AUTONOMOUS UAV NAVIGATION USING SAMPLE-BASED AND REINFORCEMENT LEARNING METHODS

LI SHEN [SHENLI1@SEAS], MENGTI SUN [MENGTI@SEAS], XUANBIAO ZHU [ZHUXB@SEAS],

ABSTRACT. The primary objective of this work is to find control inputs that enable robots, in our case unmanned aerial vehicles, to reach a goal state while safely avoiding obstacles. We used sample-based methods including RRT and RRT* in combination with Linear Quadratic Regulator(LQR) to generate minimum snap trajectories and Reinforcement Learning based method Q-learning in combination with a proportional-integral-derivative(PID) controller to acquire valid collision-free trajectories.

1. INTRODUCTION

Unmanned aerial vehicles (UAVs) are widely used as platforms to work in various environments due to their high maneuver capabilities. Motion planning is one of the most fundamental and essential functionalities that enable UAVs to complete tasks while safely avoiding obstacles. We implemented both sample-based methods and reinforcement learning methods in different type of 3D static environments. We compared the performance of these two approaches in terms of successful rate, time efficiency, and free of collisions.

1.1. **Contributions.** In this work, we followed [1, 2] to develop the motor model and develop nonlinear controllers. We implemented RRT in [3], RRT* in [4] as sample-based approaches with LQR Minimum snap method in [1] for trajectory optimization. We used different time estimation for each polynomial segments as we estimated a constant absolute acceleration value, while in the original work the author assumed constant velocity and improved upon this guess with several iterations. Q-learning as RL approach in [2, 5] to find the optimal policy with PID controller for position control to overcome the nonlinear dynamics of quadrotors. We at the end compared the performance in 3D static environments with the same obstacle sparsity. Both achieved good results, able to reach the goal in reasonable amount of time.

2. RELATED WORKS

Traditional motion planning is composed of discrete path planning and continuous trajectory optimization [6]. A collision free path is carried out in the path planning stage. Sample-based method [7] randomly sample valid robot configurations and form a graph of valid motions. Two representative methods are Probabilistic Road-Map (PRM) and Rapidly-Exploring Random Tree (RRT). Asymptotically optimal sampling-based methods like RRT*, PRM* and RRG are guaranteed to converge to global optimal as the samples increase.

At trajectory optimization stage, discrete waypoints are converted into a continuous trajectory which should be dynamically feasible for the quadrotor at every time instant in the trajectory. The goal of trajectory generation is to compute the trajectory that passes within a minimum tolerance of all the waypoints without colliding with any of the obstacles in the environment, while at the same time minimizing the run time cost for the quadrotor to take the path. This is particularly difficult due to the high degrees of freedom and non-linear dynamics of the UAVs. Methods like Minimum-energy [8], Minimum-jerk [9], Minimum-snap [1] have been applied to tackle this issue.

In recent years, reinforcement learning has been a hotspot for planning as it does not rely on the prior structured map and achieves the unification of the global planner and the local planner [10]. Yet it's still an open area for research in terms of its challenges. In [11], an experimental study is done on a UAV using reinforcement learning algorithms of two categories: discrete action space and continuous action space. In discrete action space, the agent decides to follow a policy in the form of greedy learning through choosing the optimum action given state value. This approach has been achieved by algorithms based on deep Q-Network (DQN) [12, 13], Double DQN [14] and Double Dueling DQN (D3QN) [15]. The continuous action space learning expresses actions as a single-value vector and has been achieved by algorithms including Trust Region Policy Optimization (TRPO) [16] and Deep Deterministic Policy Gradient (DDPG) [17]. However, we noticed that a lot of the papers did not provide thorough details on the practical aspects of

implementation and lacked experiment in simulation. We hoped to fill in the gap by providing a framework for applying a RL algorithm to enable UAVs to operate in an unknown environment.

3. Approach

3.1. **Rapidly-Exploring Random Trees(RRT) and RRT*.** Rapidly-Exploring Random Tree, proposed by [3], is one of the most famous sample-based methods, which makes planning by sampling the configuration space(C-space). As a single-query planner, RRT builds a tree from the start configuration to the goal configuration rather than building a Roadmap for the whole C-space [18]. The general idea of RRT is discribed as follows. It first chooses the start configuration as the root of the tree and gets a random sample in the configuration space, then connects that sample as the new state into the tree if there is no collision between the sample and the nearest vertex in the tree, and finally repeats the sampling and connecting process until it finds the sample is the exact goal configuration. Using the parent pointer, it can quickly back-search from goal configuration to start configuration, and then there is a collision-free path. Detailed explanation with pseudo codes of RRT can be found in Appendix A.

Due to the nature of random tree sampling, RRT usually suffers from local minimum and returns a sub-optimal path rather than an optimal path. Therefore, to mitigate this drawback, the RRT*, proposed in [4] adds a cost function to compute the minimum cost-to-go for each node point, rearranging nodes based on the cost. In other words, it recorded the distance each vertex has traveled relative to its parent vertex, and after the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node are examined. If a node with a cheaper cost than the proximal node is found, the cheaper node replaces the proximal node. Another distinction of RRT* is that it constantly rewriting the trajectory of the the tree. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. This feature makes the path more smooth. Pseudo code with explanation is shown in Appendix B.

3.2. **Linear Quadratic Programming(LQR) Minimum Snap.** In [1], the author proposed a method to smooth the transitions between waypoints using piece-wise polynomial functions. They considered each pair of waypoints as endpoints of a trajectory subject to polynomial constraints and solve for the coefficients of N polynomials where N is the number of waypoints - 1, with $d = 8$ terms in each polynomial and 7^{th} order polynomial for the position. This can be easily formulated as an optimization problem in the matrix form shown in (1) where the cost function is the integral of the square of the norm of the jerk as G and h, A and b is the constraints that assuming the quadrotor are moving in the absolute acceleration constant between waypoint segments. As the objective function and constraints are all linear, this optimization problem can be solved using LQR.

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^T H x + q^T x, \\ s.t. \quad & Gx \leq h, \\ & Ax = b, \end{aligned} \tag{1}$$

3.3. **Q-learning.** Given that we have a closed environment with no prior information, it is natural to consider using RL algorithms to generate the optimal path, since they rely only on the data obtained directly from the system. We chose Q-learning because of its computational efficiency and capability of representing low-dimensional data. We assume the environment has Markovian property, where the next state and reward only depend on current state. The learning model can be generalized as a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$, where

- $\mathcal{S}$ is a finite state list, $s_k \in S$ is the state of the agent at step k;
- $\mathcal{A}$ is a finite set of actions, $a_k \in \mathcal{A}$ is the action that the agent takes at step k;
- $\mathcal{T}$ is the transition probability function: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$. It's the probability that the agent takes action a_k to move from state s_k to state s_{k+1};
- $\mathcal{R}$ is the reward function: $\mathcal{S} \times \mathcal{A}$ specifies the immediate reward that the agent gets if transitioning from state s_k to state s_{k+1} by taking action a_k. We have $R(s_k, a_k) = r_{k+1}$.

The objective of the agent is to find the optimal policy that maximizes the total amount of reward. In each state, the Q function $Q(s_k, a_k)$ can be used for the agent to determine how good it is to take an action. The agent iteratively computes the optimal Q-function and records them into **Q-table**. The agent later recalls this knowledge to decide which

action to take in order to optimize its rewards over the learning episodes. In each iteration, the Q function is updated using the Bellman equation

$$Q_{k+1}(s_k, a_k) \leftarrow (1-\alpha)Q_k(s_k, a_k) + \alpha[r_{k+1} + \gamma \max_{a'} Q_k(s_{k+1}, a')]$$

where $0 \leq \alpha \leq 1$ and $0 \leq \gamma \leq 1$ are the learning rate and discount factor. To maintain a balance between exploration and exploitation, we incorporated ϵ-greedy policy as follows:

$$\pi(s) = \begin{cases} \text{a random action } a, & \text{with probability } \epsilon \\ a \in \arg\max_{a'} Q_k(s_k, a'), & \text{otherwise} \end{cases}$$

In order to use Q-learning algorithm, we defined the proper set of states $\mathcal{S}$, actions $\mathcal{A}$ and rewards R. We consider the environment as a finite set of spheres with equal radius d and their centers represent the discrete location. The state of an UAV is their position in the environment, $s_k = [x_c, y_c, z_c]$ where x_c, y_c, z_c are the coordinates of the center of a sphere c at time step k. In each state, the UAV can take an action a_k from a set of four possible actions: North, South, East, West while maintaining the same attitude. The agent gets a reward of -1 for control, -10 if it goes out of bound and 100 if it reaches the goal.

3.4. **PID-Controller.** A PID controller was designed to ensure stable hovering and navigation towards the target. Based on its current state and its learning model, the drone decides the action to the next state. This PID controller controls the motors of the drone to generate thrust force to drive it to the desired position as the following,

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de}{dt}$$

where $u(t)$ is the control input, $e(t)$ is the tracking error between real-time state $s(t)$ and desired state s_{k+1}, K_p is the proportional control gain, K_i the integral control gain and K_d the derivative control gain. In general, the derivative term can decrease the overshoot while the integral term can decrease the steady-state error but may cause larger overshoot. We tuned the control parameters in order to obtain a stable trajectory. More details of Sec. 3.3 and Sec. 3.4 can be found in Appendix C.

4. Experimental Results

4.1. **Comparison between RRT and RRT*.** Since the main difference between RRT and RRT* is that RRT* finds the current optimal path, we try to represent that advantage based on the environment we set. The waypoints generated by RRT and RRT* are described by Figure 1. As we can see, the connection of waypoints generated by RRT is more

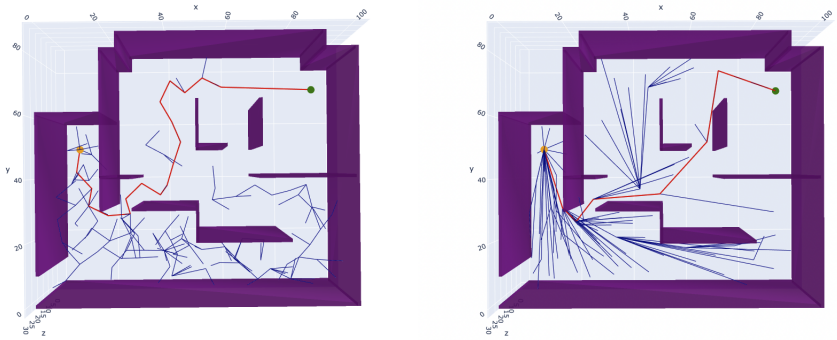

FIGURE 1. The waypoints generated by RRT and RRT*: (i) the left one is generated by RRT. The red trajectory is the final path, while all the blue trajectories are the attempted random tree; (ii) the right one is generated by RRT*. Same as RRT, the red curve is the final trajectory, and blue are the attemptions RRT* took.

squiggling than waypoints generated by RRT*, which represents that the path generated by RRT is not the current optimal. Opposed to that, it is obvious that the path generated by RRT* is more straight and it is optimal since there is

no better waypoint, given the current tree. Thus, the RRT* waypoints are better than RRT when we talk about the path, even though the time spent by RRT star is more than RRT, since RRT* needs to rewire the whole tree based on the cost.

The trajectories generated by RRT and RRT*, with Minimum snap, is shown in Figure 2. As we can see, considering

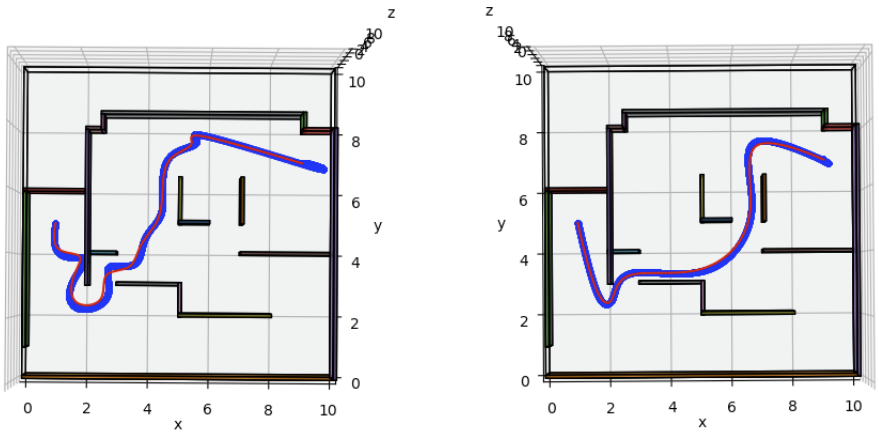

FIGURE 2. The trajectories generated by Minimum snap using the waypoints of RRT and RRT*.The blue curve is the optimized path after Minimum snap, and the red path is the actual trajectory cooperating with the dynamic of the quadrotor model. (i) the left one is using waypoints of RRT; (ii) the right one is using waypoints of RRT.

RRT trajectory, even though the Minimum snap mitigates some of the squiggling curves, the problem still exists. When it comes to RRT star trajectory, it is much more straightforward since the given waypoints are better compared to waypoints generated RRT, and therefore it is more optimal than RRT trajectory, given the same Minimum snap method.

4.2. **Q-learning.** We simulated the Q-learning based navigation is ROS Gazebo with an ARDrone model using `ardrone_autonomy`. The state space is a discretized 5 by 5 grid with a goal position at (5,5). We used learning rate $\alpha = 0.8$, discount factor $\gamma = 0.9$ and ϵ-greedy factor $\epsilon = 0.1$. For the PID controller, we set the proportional gain $K_p = 0.8$, derivative gain $K_d = 0.9$ and integral gain $K_i = 0.1$. It took 34 episodes to train the UAV to find the optimal policy and reach the goal in 8 steps.

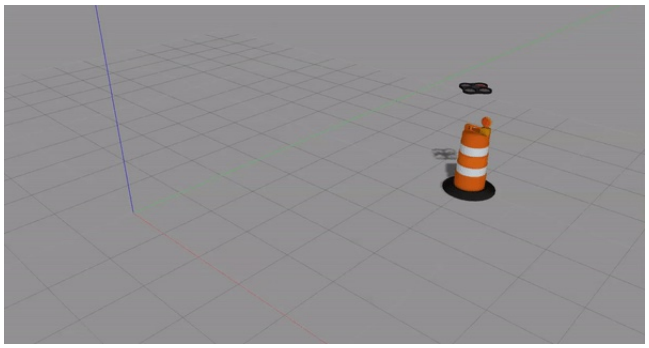

FIGURE 3. Simulation environment in ROS Gazebo.

5. DISCUSSION

5.1. **Results.** We have presented two parallel approaches for autonomous UAV navigation. Our experimental results indicate that both the sample-based and reinforcement learning based approach have the capability to produce reasonable performance regarding speed, smoothness and obstacle avoidance. Our RRT-based algorithm takes about 144.5 seconds and RRT-star-based algorithm takes about 91.33 seconds to generate smooth paths, while the Q-learning algorithm takes about 600 seconds to find the optimal path. This is a reasonable running time for static environments where the planning can be done offline.

5.2. **Future Works.** Currently, the trajectory planner sometimes samples points that are closer to the obstacles. While this might work in simulation, it may cause collision in reality when taking noise into consideration. In addition, it is necessary to investigate the hindrances for apply our method to larger maps, avoiding small-scale decision alternation through of macro-actions. For our Q-learning planner, an important future work is to extend the setting to 3D, namely to incorporate attitude planning and control. This potentially will require using the aforementioned advanced algorithms in 2 since Q-learning will not be suitable for high-dimensional complex states. In addition, we are currently using a simple world in Gazebo and it is valuable to extend the setting to more complicated environments and at the same time keep up with the rapid trajectory planning.

REFERENCES

[1] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 2520–2525.

[2] H. X. Pham, H. M. La, D. Feil-Seifer, and L. V. Nguyen, "Autonomous uav navigation using reinforcement learning," *arXiv preprint arXiv:1801.05086*, 2018.

[3] S. M. LaValle *et al.*, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[4] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[5] H. X. Pham, H. M. La, D. Feil-Seifer, and L. V. Nguyen, "Autonomous UAV navigation using reinforcement learning," *CoRR*, vol. abs/1801.05086, 2018. [Online]. Available: http://arxiv.org/abs/1801.05086

[6] M. Sharir, "Algorithmic motion planning in robotics," *Computer*, vol. 22, no. 3, pp. 9–19, 1989.

[7] L. Quan, L. Han, B. Zhou, S. Shen, and F. Gao, "Survey of uav motion planning," *IET Cyber-systems and Robotics*, vol. 2, no. 1, pp. 14–21, 2020.

[8] B. J. Martin and J. E. Bobrow, "Minimum-effort motions for open-chain manipulators with task-dependent end-effector constraints," *The international journal of robotics research*, vol. 18, no. 2, pp. 213–224, 1999.

[9] K. J. Kyriakopoulos and G. N. Saridis, "Minimum jerk path generation," in *Proceedings. 1988 IEEE international conference on robotics and automation*. IEEE, 1988, pp. 364–369.

[10] Z. He, J. Wang, and C. Song, "A review of mobile robot motion planning methods: from classical motion planning workflows to reinforcement learning-based architectures," *arXiv preprint arXiv:2108.13619*, 2021.

[11] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, "Obstacle avoidance drone by deep reinforcement learning and its racing with human pilot," *Applied Sciences*, vol. 9, no. 24, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/24/5571

[12] ——, "Automatic drone navigation in realistic 3d landscapes using deep reinforcement learning," in *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, 2019, pp. 1072–1077.

[13] C. Liu and E. van Kampen, "Her-pdqn: A reinforcement learning approach for uav navigation with hybrid action spaces and sparse rewards," in *AIAA SCITECH 2022 Forum*, ser. AIAA Science and Technology Forum and Exposition, AIAA SciTech Forum 2022, 2022, aIAA SCITECH 2022 Forum ; Conference date: 03-01-2022 Through 07-01-2022.

[14] L. Zhao, Y. Ma, and J. Zou, *3D Path Planning for UAV with Improved Double Deep Q-Network*, 09 2020, pp. 374–383.

[15] C. Yan, X. Xiaojia, and C. Wang, "Towards real-time path planning through deep reinforcement learning for a uav in dynamic environments," *Journal of Intelligent Robotic Systems*, vol. 98, 05 2020.

[16] V. J. Hodge, R. Hawkins, and R. Alexander, "Deep reinforcement learning for drone navigation using sensor data," *Neural Computing and Applications*, pp. 1–19, 2020.

[17] O. Bouhamed, H. Ghazzai, H. Besbes, and Y. Massoud, "Autonomous UAV navigation: A ddpg-based deep reinforcement learning approach," *CoRR*, vol. abs/2003.10923, 2020. [Online]. Available: https://arxiv.org/abs/2003.10923

[18] M. Elbanhawi and M. Simic, "Sampling-based robot motion planning: A review," *Ieee access*, vol. 2, pp. 56–77, 2014.

APPENDIX A. RRT ALGORITHM

In the Algorithm 1, $SampleFree()$ returns the random sample in configuration space. $Nearest()$ returns the nearest vertex in tree G by comparing the Euclidian distance between x_rand and all vertices in G. $CollisionFree(v, u)$ returns a true if there is no obstacle between v and u. $Steer(x_{nearest}, x_{rand}, step\_size)$ returns the point that a step towards x_{rand} from $x_{nearest}$ with step size $step\_size$.

APPENDIX B. RRT* ALGORITHM

In the Algorithm 2, $\texttt{Line}(x_1, x_2)$ denote a straight line from x_1 to x_2. Given Tree $G = (V, E)$, let $\texttt{Parent}(v)$ be the function that maps a vertex $v \in V$ to a unique vertex $u \in V$, with convention that $\texttt{Parent}(v_0) = v_0$ if v_0 is the root vertex of G. Let $\texttt{Cost}(v)$ be the function that maps a vertex $v \in V$ to the cost of the unique path from the root of the tree to v. Let $\texttt{Near}(G = (V, E), x_{new}, radius)$ be the function that returns the set of all vertices of G that lays in the circle with center x_{new} and radius $radius$. Based on the additive cost function, we have $\texttt{Cost}(v) = \texttt{Cost}(\texttt{Parent}(v)) + c(\texttt{Line}(\texttt{Parent}(v), v))$.

Algorithm 1 RRT

Require: $x_{\text{init}}, x_{\text{goal}} \in C$, max_samples $\in Z_+$, step_size $\in R$
Ensure: $G = (V, E)$
  $V \leftarrow \{x_{\text{init}}\}$
  $E \leftarrow \emptyset$
  for $i = 1, \ldots,$ max_samples **do**
    $x_{\text{rand}} \leftarrow \texttt{SampleFree}()$
    $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$
    $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}}, \text{step\_size})$
    if $\texttt{CollisionFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
      $V \leftarrow V \cup \{x_{\text{new}}\}$
      $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$
      if x_{new} is x_{goal} **then**
        Break
      end if
    end if
  end for=0

Algorithm 2 RRT*

Require: $x_{\text{init}}, x_{\text{goal}} \in C$, max_samples $\in Z_+$, step_size $\in R$, radius $\in R$
Ensure: $G = (V, E)$
  $V \leftarrow \{x_{\text{init}}\}$
  $E \leftarrow \emptyset$
  for $i = 1, \ldots,$ max_samples **do**
    $x_{\text{rand}} \leftarrow \texttt{SampleFree}()$
    $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$
    $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}}, \text{step\_size})$
    if $\texttt{CollisionFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
      $X_{\texttt{near}} \leftarrow \texttt{Near}(G(V, E), x_{\text{new}}, \text{radius})$
      $V \leftarrow V \cup \{x_{\text{new}}\}$
      $x_{\min} \leftarrow x_{\text{nearest}}$
      $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$
      for all x_{near} such that $x_{\text{near}} \in X_{\text{near}}$ **do**
        if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
          $x_{\min} \leftarrow x_{\text{near}}$
          $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
        end if
        $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$
      end for
      for all x_{near} such that $x_{\text{near}} \in X_{\text{near}}$ **do**
        if $\texttt{CollisionFree}(x_{\text{near}}, x_{new}) \wedge \texttt{cost}(x_{near}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
          $x_{parent} \leftarrow Parent(x_{\text{near}})$
          $E \leftarrow (E \setminus \{(x_{parent}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
        end if
      end for
      if x_{new} is x_{goal} **then**
        Break
      end if
    end if
  end for=0

APPENDIX C. Q-LEARNING AND PID CONTROL

Algorithm 3 Q-Learning and PID Control

Require: Learning parameters: γ, α, N
Require: Control parameters: K_p, K_i, K_d
Require: Goal state: G
  Initialize $Q_0(s,a) \leftarrow 0, \forall s_0 \in \mathcal{S}, \forall a_0 \in \mathcal{A}$
  for episode = $1 : N$ **do**
    Measure initial state s_0
    for $k = 0, 1, 2, ...$ **do**
      Choose a_k from $\mathcal{A}$
      Take action a_k that leads to the new state s_{k+1}
      for $t = 0, 1, 2, ...$ **do**
        $u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de}{dt}$
      end for
      until $\|s(t) - s_{k+1}\| \leq d$
      Observe reward r_{k+1}
      Update: $Q_{k+1}(s_k, a_k) \leftarrow (1-\alpha)Q_k(s_k, a_k) + \alpha[r_{k+1} + \gamma \max_{a'} Q_k(s_{k+1}, a')]$
    end for
    Until $s_{k+1} = G$
  end for=0