Kinodynamic RRT Motion Planning for Dynamic Robots

Zikai Zhou

December 9, 2025

1 Introduction

Autonomous mobile robots must plan feasible trajectories that satisfy both geometric constraints and dynamic limitations. Classical path planners such as the Rapidly-Exploring Random Tree (RRT) are efficient for high-dimensional geometric planning, yet they ignore the underlying vehicle dynamics, often producing trajectories that cannot be executed by real-world systems. For robots with nonholonomic constraints or dynamic limits—such as ground vehicles, aerial drones, or hovercrafts—the planner must reason about both where the robot can go and how it can move there.

This motivates the use of **Kinodynamic RRT (KD-RRT)**. It is a sampling-based motion planning algorithm that accounts for the robot's dynamic model. In contrast with standard RRTs that use straight-line interpolation between sampled configurations, KD-RRT integrates the robot's equations of motion. Each edge in the search tree represents a physically realizable motion that respects constraints like acceleration, velocity, and steering rate.[1]

In this project, KD-RRT is implemented for a hovercraft robot. A simulated 2-dimensional environment was built using PyBullet. The planner constructs feasible trajectories through state sampliing, dynamic propagation and collision checking. To make the algorithm more efficient and reduce unnecessary exploration, several optimization strategies are added, including goal biasing, greedy goal connection, and adaptive exploration. These techniques helps to find valid paths faster and nevigate complex areas more effctively.

Applications: Kinodynamic RRT* is applied in real-world systems such as self-driving cars for dynamic path planning, aerial drones for obstacle avoidance, and robotic arms for smooth motion generation under physical constraints.

2 Implementation

The KD-RRT planner was implemented in Python using the `PyBullet` simulation environment for collision checking and visualization. The system's six-dimensional state space is defined as:

$$(x,\, y,\, \theta,\, u,\, v,\, \omega)$$

(x, y) are the global coordinates, θ is the heading, (u, v) are the linear velocities in the robot's body frame, and ω is the angular velocity. Each node in the RRT tree corresponds to one of such state, and transitions between nodes are generated using a set of discretized control primitives.

2.1 Dynamic Model[2]

The hovercraft's dynamics are described by the following differential equations:

$$\begin{aligned}
\dot{x} &= u\cos\theta - v\sin\theta, \\
\dot{y} &= u\sin\theta + v\cos\theta, \\
\dot{\theta} &= \omega, \\
\dot{u} &= \frac{F_x}{m}, \quad \dot{v} = \frac{F_y}{m}, \quad \dot{\omega} = \frac{\tau}{I},
\end{aligned}$$

F_x and F_y are the applied body-frame forces, τ is the control torque, m is the mass, and I is the rotational inertia. The dynamics are integrated using a discrete time step $\Delta t = 0.15\,\text{s}$. This ensures smooth evolution of velocity and orientation.

2.2 Sampling and Expansion

The planner samples random states uniformly across the configuration space, with a goal bias probability $\rho_{\text{goal}} = 0.12$ to favor samples near the target. The nearest node q_{near} is selected using a weighted Euclidean metric:

$$d(q_1, q_2) = \sqrt{\sum_i w_i (q_{1i} - q_{2i})^2}, \quad w = [4, 4, 3, 2, 2, 1].$$

From q_{near}, the planner evaluates a set of 18 motion primitives. Each primitive represents a control tuple (F_x, F_y, τ). The primitive that best reduces the distance to the sampled node q_{rand} is selected. Then, the corresponding state q_{new} is computed using the dynamic model. If q_{new} is within the defined state limits and is free of collisions, it is added to the search tree.

2.3 Collision Checking

Collision detection is managed by PyBullet with a cylindrical proxy model that estimates the robot's footprint. A lightweight collision object is placed at each possible state (x, y, z), and PyBullet's `getClosestPoints()` API checks for closeness to obstacles. Only states that clear this test are included in the tree, making sure that all paths avoid any obstacle collisions.

2.4 Greedy Goal Connection

To improve efficiency near the goal, a *greedy goal-connect* strategy is used. When a node is within a proximity threshold of the goal, the planner repeatedly applies the primitive that reduces the distance to the goal until either a collision happens or the goal is reached. This hybrid strategy combines local greedy expansion with random exploration. This approach leads to faster convergence in complex environments and decreases the number of unnecessary branches.

2.5 Simulation Environment and Robot Setup

The simulation was set up in `PyBullet` with a 2D maze-like environment that includes inner walls and narrow openings to test navigation in busy spaces. The robot is a cylindrical hovercraft modeled in URDF. It moves in the x and y plane while rotating around its center. The robot has a mass of $5.0\,\text{kg}$, an inertia of $5.0\,\text{kg}{\cdot}\text{m}^2$, and a collision footprint with a radius of $0.25\,\text{m}$ and a height of $0.15\,\text{m}$. Motion is powered by force and torque primitives (f_x, f_y, τ). These produce kinodynamically feasible paths that show KD-RRT exploration in complex environments.

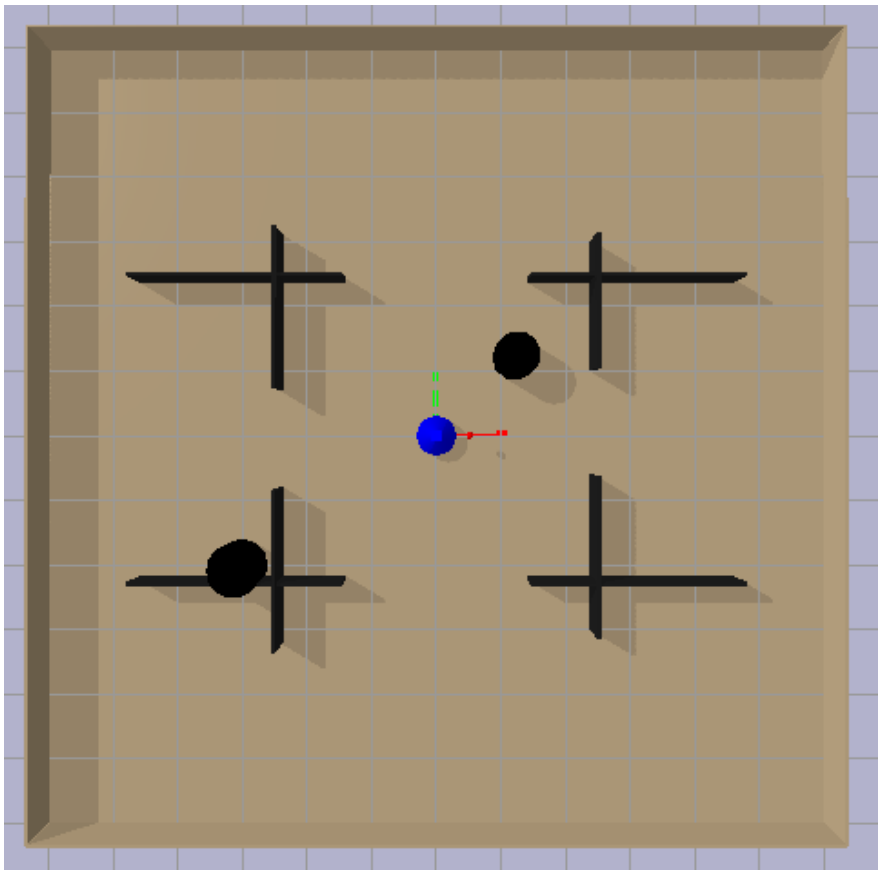

Figure 1: PyBullet simulation environment and hovercraft robot.

2.6 Visualization and Execution

The whole process is shown in real-time with PyBullet's debug line rendering. Each white line shows a tree edge and the final green path displays the successful trajectory. After reaching the goal, the waypoints are executed using the `execute_trajectory` function. This function guides the simulated robot through the computed states.

Overall, this implementation integrates physics-based dynamics, probabilistic exploration, and greedy local optimization to achieve efficient, dynamically feasible motion planning suitable for realistic robotic systems.

2.7 Algorithm and Code

The main structure of the KD-RRT algorithm is shown below. It builds a tree starting from the robot's initial state. The algorithm samples new random states and extends toward them using the robot's dynamics. It also checks for collisions. When a node gets close to the goal, the planner switches to a greedy connection strategy. This strategy tries to reach the goal directly.

Listing 1: Simplified KD-RRT Algorithm

```
def kd_rrt_connect(start_cfg, goal_cfg, collision_fn):
    T = [start_cfg]                    # Tree nodes
    parent = {start_cfg: None}
    primitives = define_motion_primitives()

    for it in range(MAX_ITERS):
        # 1. Randomly sample a state (goal-biased)
        q_rand = sample_goal_biased(goal_cfg, GOAL_BIAS)
        if not is_valid(q_rand, collision_fn):
            continue

        # 2. Find nearest existing node
        q_near = nearest(T, q_rand)

        # 3. Extend tree toward the sample using dynamics
        q_tip = q_near
        for _ in range(CONNECT_HOPS):
            q_new = best_step(q_tip, q_rand, primitives)
            if not is_valid(q_new, collision_fn):
                break
            T.append(q_new)
            parent[q_new] = q_tip
            q_tip = q_new

            # 4. Check if goal reached
            if achieve(q_tip, goal_cfg):
                return backtrace_path(parent, q_tip)

        # 5. Try greedy goal connection when close
        if distance(q_tip, goal_cfg) < 1.2:
            q_tip = try_goal_connect(q_tip, goal_cfg, primitives, collision_fn)
            if achieve(q_tip, goal_cfg):
                return backtrace_path(parent, q_tip)

    print("No valid path found.")
    return []
```

2.8 Discussion

The KD-RRT planner was designed with several choices to balance realism and efficiency. The robot's state includes both position and velocity to capture its nonholonomic dynamics, since a hovercraft cannot change direction or stop instantly. Discrete motion primitives were used instead of continuous control inputs to keep the computation simple and predictable, while still allowing various feasible maneuvers.

A small goal bias was added so the planner occasionally samples near the goal. This helps the tree grow in a purposeful direction instead of exploring randomly. The `best_step` function serves as a local greedy policy, selecting the control primitive that moves the robot closest to the sampled target at each step. This speeds up convergence without losing the randomness needed for broad exploration.

For collision checking, PyBullet's physics engine used a simple cylindrical probe to keep the checks fast and accurate enough for the robot's footprint. Finally, a greedy goal-connection phase was added. Once the tree reaches the goal's neighborhood, the planner can directly try to complete the path instead of continuing random sampling.

3 Results

3.1 Experimental Setup

All experiments took place in a 2D PyBullet simulation environment with inner walls and cylindrical pillars that limited navigation. The hovercraft robot was designed as a simple cylindrical body that could move in a plane. The starting position was set at $(x, y, \theta) = (-4.45,\ 4.45,\ 0)$, while the goal position was at $(4.45,\ -4.45,\ -\pi/2)$.

Three key KD-RRT parameters were changed to see how they affect planning performance: motion primitives, goal bias, and connect hops. The motion primitives define the control actions available during tree expansion. A larger set allows for better maneuverability but requires more computation. The goal bias decides how often the planner samples the goal directly, balancing exploration and goal-directed search. The connect hops determine the number of integration steps toward each sample, which impacts path continuity and runtime. Each parameter was tested on its own to evaluate its influence on execution time, waypoint count, and node expansion.

3.2 Effect of Motion Primitive Count (k) on Planning

All experiments in this subsection use a goal-bias probability of $\rho_{\text{goal}} = 0.12$ and a steering horizon of `CONNECT_HOPS` = 120. The start and goal are fixed, the environment is the same across runs. Timing, waypoint number and node numbers are measured from the start of planning until the first feasible path is found.

- *Execution time (s):* wall-clock time the planner needed to return the first valid path.
- *Waypoints:* number of states on the returned path (post backtrace).
- *Nodes:* number of tree nodes created during the search.

Figure 2 shows how the number of motion primitives (k) impacts planner performance. As k increases from 6 to 18, execution time and node count increase quickly, going from about 32 seconds and 1,883 nodes to over 255 seconds and 15,256 nodes. This indicates that having more control options raises the computational cost. The number of waypoints reaches its highest point at $k = 12$, which implies that a moderate variety of primitives strikes a good balance between path flexibility and efficiency. Larger sets ($k = 18$) create smoother paths but also lead to unnecessary exploration. In contrast, smaller sets ($k = 6$) are quicker but less effective in tight spaces.

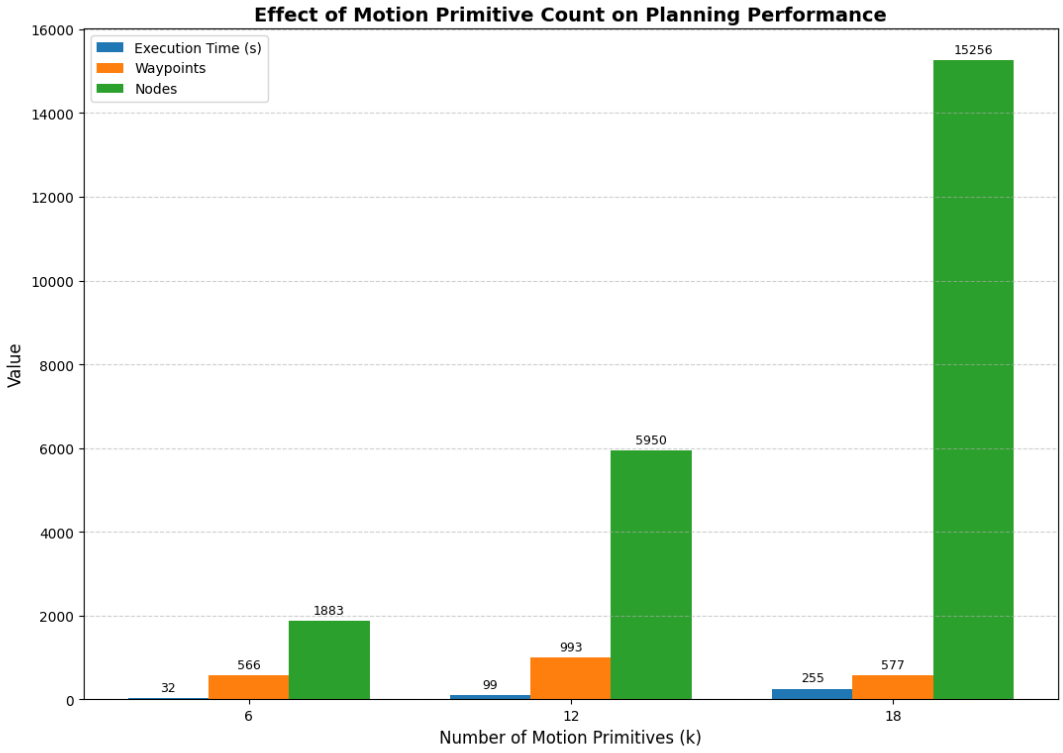

Figure 2: Aggregate performance vs. motion primitive count. Bars show execution time (s), waypoints, and total nodes explored for $k \in \{6, 12, 18\}$. Conditions: goal bias 0.12, `CONNECT_HOPS`= 120.

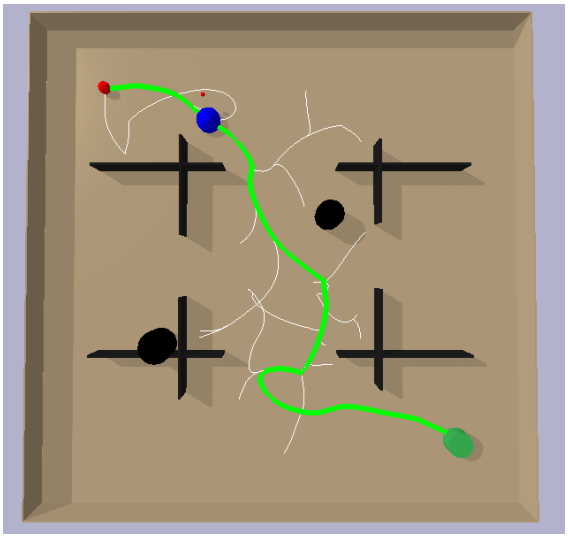

Figure 3: Example solution with $k = 6$ primitives. Fastest overall time in our tests. Path is smooth enough for the hovercraft but the tree explores fewer alternatives.

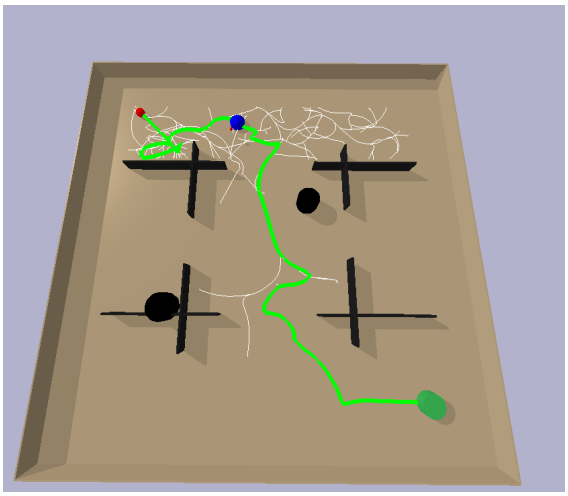

Figure 4: Example solution with $k = 12$ primitives. Richer steering than $k = 6$; planning explores more nodes and takes longer but can yield slightly cleaner routes.

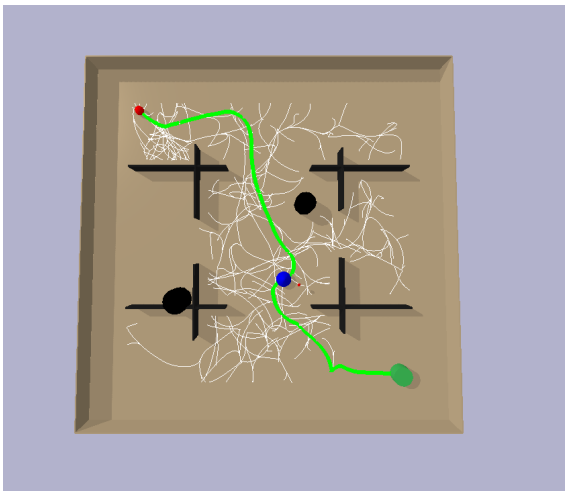

Figure 5: Example solution with $k = 18$ primitives. Most flexible local actions; highest node count and longest planning time due to larger branching and collision checking.

Figure 3 shows the planning outcome using six motion primitives. With limited control actions, the tree expands quickly toward the goal, but it follows coarse and somewhat inefficient turns. The path is found in about 32 seconds, with 566 waypoints and 1,883 nodes. Although the process is fast, the trajectory tends to lack fine directional adjustments, which leads to an unusual curve and a longer path.

In Figure 4, raising the primitive count to twelve gives the hovercraft more maneuvering options. It can adjust its heading better and create a smoother, more flexible path with about 993 waypoints and 5,950 nodes. However, this added flexibility increases the planning time to almost 99 s, showing that it takes longer to evaluate more control candidates.

Finally, Figure 5 shows that the planner uses eighteen motion primitives, which gives it the widest steering capability. The resulting trajectory is smooth and direct; however, it is also very computationally expensive, taking over 255 s and exploring about 15,256 nodes. The improvement in path quality is modest when compared to $k = 12$ and $k = 8$. This suggests diminishing returns beyond moderate primitive diversity.

In summary, increasing the number of motion primitives (k) usually improves local steering options and can shorten the geometric path. However, it also raises the branching factor and collision checks, which leads to longer computation times and more nodes being explored. In our environment, k=6 produced the fastest plans, while k=18 explored many more nodes to achieve similar or slightly shorter paths.

3.3 Effect of Goal Bias on KD-RRT Performance

The goal bias influences how often the planner directly samples the goal during tree expansion. To assess its effect, tests were run with bias values from 0.05 to 0.30 under fixed conditions: $k = 12$ motion primitives and `CONNECT_HOPS`= 120. Figure 6 shows how changes in goal bias affect execution time, waypoints, and total nodes explored.

At a low bias of 0.05, the planner explores nearly randomly. This results in slower convergence at 196.8 s and a higher node expansion of 11,820 nodes. Increasing the bias to 0.12 greatly improves efficiency. The runtime drops to 99.3 s and the node count decreases to 5,950. However, too much bias harms performance again. At 0.2 and 0.3, the planner goes beyond the goal or repeatedly tries impossible connections. This leads to longer runtimes of 427.1 s and 173.8 s.

Overall, a moderate goal bias (0.1–0.15) yields the most efficient search, balancing exploration and directed growth without excessive backtracking.

3.4 Effect of Connect Hops

Figure 7 shows how changing `CONNECT_HOPS` ({60, 80, 100, 120}) affects KD-RRT performance with a fixed goal bias of 0.12. A short horizon of 60 hops results in a very high runtime of 260.7 s and over 15,000 nodes. This indicates inefficient incremental progress and frequent re-sampling. Increasing to 80 hops sharply reduces runtime to 70.6 s and total nodes to 4,462. This suggests a better trade-off between extension length and path correction. Beyond this point, larger hop counts (100 and 120) increase both runtime and waypoint count to 920 and 993, which implies excessive forward integration and unnecessary adjustments. Thus, moderate hop lengths, around 80, provide the most efficient balance between exploration and trajectory refinement. Connect hops represent the step length toward each sample. In this maze, small hops slow progress, while large ones overshoot and collide. A moderate value offers the best balance between control and efficiency.

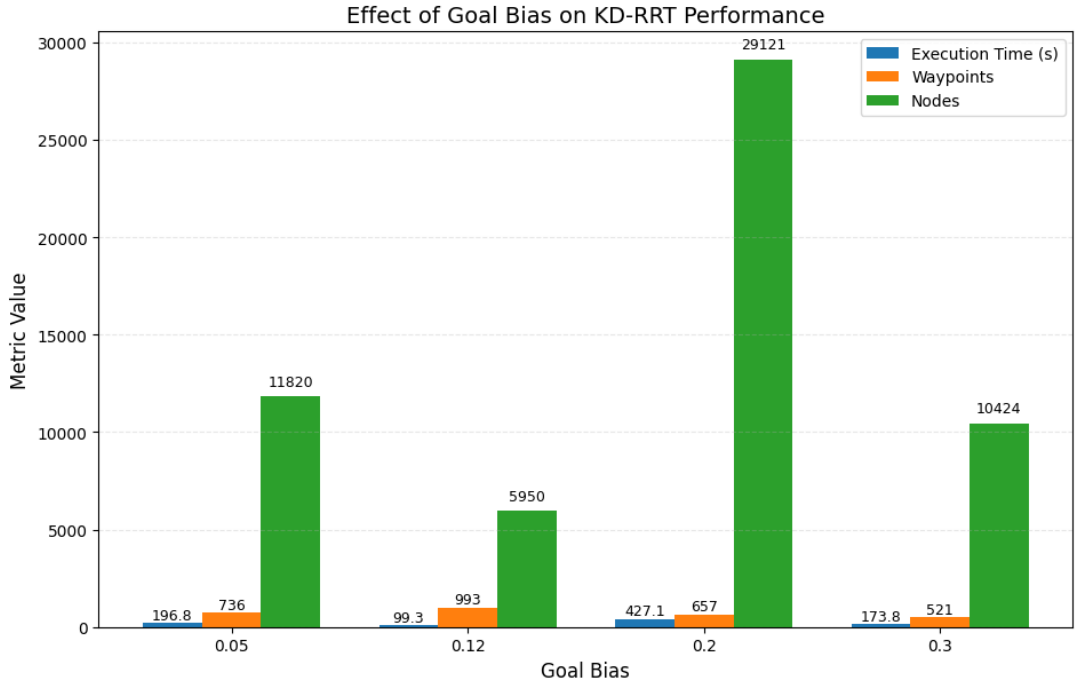

Figure 6: Performance comparison under different goal bias values. Bars show execution time, waypoints, and total nodes explored for biases of 0.05, 0.12, 0.20, and 0.30. Conditions: $k = 12$ motion primitives, `CONNECT_HOPS`$= 120$.

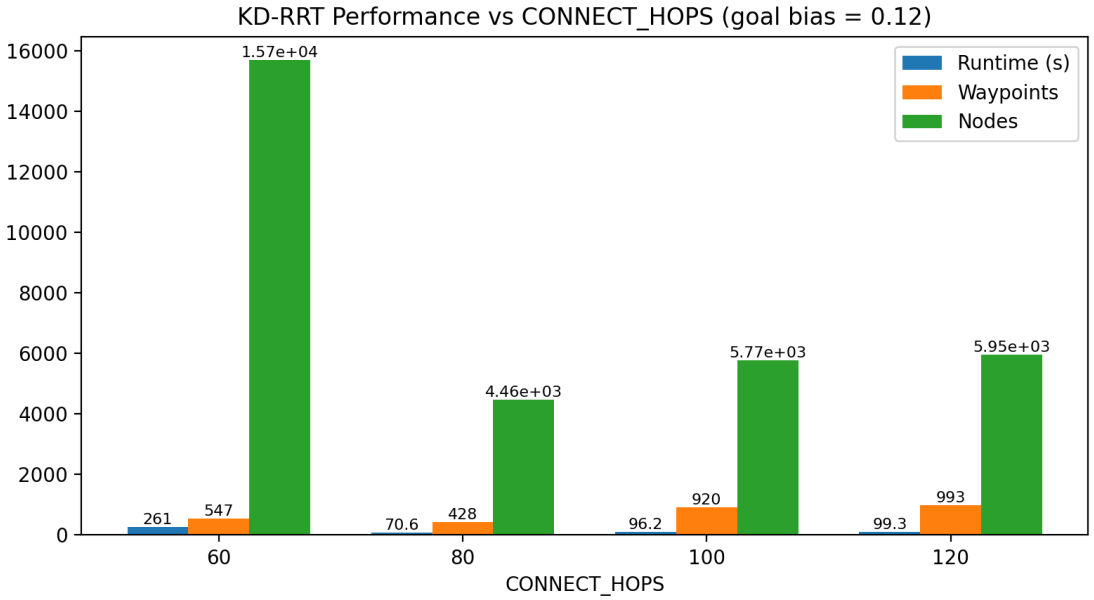

Figure 7: KD-RRT performance vs. `CONNECT_HOPS`. Bars show runtime (s), waypoints, and total nodes explored. Conditions: $k = 12$ motion primitives, goal bias 0.12.

4 Conclusion

This project used a kinodynamic RRT planner for a hovercraft robot and assessed its performance with various motion types, goal biases, and connect hop settings. Results indicate that more complex motion types enhance path quality, but they also raise computation time. Meanwhile, a moderate goal bias and connect hops provide the best balance between speed and stability. Overall, proper adjustment of these parameters is essential for effective path planning in complicated, maze-like environments.

5 Appendix

All the code for this project can be found at: GitHub Repository: Kinodynamic_RRT

References

[1] D. J. Webb and J. van den Berg, *Kinodynamic RRT*: Optimal Motion Planning for Systems with Linear Differential Constraints*, arXiv preprint arXiv:1205.5088, 2012.

[2] MIT OpenCourseWare, *Lecture Notes — "Design of Electromechanical & Robotic Systems", Fall 2009*, pages 29, MIT, 2009. URL: `https://ocw.mit.edu/courses/2-017j-design-of-electromechanical-robotic-systems-fall-2009/5c`