 1 was the best it could work with.

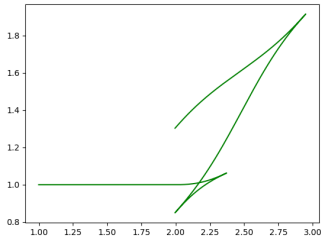

Fig. 1. Sinusoidal Planner Simple Motion

b) Parallel Park: Figure 2 Parallel park rarely respected the x,y bounds on our system, and required extensive tuning. We settled on a tuned u value of 3.5, that allowed it to parallel park well, but broke the x,y constraints sometimes.

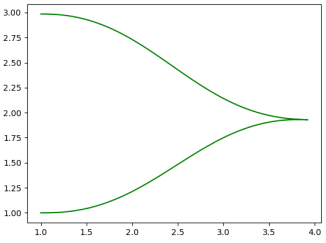

Fig. 2. Sinusoidal Planner Parallel Park

c) Point Turn: Figure 3 This diagram also used a tuned u of 1, and generally had the most error in the x,y direction (the 4th state in the model it was using). This one suffered the most from singularities, but generally didnt break the state bounds like parallel parking.

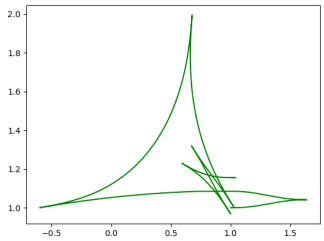

Fig. 3. Sinusoidal Planner Point Turn

2) Optimization Planner:

a) Simple Motion: Figure 4. This diagram uses an N value of 1000 and a dt value of 0.01

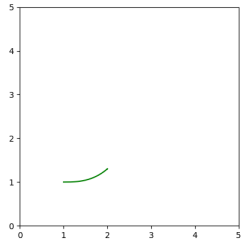

Fig. 4. Optimization Planner Simple Motion

b) Parallel Park: Figure 5. This diagram uses an N value of 1000 and a dt value of 0.01 to plan a parallel parking motion.

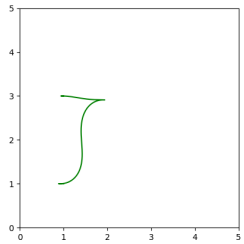

Fig. 5. Optimization Planner Parallel Park

c) Point Turn: Figure 6. This diagram uses an N value of 1000 and a dt value of 0.01 to plan a point turn in place.

3) RRT Planner:

a) Simple Motion: Figure 7. This diagram shows the RRT planner making a simple motion, which is coincidentally the very first planned attempt.

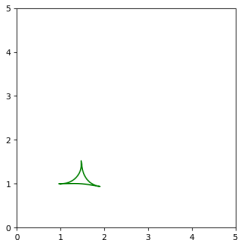

Fig. 6. Optimization Planner point turn

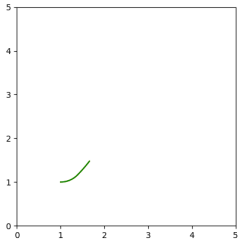

Fig. 7. RRT Simple Motion

b) Parallel Park: Figure 8. This diagram shows the RRT planner performing an elegant parallel park with sweeping curves, as a result of the motion primitives.

c) Point Turn: Figure 9. This diagram shows the RRT planner constructing a point turn solution, which is qualitatively similar to the way in which human drivers perform 3-point turns.

C. Navigation Tasks

1) Optimization Planner:

a) Map 1: Figure 10. This diagram uses an N value of 1000 and a dt value of 0.01 to plan a path from the bottom left corner to the top right, avoiding the obstacles.

b) Map 2: Figure 11. This diagram uses an N value of 1000 and a dt value of 0.01 planning a path from the bottom left to the top right corner. Notice how the path avoids the obstacles and minimizes distance.

2) RRT Planner:

a) Map 1: Figure 12. This diagram shows the RRT planner identifying a route between the obstacles in Map 1.

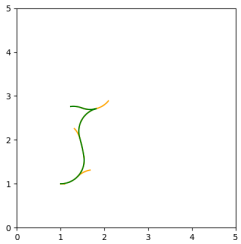

Fig. 8. RRT Parallel Park

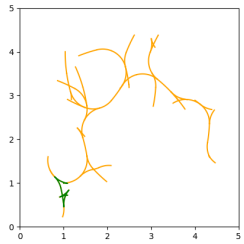

Fig. 9. RRT Planner Point Turn

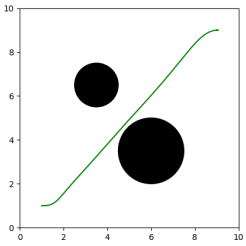

Fig. 10. Optimization Planner path planned for Map 1

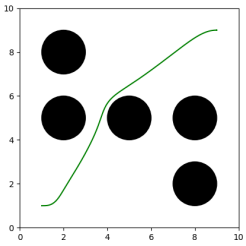

Fig. 11. Optimization Planner path planned for Map 2

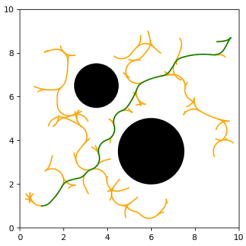

Fig. 12. RRT Planner path planned for Map 1

b) Map 2: Figure 13. This diagram shows the RRT planner identifying a route that largely navigates around the obstacles in Map 2.

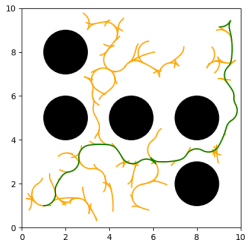

Fig. 13. RRT Planner path planned for Map 2

III. Discussion

A. Planner Performance

The performance of each planner was sporadic, but overall the optimization planner worked best in the most environments. The optimization planner worked well for environments that are easily represented in closed form. The RRT planner was good in situations where there were lots of degrees of freedom and motions that are not easily representable, but is easily sampled. The sinusoidal planner had the most random performances and was most affected by constraints, and took the most tuning with regard to singularities, and was sensitive in general.

B. Comparison

The optimization planner works best for environments which are easily representable in closed-form. and especially when constraints are linear or quadratic. Then we can explicitly solve for a solution without a requirement for significant computational power. On the other hand, RRT is faster and works better for more complex robots and environments. One example where RRT would be better is a robot arm with many degrees of freedom; the constraints here are highly nonlinear, and so any optimization approach is likely to fail. In real-time applications, RRT may be the only option that can produce fast enough results, such as for an autonomous chase car filming an action scene. Finally, planners like the sinusoidal planner rely on special knowledge of the system in question, and are not general-purpose techniques. In the case of these car motions, we found that the sinusoidal planner was excellent for small adjustments made from the initial position, but any adjustment that required going through a singularity consequently required an elaborate maneuver and waste of time.

C. Simulation

In simulation, the open loop plans were not perfect, and our actual final positions differed from the desired final positions. Especially when dealing with obstacles, the simulated execution tended to drift from the desired position over time. This is because open loop control does not consider the error between the current position and the desired trajectory, and does not adjust accordingly. This could be made better by using closed-loop control, or feedback control, to correct for error during execution.

D. Sinusoidal Planner and Obstacles

The sinusoidal planner does not work for navigation tasks because it does not account for obstacles while planning. Since the trajectory is controlled using virtual inputs, obstacles would have to undergo a nonlinear transform to be represented in the same space, which would be difficult.

IV. Bibliography

References

[1] R. Murray, Z. Li, and S. Shankar Sastry, *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994.

[2] R. M. Murray and S. S. Sastry. *Nonholonomic motion planning: steering using sinusoids*. IEEE Transactions on Automatic Control 38.5 (May 1993).

[3] A. Giese. *A Comprehensive, Step-by-Step Tutorial on Computing Dubin's Curves*. https://gieseanw.files.wordpress.com/2012/10/dubins.pdf (October 2012).

V. Appendix

A. Future Improvements

Our group struggled to understand the notation and theory behind the sinusoidal model, despite spending many hours grappling with the spec and the original paper. One way some of this frustration could be mitigated is if the equations on the spec were arranged and explained in more detail. Overall, however, we gained a deeper understanding of the different types of motion planners, their applications, and how to implement them in practice

B. Code Repository

GitHub Link

C. Demonstration Video

YouTube Link

D. Bonus

We believe the learning goals for this assignment were to both develop the skills to take theory and apply it to real life and to be able to read and re-implement research papers. We saw that, especially with the sinusoidal planner, even though the algorithm and idea behind the planner seemed simple, there were many complications and subtleties that had to be accounted for. Another goal was to have an introduction to motion planning beyond A* and other algorithms seen previously. This assignment was very effective at fostering those goals, and throughout the process of writing and debugging these motion planners we gathered an intuitive sense for how they work.