Kinodynamic RRT with PID Control for Obstacle Avoidance in a MuJoCo Environment

Mark Doughten [MD1875]
Vijayendra Sai Chennareddy [VC545]
Shubham Patil [SP2484]

October 26, 2024

1 Introduction

In robotics, autonomous path planning and control are crucial aspects of robot navigation, particularly in environments with obstacles. The objective of this project is to implement a hybrid approach that combines kinodynamic rapidly-exploring random Tree (RRT) for path planning and proportional-integral-derivative (PID) controllers for executing the motion along the planned path. This work is demonstrated using a ball in a MuJoCo simulation environment, where the ball navigates through an environment containing walls and boundaries. The goal is to move the ball from a starting position to a goal position while avoiding obstacles using the kinodynamic RRT algorithm and accurately controlling the movement with PID.

1.1 Objectives

The main challenge is efficiently generating a collision-free path from a start point to a goal point in a cluttered environment and ensuring the ball follows this path with minimal deviation. We undertook the following steps to achieve this objective:

- Configuring MuJoCo and exploring various models, including Newton's cradle with no gravity.
- Developing basic pathfinding in a 3D environment using Dijkstra's algorithm for initial path selection and fixed velocity controls.
- Enhancing pathfinding with kinodynamic RRT, incorporating a safety margin around obstacles.
- Replacing basic controls with a PID controller to follow the planned path.
- Simulating and visualizing the robot's movement within the MuJoCo physics engine.

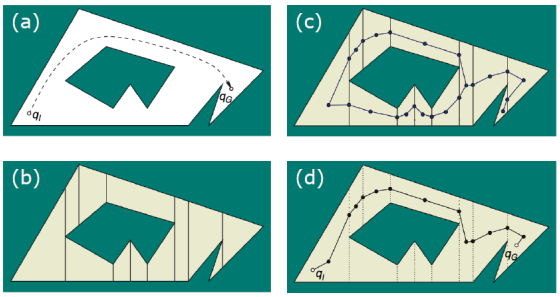


Figure 1: Trapezoidal decomposition ($\mathcal{C} = \mathbb{R}^3_{\max}$)[1]

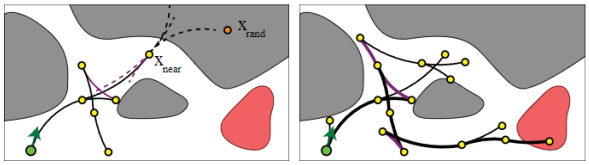


Figure 2: Kinodynamic RRT

2 Environment

The environment consists of a 2D plane with predefined walls and boundaries. The simulation is conducted using the MuJoCo physics engine, which allows for the creation and control of dynamic models. For this project, a ball is moved through the environment containing a set of obstacles to reach the goal. The model used is defined in the `ball_square.xml` file, and the walls are represented as both external boundaries and internal obstacles.

2.1 Map

- **Dimensions:** The 2D environment spans from $x = -0.5$ to $x = 1.5$ and $y = -0.4$ to $y = 0.4$.
- **Obstacles:** A rectangular obstacle is placed in the middle of the environment between coordinates (0.5, -0.15) and (0.6, 0.15), acting as a wall.
- **Goal Area:** The target area for the robot is set between coordinates (0.9, -0.3) and (1.1, 0.3).
- **Boundaries:** The outer walls of the environment prevent the robot from moving outside the area.

This setup ensures the presence of both static and dynamic constraints, making path planning non-trivial and control challenging.

3 Path Planning

3.1 Algorithm Overview

RRT[2] are widely used for motion planning in high-dimensional spaces. In this project, we implement a kinodynamic variant of RRT. Unlike standard RRT, the kinodynamic

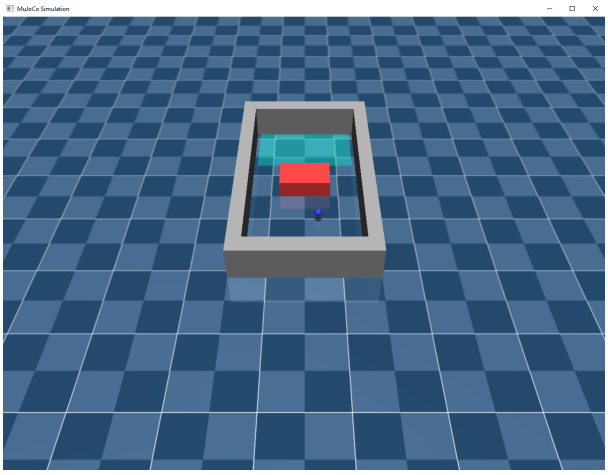


Figure 3: Environment Setup

RRT accounts for the robot's dynamics while generating the tree, ensuring that the planned path can be followed realistically by the robot. This approach helps avoid sharp or unrealistic turns that the robot may not be able to follow.

The algorithm grows a tree from the start position by:

1. Sampling a random point in the environment.
2. Identifying the nearest node in the tree.
3. Applying a control input to move the robot towards the sampled point, simulating its dynamics.
4. Checking the resulting position for collisions.
5. Adding the new position to the tree if no collision occurs.

This process repeats until the robot reaches the goal area or a maximum number of iterations is reached. The function `kinodynamic_rrt` implements this process, while `simulate` handles the forward simulation of the robot's dynamics.

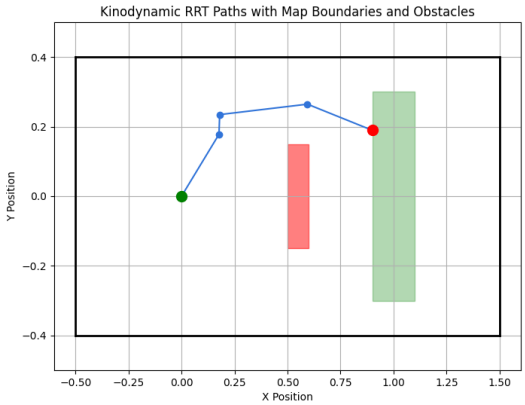


Figure 4: Kinodynamic RRT

3.2 Purpose of `simulate()` in `kinodynamic_rrt`

In traditional RRT, paths are generated based solely on geometric feasibility. However, for real systems with dynamic constraints, `simulate()` serves several critical functions:

- **Ensures Dynamic Feasibility:** Calculates movement towards each sampled point, adjusting with PID control.
- **Incremental Adjustment:** Makes small, corrective adjustments to follow a feasible path.
- **Collision-Free Verification:** Confirms both geometric and dynamic feasibility.
- **Realistic Path Execution:** Validates that each RRT segment can be executed under real-world constraints.

3.3 Collision Avoidance

Collision detection is handled by the function `is_collision_free`, which checks if the robot's new position lies within any of the obstacle regions or the environment boundaries. A safety margin is applied to ensure that the robot does not approach obstacles too closely. The function `is_collision_free_line` checks if a straight line between two points is free of obstacles by verifying that there are no collisions with specified walls.

3.4 Path Construction

Once the algorithm successfully identifies a path to the goal, the tree is traversed from the goal node back to the start node to construct the complete path. This backtracking process is handled by the function `construct_path`. The final path is then visualized, displaying the trajectory, environmental boundaries, and any obstacles encountered along the way.

When a path is established between two nodes, the function `line_goes_through_goal` checks if the path intersects with the goal area by interpolating and sampling points along the path. If an intersection is detected, the endpoint is adjusted to the coordinates (0.9, y) based on the last generated point. This modification effectively minimizes the distance required to reach the goal, as evidenced by the time trial paths.

3.5 Smoothing

After selecting the path within the Rapidly-exploring Random Tree, post-processing occurs using the `smooth_path` function. This function randomly selects points in the graph to determine if they can be directly connected while eliminating intermediate points. As a result, the path is reduced to a series of critical waypoints, simplifying navigation. This reduction allows the control implementation to restart upon reaching each subsequent point, providing the controller with adequate time to complete the feedback loop before moving on to the next waypoint.

4 Navigation

4.1 Controller

PID controllers are commonly used in control systems to achieve stable and accurate control. In this project, two PID controllers are implemented: one for controlling the movement of the ball in the x-direction and the other for the y-direction. Each controller computes the control input based on the error between the current and target positions, aiming to reduce this error over time.

The function `move_ball_to_position_with_pid` implements the control loop, where the ball's current position is compared to the target position from the planned path. The PID controllers compute the necessary force to move the ball toward the target, and the control inputs are applied in each iteration of the simulation.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{d}{dt} e(t) \tag{1}$$

where:

- K_p: Proportional gain,
- K_i: Integral gain,
- K_d: Derivative gain,
- $e(t) = r(t) - y(t)$: Error at time t.

4.2 Parameters

The gains for the PID controllers were manually tuned to achieve smooth movement without overshooting. This tuning required observing the patterns in the model simulation and adjusting accordingly; no penalty for previous errors was applied. The proportional (`kp`), integral (`ki`), and derivative (`kd`) gains were set as follows:

- **Proportional Gain (`kp`):** 0.58 — provides a basic correction based on the current error.
- **Integral Gain (`ki`):** 0 — not used in this implementation to avoid cumulative errors from drift.
- **Derivative Gain (`kd`):** 0.5 — helps to smooth the response by anticipating future errors based on the rate of change.

The higher proportional gain results in more aggressive error correction, which was necessary to achieve trials below a certain time threshold. Once the new point is set, the ball minimizes the error by updating its velocity. The x-direction and y-direction have the same controller configuration for symmetry. The same controllers are used to simulate the motion planning between two nodes in the Rapidly-exploring Random Tree.

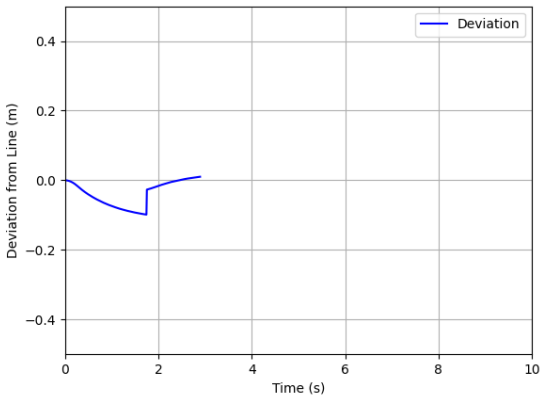


Figure 5: Path Deviation

4.3 Pathfinder

The PID controllers move the ball sequentially through each point in the planned path. Movement is clamped to prevent overshooting, and the simulation continues until the ball reaches the goal position with minimal error (set to 0.075 in this case).

The deviation is calculated using the `calculate_deviation` function. This feedback is important for determining how far the ball is from the trajectory, what adjustments to the controllers are necessary, and ensuring that the ball will not collide with obstacles between two points.

4.4 Adjustments

Several challenges arise during the control process, including handling sharp turns in the path and adjusting the control parameters to prevent oscillations. The derivative gain plays a crucial role in damping the response and ensuring smooth transitions between points. The best method for finding the right parameter was using a process that required increased the K_p value until oscillation was guaranteed. The Ziegler–Nichols tuning method [4] is a heuristic approach to tuning a PID controller, developed by John G. Ziegler and Nathaniel B. Nichols. It is used to determine the proportional, integral, and derivative gains K_p, K_i, and K_d for a PID controller.

Controller Type	K_p	K_i	K_d
P	$0.5K_u$	-	-
PI	$0.45K_u$	$\frac{K_p}{0.83T_u}$	-
PD	$0.8K_u$	-	$K_p \cdot 0.125T_u$
PID	$0.6K_u$	$\frac{K_p}{0.5T_u}$	$K_p \cdot 0.125T_u$

Table 1: Ziegler–Nichols Tuning Parameters

4.5 Path Visualization and Obstacle Mapping

The function `visualize_final_tree` generates a graphical representation of the kinodynamic RRT, illustrating the exploration process and highlighting the path to the goal.

The easiest approach to path visualization was created a top down $2D$ view for visualizing the path on the map. This perceptive aided the team in debugging how the path was getting created, the optimal path, and dangers in specific areas. As an example, the $2d$ view showed that the ball was getting too close to the front of the obstacle. Based on the limited controls and the time constraints, the ball would get too close and not know how to get out of the situation. The new information allowed use to implement a buffer around the object preventing the path from entering that area.

4.6 Dynamic Speed Adjustment for Smooth Deceleration

The function `update_speed` adjusts the robot's speed based on its distance from the goal, allowing for controlled deceleration to avoid overshooting.

5 Results

There were three main objectives in this experiment. The first one was getting the environment running, the second generating a path using kinodynamic RRT, and the third was getting the ball to the goal using controllers. The team demonstrated success on all three objectives.

5.1 Environment

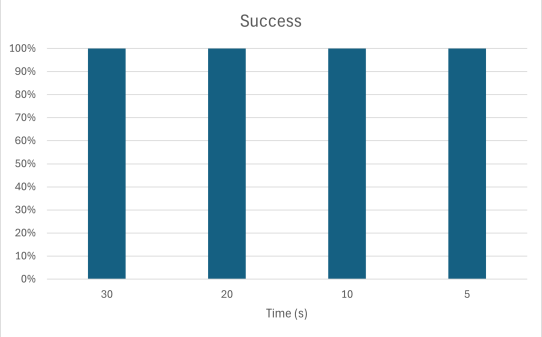
The model creation and execution worked well in the MuJoCo environment and allowed the team to witness the controller behavior prior to using in the kinodynamic RRT search. The simulated search for the path should reflect accuracy how the ball would actually move in the space to the way points and maintain control. It prevented the team was implement controller that behave erratically and would not lead to generating a complete path.

5.2 Visualize Trees

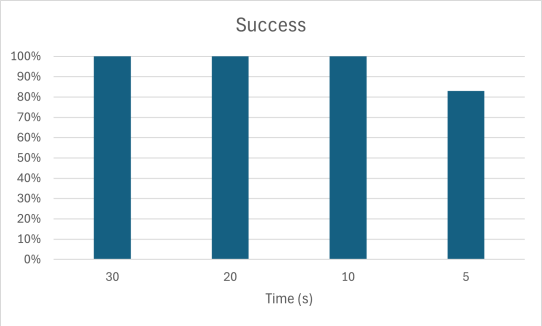
At the end of the report, a complete tree generated using kinodynamic RRT used as a path to the goal. The trees show the effectiveness of using kinodynamic RRT and the success rate for finding the goal despite the different seed. In trails 2, 4, and 5, the tree is not expansive showing the quickness for finding a node in the goal. An area worth exploring in the future is implement a suite of controllers and randomly selecting them for a better representation. In this experiment, the team is using one controller for all the path finding.

5.3 Time Trials

The `run_planning_trial` function runs multiple trials and evaluates the success rate of the kinodynamic RRT in reaching the goal area in the a defined time limit. The team was successful in getting a path each trail and under the time contraints with a 100% accuracy. The quickest shows the effectiveness for using a PID controller for path finding and simulating the ball movements to better represent the actual movement. Similar to the drone movement in the University of Utah technical report [3], kinodynamic RRT is a reasonable approach for solving this problem.



(a) Planning



(b) Execution

Figure 6: Success Rates

6 Conclusion

This project successfully integrated kinodynamic RRT for path planning and PID control for path execution in a simulated 3D environment. By leveraging these techniques, ball can navigate a complex environment with obstacles and boundaries while accurately following the planned path to the goal.

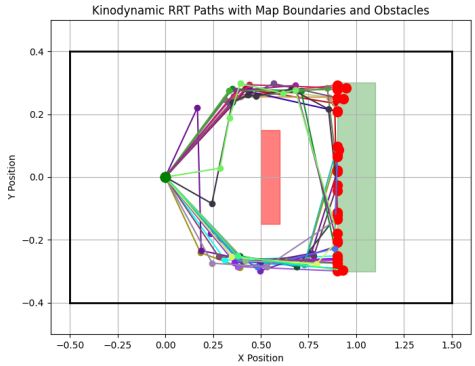
References

[1] Burgard, Wolfram, et al. Introduction to Mobile Robotics. Slides by Kai Arras, last updated July 2011, with material from S. LaValle, J.C. Latombe, H. Choset, and W. Burgard. http://ais.informatik.uni-freiburg.de/news/index_en.php

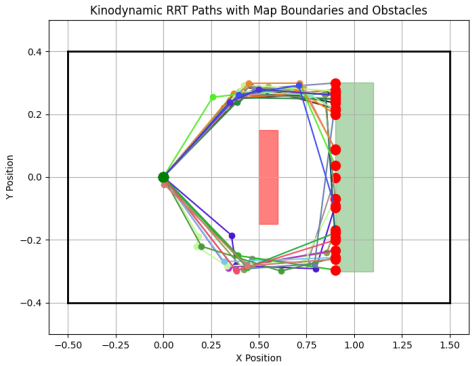
[2] Planning with Dynamics and Uncertainty. https://motion.cs.illinois.edu/RoboticSystems/PlanningWithDynamicsAndUncertainty.html

[3] University of Utah, School of Computing. Technical Report UUCS-12-002, 2012. https://www-old.cs.utah.edu/docs/techreports/2012/pdf/UUCS-12-002.pdf

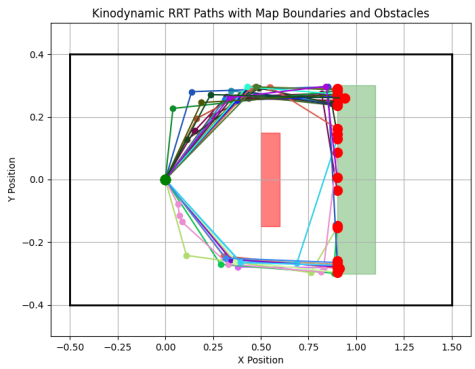
[4] Wikipedia contributors, "Ziegler–Nichols method," *Wikipedia*, `https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method`, accessed 26-October-2024.



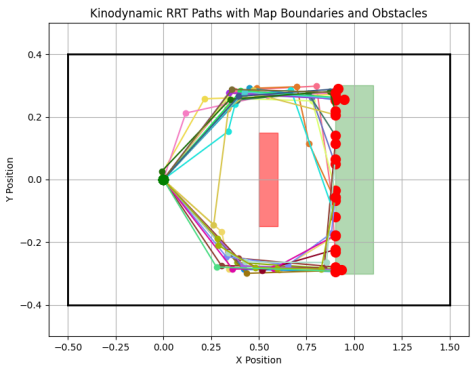
(a) Tmax = 5



(b) Tmax = 10

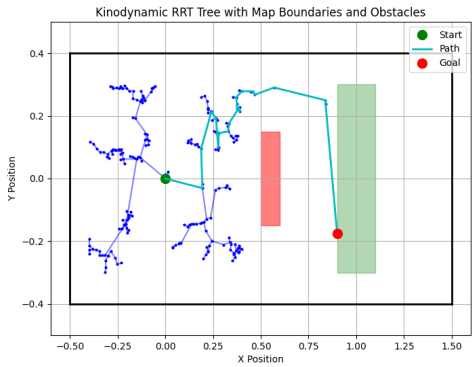


(c) Tmax = 20

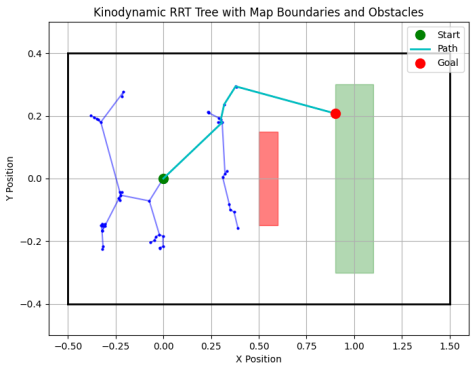


(d) Tmax = 30

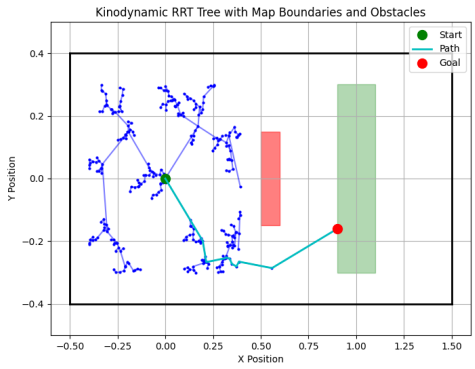
Figure 7: Visualization of path exploration



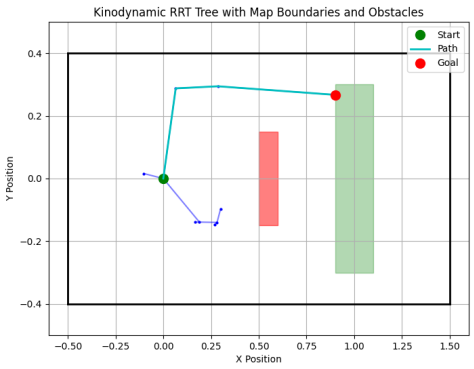
(a) Trial 1



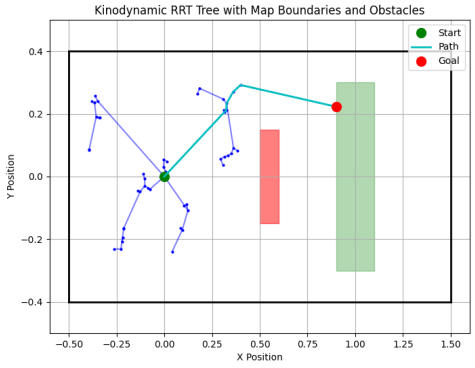
(b) Trial 2



(c) Trial 3



(d) Trial 4



(e) Trial 5

Figure 8: Visualization of trees for different trials.