Path Planning for a Nonholonomic System using a Unicycle Model Robot

Tyler Osbey
EECS
College of Engineering
Berkeley, USA
tosbey836@berkeley.edu

Edwin Villalpando
EECS
College of Engineering
Berkeley, USA
edwin23luv@berkeley.edu

Tenzin Norphel
EECS
College of Engineering
Berkeley, USA
tnorphel@berkeley.edu

***Abstract*—Throughout our project, we implemented various path planners, testing each through a set of different manipulation and navigation tasks. We first implemented an optimization planner composed of various constraints and a cost function. In addition, we implemented a sinusoidal path planner. We calculated a series of steps from *t = 0* to *n + 1* time steps using a canonical car model. We considered all constraints, configurations, sample configurations (using the phi limit between -π and π), and then checked the feasibility of all the generated paths. RRT was implemented with the help of Dubins path. A tree is generated to properly execute a feasible trajectory. While the generated trajectory is ideal, problems calculating the steering angle caused undesired results. In the sinusoid planner, we implemented the steer in phi direction, which revolves around itself without forward or backward movement. Along with steer ϕ, it steer y has the initial guess and performed binary search to get a_1 and a_2. As a result, we were able to get the desired trajectory for optimization, sinusoidal, for now.**

I. Methods

To control our unicycle model robot, we implemented an optimization-based planner, an Rapid Exploring Random Tree (RRT) planner, and a sinusoidal path planner on a nonholonomic system. Throughout the path planning procedure, we incorporated different constraints, waypoints, and δt (discretized steps) and generated a trajectory with the help of the plan to pose function.

A. RRT Planner

In our RRT Planner, we first define our start configuration, goal configuration, and configuration space. We follow an algorithm that generates random points within the configuration space until it has reached the goal configuration given some threshold. To accomplish this procedure, we will either hit the maximum allowable iterations or successfully reach our goal configuration. This procedure avoids collisions while creating edges between two nodes as it branches towards the goal state.

To accompany the provided RRT algorithm above, we will incorporate a distance function to calculate how much change is required to go from a given position to the next way-point. To do this we will have to calculate the change in distance for the states (x_i, y_i, θ_i) and (x_{i+1}, y_{i+1}, θ_{i+1}). The change for (x, y) will proceed as seen with a Euclidean space as seen in (1) [1].

$$\rho_1 (x_1, y_1, x_2, y_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{1}$$

To calculate the change in heading direction, θ, we will use a metric to compare angles [1]:

$$\rho_2 (\theta_1, \theta_2) = \min \{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\} \tag{2}$$

You may notice that within our distance calculations, we lack a measurement for the steering angle, ϕ. Since we only care about the cars position and heading angle once it reaches a given state, ϕ is ignored. Adding both of these metrics, we define a metric for distance (3).

$$\begin{aligned} d (x_1, y_1, x_2, y_2, \theta_1, \theta_2) = \rho_1 (x_1, y_1, x_2, y_2) + \\ K (\rho_2 (\theta_1, \theta_2)) \end{aligned} \tag{3}$$

$$\mathrm{K} = 0.5 \tag{4}$$

K in this equation represents a weight given to the change in heading angle calculation such that the distance function is not too dependent on this value. On top of the distance function, we create a local planner that selects from a set of motion primitives to move from one point to the next. These motion primitives represent basic maneuvers that the vehicle can execute within its constraints, similar to how a car navigates its environment. To avoid implementing motion primitives, we used Dubins path within our local planner algorithm. As Dubins path gives the shortest curve to get from one way point to another, it works perfectly for our algorithm. Importing the dubins library in python, we passed in the initial and final states of the trajectory, (x, y, θ), and the turning radius of the car. This gives a feasible path according to the turning and movement constraints of a car.

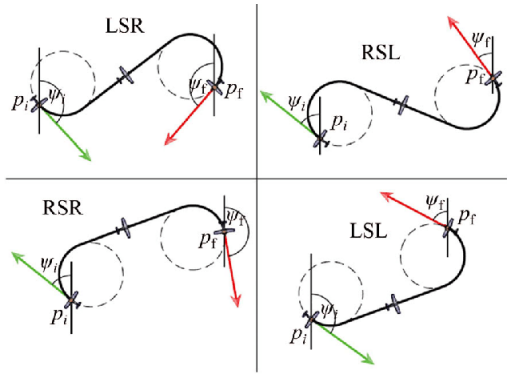

Fig. 1. An example of a Dubins path from point (x_i, y_i, θ_i) to point (x_{i+1}, y_{i+1}, θ_{i+1}) [2]

The given trajectory appears as shown above for different scenarios, where many way-points are included along the generated trajectory. Furthermore, to help our program define this path, we define ϕ at each way-point to be the same as the heading angle, and we define our inputs (v , $\dot{\phi}$).

$$v = \frac{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}{dt} \tag{5}$$

$$\dot{\phi} = \frac{\phi_2 - \phi_1}{dt} \tag{6}$$

However, it should be noted that Dubins path is a forward-driving model only. This will become apparent after looking at generated paths in the Experimental Results section.

B. Sinusoid Planner

We implemented the sinusoidal planner using an approximation of the canonical bicycle model:

$$\begin{aligned} \dot{x} &= v_1 \quad v_1 = cos(\theta)u_1 \\ \dot{\phi} &= v_2 \quad v_2 = u_2 \\ \dot{\alpha} &= \frac{1}{l} tan(\phi) v_1 \quad \alpha = sin(\theta) \\ \dot{y} &= \frac{\alpha}{\sqrt{1-\alpha^2}} v_1 \end{aligned} \tag{7}$$

Here, the calculations for x and ϕ are trivial: they are just the state inputs, which makes sense physically. For the heading state variable α and the horizontal movement state variable y, we introduce sinusoidal inputs (see Eq 8) to avoid changing x and ϕ. However, the values of the amplitudes a_1 and a_2 are unknown, and thus have to be determined.

For y, the value of a_1 cannot be directly solved for. However, we can compute a guess using a binary search algorithm: by setting initial start and end bounds of the search and an initial guess for a_2, we can solve for a_1 by approximating our guess $G = a_1 \frac{\pi}{\omega} \beta_1$ to Δy. We then use the approximated solution for a_1 and our initial guess for a_2 to compute the control inputs v_1 and v_2.

Because the Turtlebot follows the bicycle kinematics model, our controller is subject to constraints on x, y, and ϕ: x and y because our robot is limited in movement both by the environment due to obstacles and by the layout of the wheels on the robot, and ϕ because the robot's wheels are limited in their steering angle. In addition, we have constraints on the input velocity and steering rate. This highly constrained system is not ideal for sinusoidal planning, as it was designed for unconstrained systems [3].

To get the controller to work with the sinusoidal planner, specifically for α and y, we introduced sinusoidal "guesses" for our inputs:

$$\begin{aligned} v_1 &= a_1 sin(\omega t) \\ v_2 &= a_2 sin(\omega t) \end{aligned} \tag{8}$$

Here a_1 and a_2 are amplitudes that we can tune to constrain the state variables. We did not give the heading α a constraint to allow the Turtlebot to rotate completely. We determined that the variables x and ϕ could be directly constrained by adjusting the amplitudes but found it unhelpful for us. The state variable where we most considered input and state constraints was y, as it was the most involved. Much of the work on the sinusoidal planner involved adjusting the start and end search bounds for a_1, the initial guess for a_2, and the constraints on the upper and lower bounds for both amplitudes.

We did not take any steps to address the singularity that occurred when the robot's heading θ reached 90 or -90, but determined that it was caused by the constraints. One possible solution we suggest would be to momentarily switch to another planner, such as Optimization or RRT.

C. Optimization Planner

When dealing with the optimization planner, we tried to adjust different values for waypoints and come up with an optimum path that never gets jolted. In terms of δt5 and number of waypoints (N), we used default values, and we figured that a small amount of N leads to an infeasible path and becomes more riskier in terms of robot trajectory. Also, increasing N leads to higher computation and is time-consuming. Increasing δt makes the car turn too much, and decreasing δt does not provide enough time to turn. To tackle with this, we could increase or decrease with value; however, we didn't encounter any problem with default values, so we go with it.

For this project, we used this optimization equation:

$$\begin{aligned} q^*, u^* = \arg\min \sum_{i=1}^{N} \Big((q_i - q_{\text{goal}})^T Q (q_i - q_{\text{goal}}) + u_i^T R u_i \Big) \\ + (q_{N+1} - q_{\text{goal}})^T P (q_{N+1} - q_{\text{goal}}) \end{aligned} \tag{9}$$

And the corresponding constraints:

$$
\begin{aligned}
\text{s.t.} \quad & q_i \geq q_{\min}, \quad \forall i \\
& q_i \leq q_{\max}, \quad \forall i \\
& u_i \geq u_{\min}, \quad \forall i \\
& u_i \leq u_{\max}, \quad \forall i \\
& q_{i+1} = F(q_i, u_i), \quad \forall i \leq N \\
& (x_i - \text{obs}_{j_x})^2 + (y_i - \text{obs}_{j_y})^2 \geq \text{obs}_{j_r}^2, \quad \forall i, j \\
& q_1 = q_{\text{start}} \\
& q_{N+1} = q_{\text{goal}}
\end{aligned} \tag{10}
$$

When we were tuning δt (timestep size) and N (number of waypoints), we tried different combinations of numbers and came to realize that the default values are δt = 0.005 and N = 1000, which causes the robot in the simulator to move in a straight line. In other cases, the robot circles itself and moves backwards, etc. When there is arbitrary goal state, we need to adjust the value of N waypoints and making fewer N waypoints makes it easier and closer to the goal point without tuning much of the δt.

II. Experimental Results

A. Sinusoid planner

Manipulation Tasks

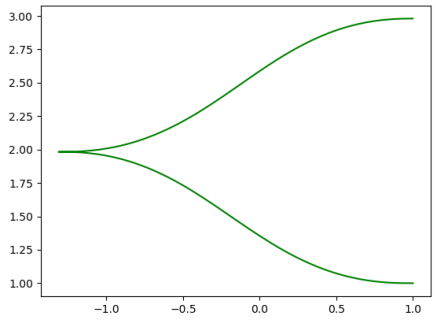

Fig. 2. Plan from position (1, 1, 0, 0) to (1, 3, 0, 0) - parallel parking

For the point turn, after attempting different values for x, y, θ, and ϕ, we kept hitting the singularity and does not show any graph. The reason we are assuming is because we were feeding in the command for values that tells the turtlebot to perform a sharp turn and therefore cannot perform lateral motion.

B. RRT planner

Manipulation Tasks

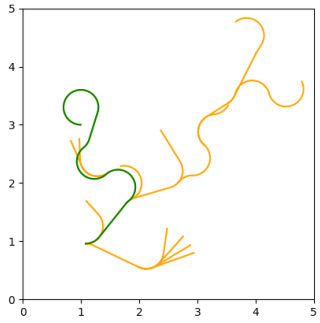

Fig. 3. Plan from position (1, 1, 0, 0) to (1, 3, 0, 0) - parallel parking

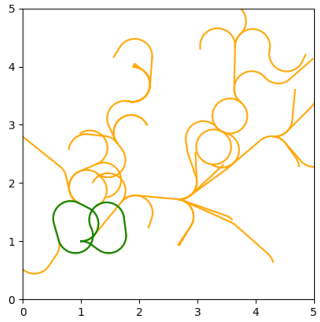

Fig. 4. Plan from position (1, 1, 0, 0) to (1, 1, π, 0) - point turn

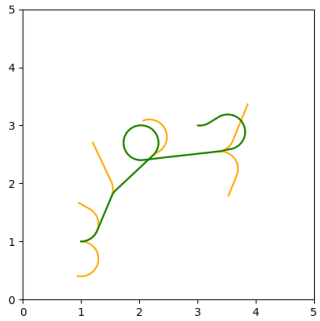

Fig. 5. Plan from position (1, 1, 0, 0) to (3, 3, 3, 0) - custom

Navigation Tasks

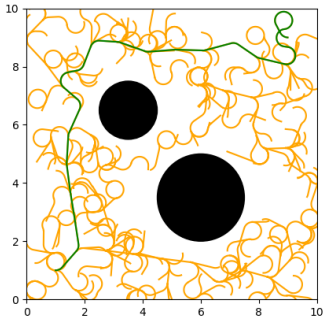

Fig. 6. Plan from position (1, 1, 0, 0) to (9, 9, 0, 0) - map 1

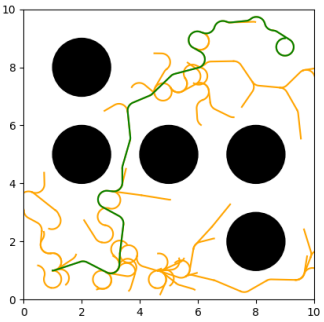

Fig. 7. Plan from position (1, 1, 0, 0) to (9, 9, 0, 0) - map 2

C. Optimization planner

Manipulation Tasks

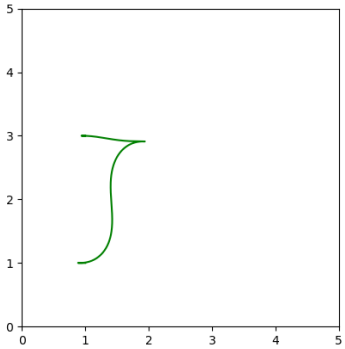

Fig. 8. Plan from position (1, 1, 0, 0) to (1, 3, 0, 0) - parallel parking

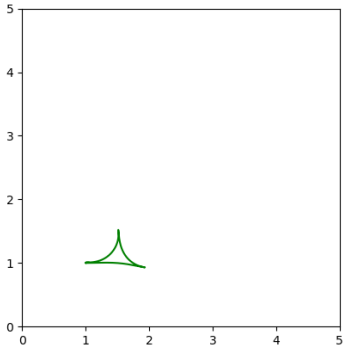

Fig. 9. Plan from position (1, 1, 0, 0) to (1, 1, π, 0) - point turn

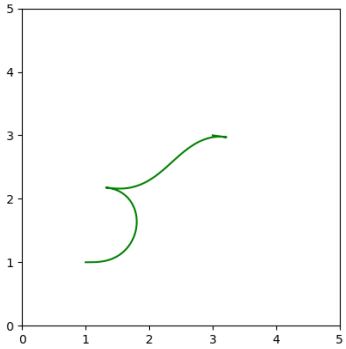

Fig. 10. Plan from position from (1, 1, 0, 0) to (3, 3, 3, 0) - custom

Navigation Task

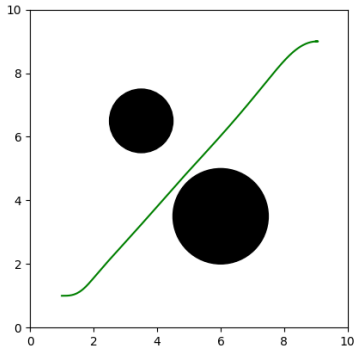

Fig. 11. Map 1 Navigation for optimization planner from (1, 1, 0, 0) to (9, 9, 0, 0)

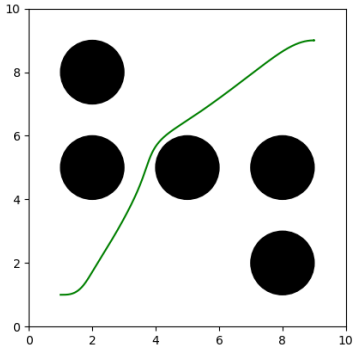

Fig. 12. Map 2 Navigation for optimization planner from (1, 1, 0, 0) to (9, 9, 0, 0)

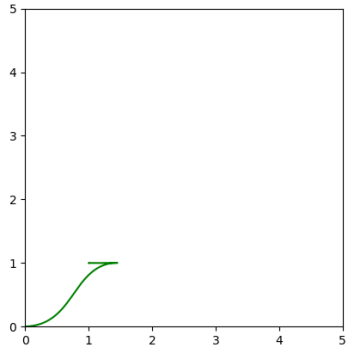

Fig. 13. Turtbot path, starting from [0, 0, 0, 0] and ending on [1, 1, 0, 0]

$$\begin{pmatrix} x_e \\ y_e \\ \theta_e \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{\text{ref}} - x_r \\ y_{\text{ref}} - y_r \\ \theta_{\text{ref}} - \theta \end{pmatrix} \tag{11}$$

$$v_r = v_{ref} \cos(\theta_e) + k_1 x_e, \tag{12}$$

$$\omega = \omega_{ref} + v_{ref} \left(k_2 y_e + k_3 \sin(\theta_e)\right) \tag{13}$$

After choosing Control Lyapunov-based design, we implemented the open-loop input, and it turned out that tuning k_1 affects velocity while k_2/k_3 affects the steering. As we tried different values for all k_1, k_2,and k_3 we observed their impact on the system's performance. After many attempts to get to the one value that the robot is starting to behave as expected, we use k_1 = 0.01, k_2 = -0.001, and k_3 = -0.0001. We realized that tuning for k1 too high and high values of k_2/k_3 make the car turn too much. As seen with our video, it is comparable to the image below, however, the calculated final state is significantly off. The expected final state was [1, 1, 0, 0] as it started from point [0, 0, 0, 0]. The actual final state was [-0.2, -0.2, 0.1, 0]. Other than the obvious tuning that was needed for this controller, the negative x and y values suggest some necessary manipulation within our `bicycle_converter.py` file. Additionally, some tweaking may be necessary for the Lyapunov-based controller equations. By definition, k_1, k_2,and k_3 should all be greater than 0 [5]. The original derivation provided resulted in a turtlebot that would only reverse and drive forwards, no matter the generated path or tuning values. Thus, some minor tweaking was done to get some better results, flipping the reference states and the target states in the matrix-vector multiplication, along with violating a couple of definitions. Further testing was desired; however, this was unable to take place.

III. Discussions

For the optimization planner, we used different values for x, y, θ and ϕ, and depending on individual values, sometimes it goes out of bounds, and sometimes it collides with the obstacles, and we had to incorporate some padding to avoid collisions with the obstacles. Changing the padding made a huge difference in terms of trajectory. We implemented under the function "check-collision" that takes in the configuration c and checks for collision c. Overall, the optimization plan was able to generate feasible paths given the maps without issue. Problems may arise with more complex maps with more obstacles, such as a maze. A maze would require more waypoints, and as discussed previously, it is quite tedious to tune.

RRT also worked great, as it was able to generate any path given any configuration within the configuration space. Regardless of its ability to generate trajectories, we run into a problem when attempting to parallel park. As described previously, Dubins path is forward driving only. To fix this issue, Dubins path would need to be paired with the Reeds–Shepp curve. Due to the randomness of RRT, although the paths generated were feasible, it would often generate a path that is not optimized as seen with RRT in Map 1.

The sinusoid planner worked great for points within the constraints and for points that did not involve a singularity angle. The sinusoidal planner failed to generate a viable path for points outside of the bounds we set for the integral for the state variable y and for points that involved a turn with the singularity angles. We did not address the singularity issue, but to fix the integral bound issue we simply increased the start and end search bounds for the amplitude a_1 for y, which allowed us to reach points that were further out.

When comparing the optimization planner with other planners like RRT and sinusoidal, its easier to implement in the sense that most of the time the trajectory follows as expected, unlike RRT and sinusoidal, where we get cumbersome trajectories. In our sinusoidal planner, we often hit singularities and nan (not a number) issues which are unavoidable in some scenarios. The sinusoidal planner is beneficial when a combination of reverse and forward driving is needed for a

given trajectory. This combination makes it really only useful for lateral movements that require a car to move sideways. It struggles with constraints and with obstacles. This is why the planner does not work well with navigation tasks. How might you modify the steering with sinusoids algorithm to work with obstacles?

As we implemented the open loop controller, we came to know that u_1 and u_2 represent v_{ref} and ϕ respectively. The optimization planner performed flawlessly with little to no error found between its predicted and actual final states. The RRT algorithm exhibited predictable yet fundamentally flawed behavior. We were told to ignore simulation error with RRT, thus, not much debugging was done to uncover and fix this mistake. We did make a note that to fix RRT's simulation, we most likely had to fix the calculations for ϕ at each state within the generated Dubins path would require a more in depth derivation to be successful. This was inferred because setting it to a constant zero for each way-point along a trajectory made the car drive straight, and setting it to any other constant made it turn in that direction until the simulation terminated. For sinusoid, with the only path it was able to generate without a singularity or Nan issue, it had slight error of about 1 unit in the y coordinate. This value was then reduced, with some tuning of the variables, to about 0.5.

We used the Control Lyapunov Based Design, where the math equations are expressed above (11)(12)(13). In (11) we are trying to minimize the errors, which are denoted as x_e, y_e, θ_e, and we have reference position as x_{ref}, y_{ref}, and θ_{ref}, and and target positions as x_r, y_r, θ_r. Since we need to tune k_1, k_2, k_3, based on how we implemented tracking error, we used (12) and (13) for v_r and ω. k_1 directly affects x_e, k_2 directly affects y_e, and k_3 directly affects θ_e. For more thoughts, please read the last paragraph of the Experimental Results section.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to EE106B Distinguished Professor S. Shankar Sastry for his guidance and support throughout the project. Special thanks to our lab TAs, K. El-Refai and Yarden Goraly, for their invaluable assistance in the development and conceptual understanding of the optimization planner, RRT Planner (Rapid Exploring Random Tree), and sinusoid planner with their implementations on TurtleBot. Additionally, we extend our appreciation to Jaeyun Stella for her support in logistics and policy-related aspects of the project.

REFERENCES

[1] S. M. LaValle. Planning algorithms. Cambridge university press, 2006.
[2] Chen, Qingyang & Lu, Ya-fei & Jia, Gao-wei & Li, Yue & Zhu, Bing-jie & Lin, Jun-can. (2018). Path planning for UAVs formation reconfiguration based on Dubins trajectory. Journal of Central South University. 25. 2664-2676. 10.1007/s11771-018-3944-z.
[3] Hoang Nguyen, Han. A Guide to Sinusoidal Planning. 2023.
[4] R. M. Murray and S. S. Sastry, "Nonholonomic motion planning: steering using sinusoids," in *IEEE*
[5] Transactions on Automatic Control, vol. 38, no. 5, pp. 700-716, May 1993, doi: 10.1109/9.277235. B. Paden et al. A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. 2016.

APPENDIX

** Note ** : The videos will only contain one RRT video because RRT simulation was consistent (it just spun in circles). Sinusoid gave a singularity for the custom data point.

- Github: https://github.com/RobotT836/EE106B
- Videos: https://tinyurl.com/106BProject2