# Nonholonomic Control

Ziteng (Ender) Ji Yuvan Sharma

***Abstract*—This project explores nonholonomic path planning for a front-wheel steering car model, also known as the bicycle model. The primary goal is to develop multiple motion planners that respect the system's nonholonomic constraints, including an optimization-based planner, a modified Rapidly-exploring Random Tree (RRT) planner, and a sinusoidal path planner. The optimization-based planner, implemented using CasADi, formulates the planning problem as a nonlinear optimization problem with constraints on state variables, control inputs, and obstacle avoidance. This planner minimizes a quadratic cost function while ensuring dynamically feasible trajectories. The RRT planner extends traditional sampling-based motion planning by incorporating system-specific motion primitives. The sinusoidal planner, inspired by Sastry and Murray, generates feasible paths through sinusoidal steering maneuvers. This project demonstrates the challenges of planning for nonholonomic systems and evaluates these different approaches for trajectory generation, emphasizing the trade-offs between optimality, computational efficiency, and feasibility in real-world applications.**

## I. Introduction

Path planning for nonholonomic systems is crucial in robotics and autonomous navigation, where vehicles face motion constraints that limit their maneuverability. This project implements and compares three motion planning approaches for a bicycle-model robot: an optimization-based planner, a Rapidly-exploring Random Tree (RRT) planner, and a sinusoidal path planner.

The optimization-based planner solves a nonlinear optimization problem to generate a feasible path while enforcing constraints on state, control inputs, and obstacles. The RRT planner focuses more on randomly sampling the configuration space to build up a network of system-specific motion primitives, and lastly, the sinusoidal planner constructs trajectories using sinusoidal steering maneuvers.

These methods are widely applicable in autonomous driving, robotic navigation, and industrial automation, where balancing computational efficiency and motion feasibility is essential. This project explores their strengths and trade-offs to enhance motion planning for real-world systems.

## II. Method

### A. Optimization Planner

As described previously, the optimization-based planner formulates nonholonomic path planning as a nonlinear optimization problem.

*1) Optimization Planner Implementation:* The trajectory is discretized with state variables $q \in \mathbb{R}^{4\times(N+1)}$, where each column represents

$$q_t = \left[x_t, y_t, \theta_t, \phi_t\right]^T, \tag{1}$$

and control inputs $u \in \mathbb{R}^{2\times N}$, where

$$u_t = \left[u_1, u_2\right]^T. \tag{2}$$

We set $N = 1000$ and $\delta t = 0.01$ seconds to balance trajectory smoothness and computational efficiency; a detailed explanation can be found in the next section.

To warm-start the solver, an initial guess is generated via linear interpolation between $q_{\text{start}}$ and $q_{\text{goal}}$, ensuring a straight-line initialization in configuration space, even if it does not fully respect nonholonomic constraints. The system dynamics are discretized using the Euler method:

$$x_{t+1} = x_t + u_1 \cos(\theta_t) \cdot \delta t \tag{3}$$
$$y_{t+1} = y_t + u_1 \sin(\theta_t) \cdot \delta t \tag{4}$$
$$\theta_{t+1} = \theta_t + \frac{u_1}{L} \tan(\phi_t) \cdot \delta t \tag{5}$$
$$\phi_{t+1} = \phi_t + u_2 \cdot \delta t \tag{6}$$

where $L$ is the axle-to-axle length. This ensures computational efficiency while capturing nonholonomic behavior.

The cost function used for the optimization problem minimizes deviation from the goal and excessive control effort. Specifically, the state cost penalizes deviation from $q_{\text{goal}}$ using:

$$(q_t - q_{\text{goal}})^T Q (q_t - q_{\text{goal}}) \tag{7}$$

while the control cost minimizes input magnitudes:

$$u_t^T R u_t \tag{8}$$

and the terminal cost ensures proximity to $q_{\text{goal}}$:

$$(q_N - q_{\text{goal}})^T P (q_N - q_{\text{goal}}) \tag{9}$$

where $Q$, $R$, and $P$ balance tracking accuracy and control effort.

Constraints are added to the problem to ensure feasibility by enforcing state and control limits:

$$q_{\min} \le q_t \le q_{\max}, \quad u_{\min} \le u_t \le u_{\max} \tag{10}$$

while obstacle avoidance is modeled as circular constraints:

$$(x_t - x_{\text{obs}})^2 + (y_t - y_{\text{obs}})^2 \ge r_{\text{obs}}^2 \tag{11}$$

for each obstacle at $(x_{\text{obs}}, y_{\text{obs}})$ with radius $r_{\text{obs}}$. Initial and final state constraints ensure the trajectory starts and ends at the correct locations.

*2) Trajectory Discretization: Choosing $N$ and $\delta t$:* In the current implementation, we chose $N = 1000$ and $\delta t = 0.01$. These values were selected to balance trajectory smoothness and computational efficiency. Higher $N$ allows for a finer resolution of the path, ensuring the system respects the nonholonomic constraints while moving towards the goal. Smaller $\delta t$ results in more frequent control updates, improving accuracy but increasing computational cost. The product $N\delta t = 10$ seconds ensures the planner has sufficient time to generate a feasible path within a reasonable execution window.

*3) Generalize $N$ and $\delta t$ for Arbitrary Goal States:* While the chosen values work well for this scenario, they must be adapted dynamically for different goal states. Several heuristics can guide this selection. Firstly, a distance-based heuristic can be used, where the required number of timesteps should scale with the Euclidean distance between the start and goal positions:

$$N \propto \frac{\|q_{\text{goal}} - q_{\text{start}}\|}{v_{\max}\delta t}$$

where $v_{max}$ is the maximum velocity of the robot. This ensure that longer trajectories have a sufficient number of waypoints. We can also use velocity-dependent adjustment to decide the size of the step $\delta t$, as higher velocities require smaller $\delta t$ to prevent large discretization errors, and lower velocities allow for larger timesteps. Lastly, the value of $N$ can be adjusted in scenarios where the solver detects constraint violations (e.g., collision with obstacles, dynamic infeasibility); iteratively increasing $N$ until feasibility is achieved is a good heuristic for this purpose.

### B. RRT Planner

The RRT planner is a widely used sampling-based motion planning algorithm designed for high-dimensional and constrained configuration spaces. Unlike traditional graph-based methods, RRT incrementally builds a tree that efficiently explores feasible paths by randomly sampling the configuration space and expanding the tree in dynamically feasible directions. This makes RRT particularly well-suited for nonholonomic systems, such as the front-wheel steering car model, where constraints limit the set of possible motions.

*1) RRT Planner Implementation & Design:* Our implementation of the RRT planner follows the standard framework while incorporating modifications to accommodate the nonholonomic constraints of the front-wheel steering car model. The planner operates in a four-dimensional configuration space $C$, where each state is defined as

$$q = (x, y, \theta, \phi),$$

representing the vehicle's position, heading angle, and steering angle. The algorithm incrementally expands a search tree from an initial configuration $q_{\text{start}}$ toward a goal configuration $q_{\text{goal}}$.

At each iteration, the planner first samples a new random configuration $q_{\text{rand}}$ from the configuration space. To determine the next expansion step, the nearest neighbor search identifies the closest node $q_{\text{near}}$ in the existing tree using a distance metric (which is discussed in the next section). Instead of directly interpolating between $q_{\text{near}}$ and $q_{\text{rand}}$, a local motion plan is generated to respect the vehicle's dynamic constraints. This motion plan consists of a motion primitive (selected from a set), which applies dynamically feasible control inputs over a short time horizon $\Delta t$ to get the vehicle as close to $q_{\text{rand}}$ as possible. To accurately check which primitive works best, we simulated the dynamics of the robot and calculated the end position for each primitive. A prefix of the generated motion plan is then added as an edge to the graph, with the endpoint being $q_{\text{new}}$, the point where the prefix ends. The RRT algorithm then loops over this process, with collision checks used to avoid adding incorrect or infeasible configurations and paths.

*2) Distance Metric:* In RRT-based motion planning, the choice of the distance function is crucial, as it determines how new configurations are selected for expanding the search tree. Directly computing Euclidean distance is infeasible for our use case due to the need to accurately calculate angular distance as well. We thus use the complex number representation of SO(2) space to convert the angle $\theta$ into $a = \cos\theta, b = \sin\theta$. This gives us the new tuple $(x, y, a, b)$ for each state, following which the standard Euclidean norm is used as shown in the equation below. This approach was borrowed from Lavalle [2].

$$d((x_1, y_1, \theta_1), (x_2, y_2, \theta_2)) \\ = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (a_2 - a_1)^2 + (b_2 - b_1)^2}$$

We found this metric to be effective in accurately capturing distances in the configuration space, as it also takes into account the periodic nature of $\theta$.

*3) Sampling Method:* In addition to the distance metric, the sampling strategy plays a significant role in improving the efficiency and speed of the RRT planner. Instead of uniformly sampling the entire configuration space, our implementation incorporates the goal zoom strategy to improve search efficiency. This strategy involves refining the search by sampling within a shrinking radius centered around $q_{\text{goal}}$. Specifically, the algorithm selects samples from a ball of radius $\min d(x_i, g)$, where $x_1, \ldots, x_n$ are the existing nodes in the tree, and $d$ is the distance function. This technique helps concentrate sampling in promising regions, ensuring finer granularity as the tree nears the goal. We found this strategy to improve the speed of RRT significantly compared to when goal bias sampling was used, which is mostly because goal zoom allows for random sampling around the goal while goal bias strictly returns the goal itself. In cases where reaching the goal exactly is difficult or even infeasible for the motion primitives from the current tree, it becomes difficult for the tree to grow in meaningful ways with goal bias sampling.

### C. Sinusoid Planner

The sinusoidal planner offers a structured approach to nonholonomic control by leveraging sinusoidal control inputs to achieve smooth and feasible trajectories. This section details the design and implementation of the sinusoidal planner,

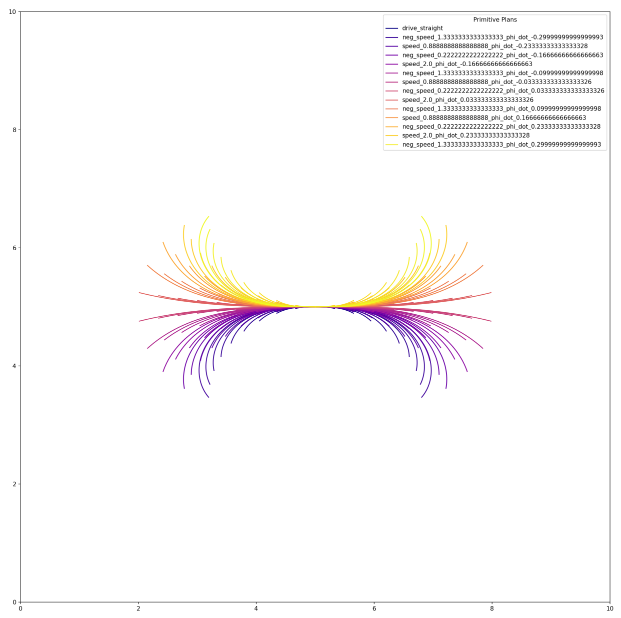

Fig. 1. The motion primitives used for RRT, graphed with start point $[5, 5, 0, 0]$. The final version we used had around 200 primitives, made from different combinations of linear and angular velocity. All primitives are graphed above, with the legend only having a subset for clarity.

followed by an analysis of how input constraints, state bounds, and singularities were addressed in the formulation.

*1) Sinusoid Planner Implementation & Design:* The sinusoidal planner generates motion trajectories by sequentially modifying $x, \phi, \alpha,$ and $y$ to reach the goal state. Given an initial state $s_0 = (x_0, y_0, \theta_0, \phi_0)$ and a goal state $s_g = (x_g, y_g, \theta_g, \phi_g)$, the planner produces a trajectory that adheres to the bicycle model dynamics.

The planning process consists of four key steering functions: steer$(x)$, steer$(\phi)$, steer$(\alpha)$, and steer$(y)$, each generating control inputs $u_1$ (linear velocity) and $u_2$ (steering rate).

*2) Input Constraints & State Bounds:* To enforce input constraints, the planner must restrict the magnitude of control inputs such that

$$|u_1| \leq u_{1,\max}, \quad |u_2| \leq u_{2,\max}.$$

Since steer$(x)$, steer$(\phi)$ set a constant linear/angular velocity, we implement the input constraints for these functions heuristically by giving them enough time to complete the motion. Additionally, we found that limiting the magnitudes for the sinusoidal steering in steer$(\alpha)$, steer$(y)$ based on the relation $v_1 = u_1 \cdot \cos\theta$ was sufficient for generating plans that satisfied all input constraints. We did not explicitly set any state bounds as we found the trajectories that were generated to be well within the bounds of the configuration space, and enforcing the input constraints was enough to get smooth, maneuverable trajectories.

*3) Extra Credit - Singularity:* A major challenge in sinusoidal motion planning is handling singularities that arise when the robot's orientation approaches $\theta = \pm 90°$, where standard transformations become ill-defined. These singularities occur because the forward velocity component $u_1$ is defined in terms of $\cos(\theta)$, which approaches zero as $\theta$ approaches $\pm 90°$, making direct control infeasible.

**Algorithm 1** Planning Algorithm for Sinusoidal Steering

```
Require: Initial state $(x_i, \phi_i, \alpha_i, y_i)$ and goal state $(x_f, \phi_f, \alpha_f, y_f)$
Ensure: A plan to steer from the initial state to the goal state
 1: if Path does not pass through singularity at $\theta = \pm 90°$ then
 2:     Generate a plan for $x, \phi, \alpha, y$ using the regular planner.
 3:     Chain the four plans together and return the complete plan.
 4: else
 5:     Split the Path: Split into three segments: start $\to$ mid1, mid1 $\to$ mid2, mid2 $\to$ goal
 6:     First Segment: Use the regular planner to generate a sinusoidal plan, since this segment does not have singularity.
 7:     Second Segment: Use the alternate model to steer from the first middle state to the second middle state, and thus through the singularity.
 8:     Third Segment: Switch back to the regular planner and steer from mid2 to goal.
 9: end if
10: return the complete plan.
```

To mitigate this issue, our planner includes a singularity detection mechanism. When a potential singularity is detected, the trajectory is divided into three segments. The first segment initiates motion using conventional sinusoidal controls. In the second segment, where the singularity is expected to occur, an alternative control formulation is applied. The change in dynamics of this alternate planner can be summarized as:

$$\dot{\alpha} = -\frac{1}{l}\tan(\phi) \cdot u_1, \alpha = \cos\theta, v_1 = \sin\theta \cdot u_1$$

This planner steers the states to their desired positions in the order $y, \phi, \alpha, x$ rather than the usual order $x, \phi, \alpha, y$. As a result, to implement this alternate planner, we wrote four new functions steer_alternate$(y)$, steer_alternate$(\phi)$, steer_alternate$(\alpha)$, and steer_alternate$(x)$. These functions work very similarly to the normal planner, with small changes made to reflect the new dynamics. We note that it is equally valid to use the normal planner to steer $x, \phi$, use the alternate planner to steer $\alpha$ through the singularity and use the normal planner to steer $y$ again; we chose to fully implement the alternate planner mostly as a learning exercise. After the model is steered through the singularity, the normal planner is used once again to steer the vehicle through the final third of the trajectory. This temporary inversion ensures smooth passage through the singular region while maintaining kinematic feasibility. The complete algorithm is presented in Algorithm 1.

## III. Experiments

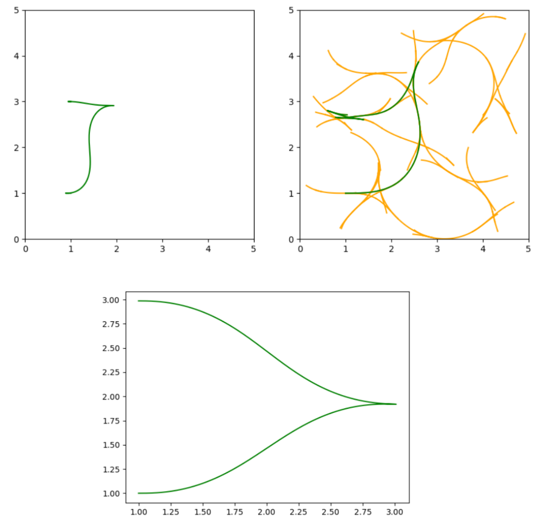

Fig. 2. Generated Paths for Manipulation task (1, 3, 0, 0), with Optimization Planner on the left, RRT Planner on the right, and Sinusoidal Planner at the bottom. All three planners completed this task with accuracy in simulation, especially the Optimization and Sinusoidal planners.

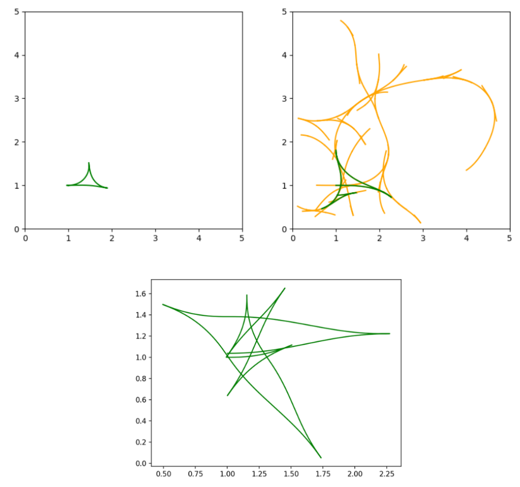

Fig. 3. Generated Paths for Manipulation task (1, 1, $\pi$, 0), with Optimization Planner on the left, RRT Planner on the right, and Sinusoidal Planner at the bottom. All three planers were able to complete the task when tested in simulation.

*1) Plots & Graphs:* Figures 2, 3, and 4 illustrate the trajectory plots for the manipulation tasks. In these visualizations, the Optimization Planner is presented on the left, the RRT Planner on the right, and the Sinusoidal Planner at the bottom. The green line represents the final paths generated by the planners and executed on the robot, while the yellow lines for the RRT Planner denotes the RRT tree (composed of motion primitives) that was constructed to reach the goal. All planners followed the trajectory with precision in executing the manipulation tasks. The accuracy of trajectory tracking with error graphs is discussed in Section IV.

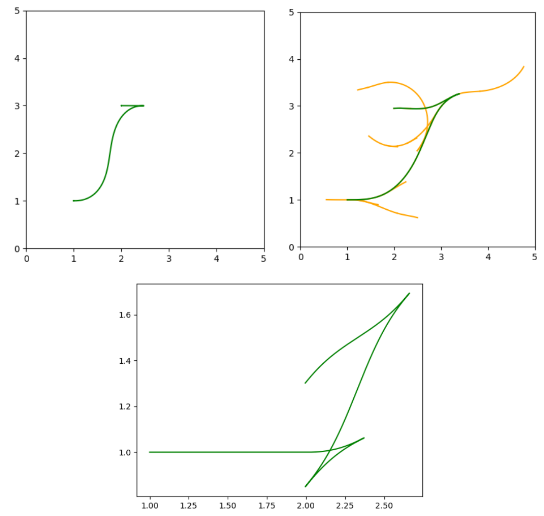

Fig. 4. Trajectory for Third Manipulation task, with Optimization Planner is presented on the left, the RRT Planner on the right, and the Sinusoidal Planner at the bottom. For Optimization Planner and RRT Planner, we chose (2, 3, 0, 0) as the destination, and for Sinusoidal Planner, we used the trajectory (2, 1.3, 0.7, 0)

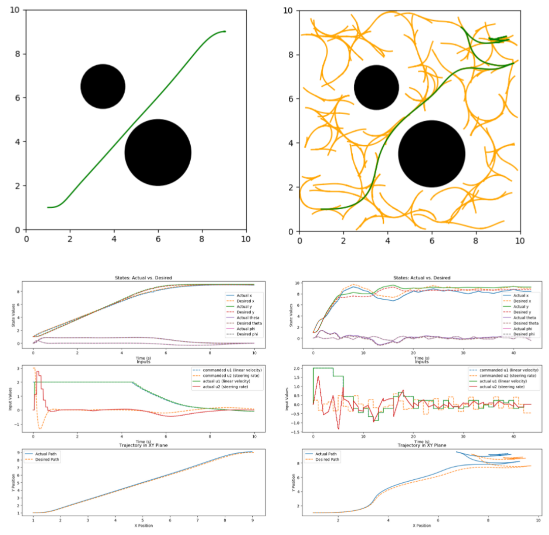

Fig. 5. Trajectory for Navigation Task of Map 1, with the Optimization Planner on the left and the RRT Planner on the right. At the bottom, we have comparisons for desired versus actual state values over time, input values over time, and the robot's trajectory visualized in the XY plane. The error plots shown are for our final proportional controller.

Figures 5 and 6 further depict trajectory plots for manipulation tasks, with the Optimization Planner on the left and the RRT Planner on the right. Additionally, we provide plots for all four state values (desired versus actual) over time, input values (commanded versus actual) over time, and the robot's path visualized on the XY plane. These are also included

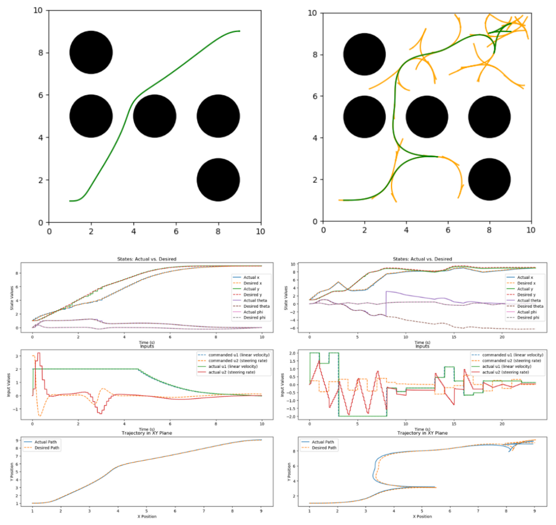

Fig. 6. Trajectory for Navigation Task of Map 2, with the Optimization Planner on the left and the RRT Planner on the right. At the bottom, we have comparisons for desired versus actual state values over time, input values over time, and the robot's trajectory visualized in the XY plane. The error plots shown are for our final proportional controller.

in Section VI, along with similar plots for the manipulation tasks. All these plots were created using the proportional (P) controller, more details are in Section IV.

*2) Real World TurtleBot:* In addition to implementing and testing the three different planners in simulation, we also deployed all three planners on a real turtlebot, with three controllers: open loop, proportional feedback (P), and a Lyapunov controller (as described in Paden et. al. [3]). We found that all three controllers track trajectories with acceptable accuracy; the path along with the plots for each controller are presented in Figure 7 (the optimization planner was used for all these experiments). Further analysis is presented in Section IV.

## IV. Discussion

*1) Performance & Comparison:* Despite encountering some difficulties while developing and optimizing the planners, all three finally completed the required tasks. Comparing performance in terms of generated paths, we believe the optimization planner is the best since it generates simple yet smooth trajectories that are easy to track even with an open loop controller. For instance, the trajectory generated by the optimization planner for the three point turn (Figure 3) is exactly a three point turn with smooth motions. In contrast, the RRT planner, due to its inherent randomness, finds a more complex path. We found that an open loop controller (as well as a feedback controller) is worse at tracking RRT paths, which is to be expected since these are several motion primitives changed together which are not necessarily easy to track one after the other. Similarly, the sinusoid planner also comes up with a more complicated motion since it divides the path into three segments due to the singularity present in this task.

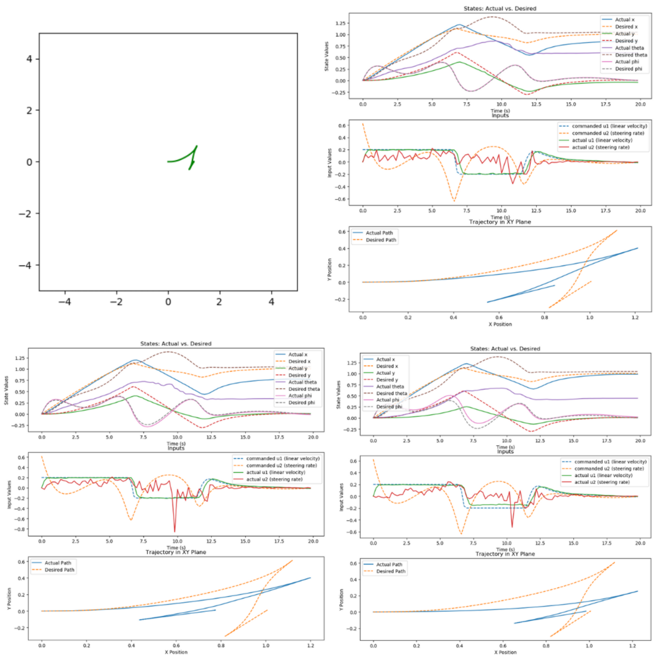

Fig. 7. Top Left: Path generated by the optimization planner for the real turtlebot to go from [0,0,0,0] to [1,0,$\pi/3$, 0]. Top Right: Plots for desired versus actual states, inputs and XY trajectory for the open loop controller. Bottom Left and Right: corresponding plots for the proportional controller and the Lyapunov controller.

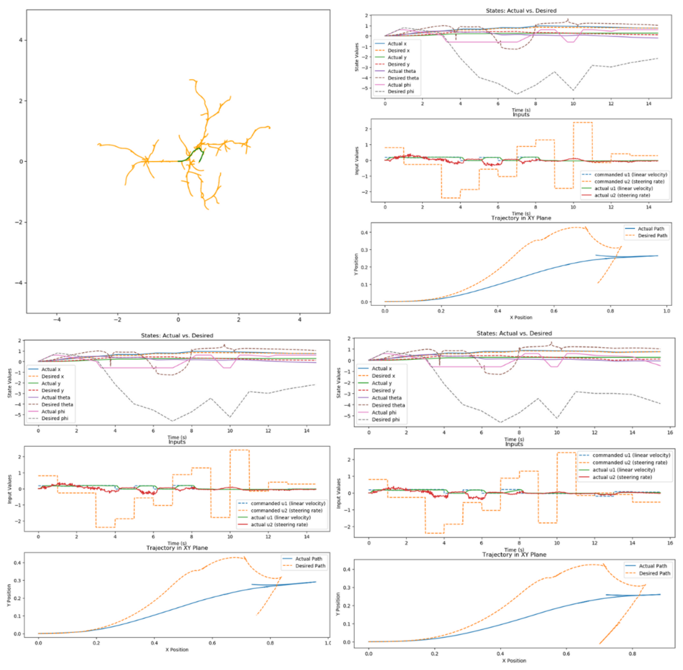

Fig. 8. Top Left: Path generated by the RRT planner for the real turtlebot to go from [0,0,0,0] to [1,0,$\pi/3$, 0]. Top Right: Plots for desired versus actual states, inputs and XY trajectory for the open loop controller. Bottom Left and Right: corresponding plots for the proportional controller and the Lyapunov controller.

The difference in trajectory tracking can be seen in the bottom plots in Figures 5 and 6. These plots are for a proportional controller, and it can be seen that for both maps, the controller is able to accurately track the open-loop path. However, it struggles with RRT when the generated path makes sudden changes in direction or speed.

In terms of implementation difficulty, the optimization planner was the easiest to implement and did not require extensive tuning. However, RRT was tricky because of the problem of designing diverse primitives. We spent a lot of time perfecting our primitives and local planning algorithm, and eventually settled on an approach that simulates the discrete-time dynamics of the car to find the end-position of each primitive, before using the distance function between these end-positions and the real goal to choose the best primitive. The sinusoidal planning was the hardest to understand mathematically, but relatively easier to implement.

We note that each planner has its own advantages. The optimization planner's advantages include ease of implementation and simple trajectories, while RRT is great at exploring complicated environments due to its ability to grow a network of primitives and later chain them together. Sinusoidal paths, on the other hand, are best suited for navigating an empty environment in their current version since obstacle avoidance is not easy to implement for that framework.

*2) Open Loop:* Running our planners' generated paths in an open-loop manner, we found that the simulated vehicle is able to accurately track the optimization and sinusoidal trajectories. It also follows the RRT trajectory, but a little less accurately due to the random nature of the chosen primitives. We also found that for the three point turn, despite our sinusoidal generated trajectory not violating any constraints, the simulator would get stuck (presumably because the car would go near the $x = 0$ bound), and so the video we recorded for this trajectory is done at $(2, 2)$ with the same path.

*3) Sinusoidal Planner & Navigation Tasks:* The sinusoidal planner does not work for navigation tasks in its current form since it does not take into account any obstacles in the environment. It simply considers state and input bounds, and outputs a path. To adapt this planner for navigation, one might consider an RRT-style approach where, if an obstacle blocks the path generated by the sinusoid planner from the start to the goal, we split the path into several segments that do not intersect with obstacles and try to individually complete them with sinusoids. If these subpaths still collide, we repeat and break down the segment again, until we get a non-colliding path.

*4) Control Law:* We implemented both a proportional (P) controller and a Lyapunov controller. For the proportional controller, similar to what we did in project 1, the error in the inputs at time $t$ is defined as:

$$e(t) = u_d(t) - u(t) \tag{12}$$

The control input for the robot is adjusted using a proportional feedback approach:

$$u(t+1) = u_d(t+1) + K_p e(t) \tag{13}$$

where we have $u(t+1)$ as the feedforward term, which ensures that the robot follows the nominal desired trajectory. $K_p e(t)$ is the proportional term. In practice, we set $K_p = 0.05$ for both the linear velocity and the steering rate for the real turtlebot, and $K_p = 0.02$ for simulation.

For the Lyapunov controller, we follow the design in Paden et. al [3], which is a natural fit since the real Turtlebot is also a unicycle model. Specifically, given a target position $(x_{ref}, y_{ref}, \theta_{ref}, \phi_{ref})$ and the open loop input $(v_{ref}, \omega_{ref})$ for the current timestep, we first find the errors as

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

where $x_r, y_r, \theta$ represents the current state variables. Then, finding the new input values as

$$\begin{aligned} v &= v_{ref}\cos(\theta_e) + k_1 x_e, \\ \omega &= \omega_{ref} + v_{ref}\left(k_2 y_e + k_3 \sin(\theta_e)\right). \end{aligned}$$

creates the Lyapunov control law, with $k_1, k_2, k_3$ being tunable parameters. In practice, we set $k_1 = 0.2, k_2 = 0.2, k_3 = 0.3$.

*5) Analysis of Results on Real Turtlebot:* We originally attempted to use a PD controller but found tuning to be extremely difficult, and then resorted to just using a proportional controller. Even then, it was difficult to improve on the open loop controller, with the plots in Figure 7 showing the small improvement that the feedback controller gives over the open loop controller. In contrast, the Lyapunov controller works much better, and was also easier to tune.

We also found that because the turtlebot sways and jerks while reversing, which we hypothesize occurs because of a small protusion in its structure near the rear end. As a result, we found it was difficult for the turtlebot it was difficult to accurately track the desired rotation with any of the three tested controllers, as can be seen from the plots: the general trend $\theta$ is tracked, but with a significant offset. The plots also show that all three controllers struggle to accurately follow the desired yaw. Testing with the RRT planner instead of the optimization planner gave similar results; the plots are presented in Figure 8. We note that tracking results for the RRT planner are worse, which is to be expected because of the chained primitives that can require sudden changes in velocity and direction.

## V. Bibliography

[1] R. M. Murray and S. S. Sastry, "Nonholonomic Motion Planning: Steering Using Sinusoids," IEEE Transactions on Automatic Control, vol. 38, no. 5, pp. 700–716, May 1993.

[2] Lavalle, Steven M.. "Planning Algorithms". Cambridge University Press, 2006.

[3] Paden, Brian, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. "A survey of motion planning and control techniques for self-driving urban vehicles." IEEE Transactions on intelligent vehicles 1, no. 1 (2016): 33-55.

## VI. Appendix

Our video with demonstrations for each planner can be found here: https://drive.google.com/file/d/1toTE8Z87vMwxjUtRET8WYT1WLad4j0yz/view?usp=sharing.

Our github repository can be found here: https://github.berkeley.edu/yuvan/project2.

Below we have the plots we mentioned in Section III. They show the desired versus actual states, the commanded versus actual inputs, and the trajectory visualized in the XY plane for all manipulation tasks and planners, as well as the navigation tasks for optimization and RRT planners.

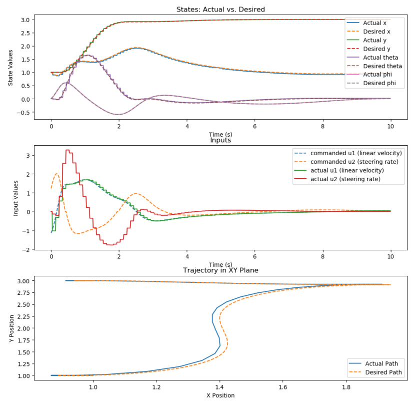

Fig. 9. Optimization Planner Manipulation Task (1, 3, 0, 0) Plot

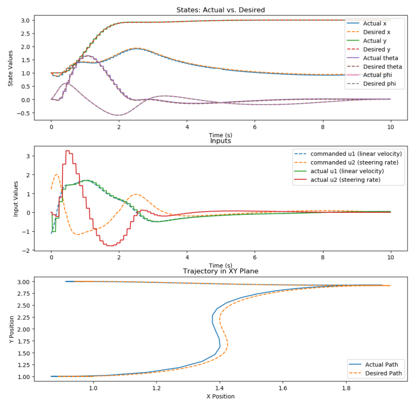

Fig. 10. Optimization Planner Manipulation Task (1, 1, $\pi$, 0) Plot

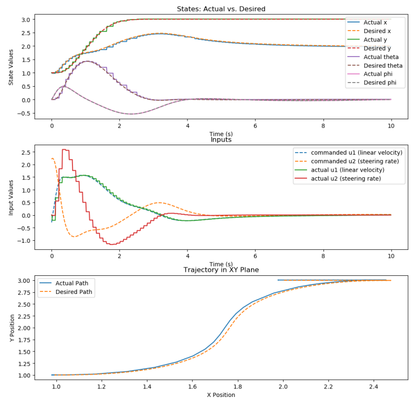

Fig. 11. Optimization Planner Manipulation Task (2, 3, 0, 0) Plot

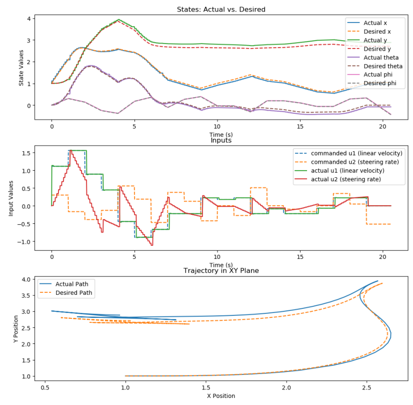

Fig. 12. RRT Planner Manipulation Task (1, 3, 0, 0) Plot

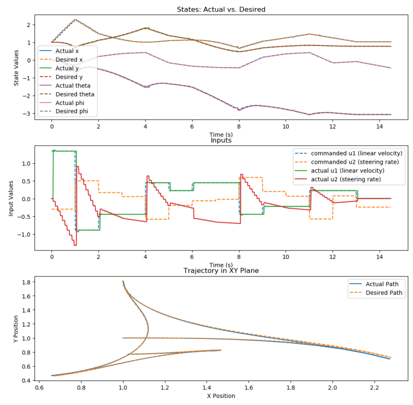

Fig. 13. RRT Planner Manipulation Task (1, 1, $\pi$, 0) Plot

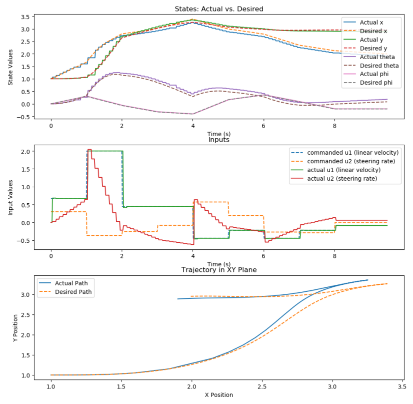
Fig. 14. RRT Planner Manipulation Task (2, 3, 0, 0) Plot

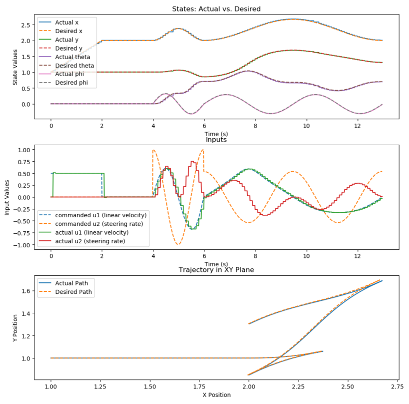
Fig. 17. Sinusoidal Planner Manipulation Task (2, 1.3, 0.7, 0) Plot

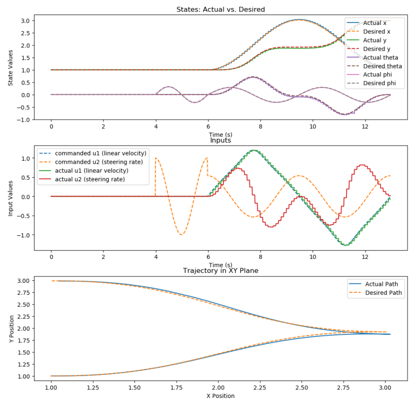
Fig. 15. Sinusoidal Planner Manipulation Task (1, 3, 0, 0) Plot

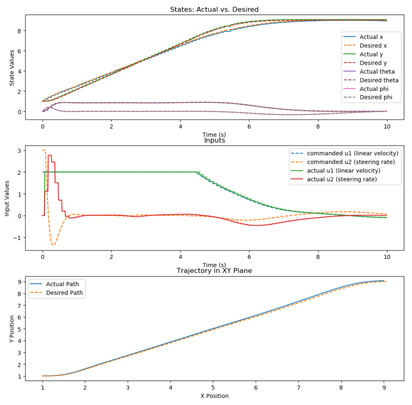
Fig. 18. Optimization Planner Navigation Task Map 1

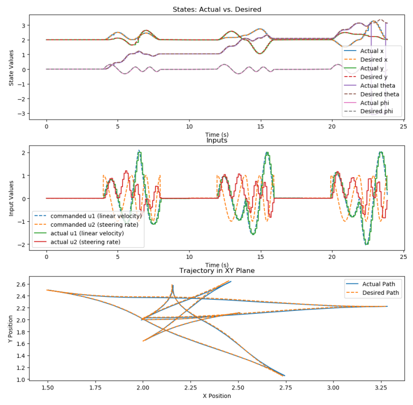
Fig. 16. Sinusoidal Planner Manipulation Task (1, 1, $\pi$, 0) Plot

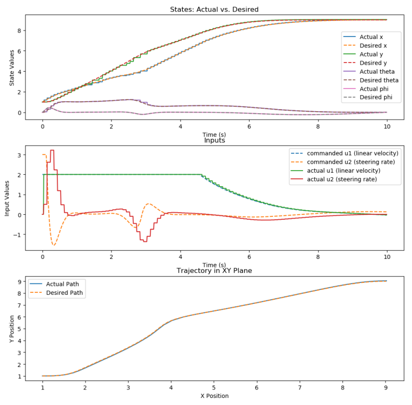
Fig. 19. Optimization Planner Navigation Task Map 2

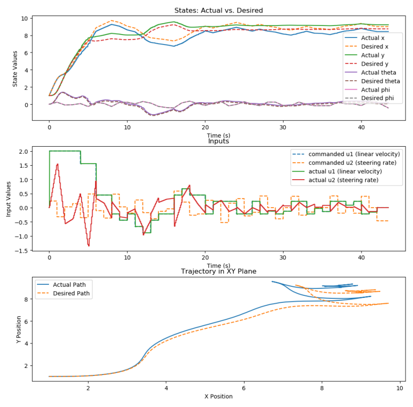

Fig. 20. RRT Planner Navigation Task Map 1

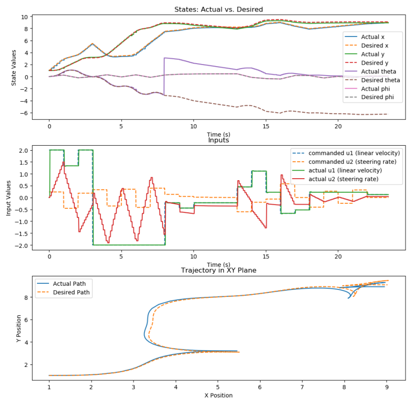

Fig. 21. RRT Planner Navigation Task Map 2