**Noname manuscript No.**
(will be inserted by the editor)

# Deep Reinforcement Learning for Quadrotor Path Following with Adaptive Velocity

**Bartomeu Rubí · Bernardo Morcego · Ramon Pérez**

Received: date / Accepted: date

**Abstract** This paper proposes a solution for the path following problem of a quadrotor vehicle based on deep reinforcement learning theory. Three different approaches implementing the Deep Deterministic Policy Gradient algorithm are presented. Each approach emerges as an improved version of the preceding one. The first approach uses only instantaneous information of the path for solving the problem. The second approach includes a structure that allows the agent to anticipate to the curves. The third agent is capable to compute the optimal velocity according to the path's shape.

A training framework that combines the tensorflow-python environment with Gazebo-ROS using the RotorS simulator is built. The three agents are tested in RotorS and experimentally with the Asctec Hummingbird quadrotor. Experimental results prove the validity of the agents, which are able to achieve a generalized solution for the path following problem.

**Keywords** Unmanned Aerial Vehicles · Trajectory Control · Path Following · Deep Reinforcement Learning · Deep Deterministic Policy Gradient · Quadrotor

## 1 Introduction

It is well known that unmanned aerial vehicles (UAV) are prepared to undertake a large number of applications in the upcoming future (e.g. transportation, surveil-

This work has been partially funded by the Spanish State Research Agency (AEI) and the European Regional Development Fund (ERDF) through the SCAV project (ref. MINECO DPI2017-88403-R), and by SMART project (ref. EFA 153/16 Interreg Cooperation Program POCTEFA 2014-2020). Bartomeu Rubí is also supported by the Secretaria d'Universitats i Recerca de la Generalitat de Catalunya, the European Social Fund (ESF) and AGAUR under a FI grant (ref. 2017FI_B_00212)

Bartomeu Rubí, Bernardo Morcego and Ramon Pérez
Research Center for Supervision, Safety and Automatic Control (CS2AC), Universitat Politècnica de Catalunya (UPC). Rbla Sant Nebridi 22, Terrassa (Spain).
Tel.: +34-937398973
E-mail: tomeu.rubi@upc.edu, bernardo.morcego@upc.edu, ramon.perez.@upc.edu.

lance, mapping, exploration, search & rescue, maintenance, filming). It is for this reason that the research on these vehicles is constantly growing and keeps developing and implementing the most innovative solutions of control theory, computer vision and artificial intelligence. To accomplish the final applications, the research on UAVs tackles several different problems which derive in diverse research fields, such as the stabilization control, trajectory control, obstacle detection and avoidance, path planning, mission control, fault tolerant control, formation control and many more.

In the last few years the authors of this paper focused their effort on the path following problem, studying and developing the latest techniques to solve this problem. Path following (PF) is a control approach to solve the trajectory control problem that removes the time dependence from the problem resulting in many advantages over the standard trajectory tracking approach [1][2]. In [3] a survey of quadrotor path following algorithms is presented. Several control-oriented and geometric algorithms are reviewed and compared. The most prominent are implemented in a realistic quadrotor model. Conclusions reveal that, in spite of its slightly worse performance in comparison with the control-oriented algorithms, geometrical algorithms are easier to implement, require less state information and result in a lower computational and control effort. Therefore, they become a wise solution for the PF problem. Nevertheless, the main problem of geometric PF algorithms is that tuning their control parameters, which define their performance and stability, depends on factors such as the velocity of the vehicle and the path's shape [4][5]. Thus, those parameters need to be retuned when experimental conditions change.

Machine learning is an interesting research field to address the mentioned problem of geometric algorithms. Its application would aim to achieve an adaptive and tuning-less approach while preserving the control structure and the advantages of the geometrical algorithms. Amongst the diverse machine learning techniques the emerging deep reinforcement learning theory appeared as a promising option to accomplish those objectives. In recent years, a significant progress has been made in the fields of reinforcement learning (RL) and deep learning. Thus, now RL is no longer constrained to discrete and small environments. *Deep Q-Network* (DQN) [6] and *Deep Deterministic Policy Gradient* (DDPG) [7] are two of the most popular deep RL algorithms. In *DQN* the inputs of the agent are images, while *DDPG* is especially designed for continuous state-action spaces. Both algorithms have been used to solve diverse computer science and engineering problems [8][9][10][11][12]. *DDPG* has been also implemented on a quadrotor vehicle to solve the landing problem [13] with successful results. Other quadrotor applications of deep reinforcement learning can also be found in the literature [14][15][16][17][18].

The authors implemented the standard DDPG algorithm to solve the path following problem in a quadrotor simulation environment in [19]. That approach implemented the same structure and concept of the geometrical algorithms. The agent was trained in the Gazebo-RotorS environment in ROS. Simulation results in ideal conditions (without wind and noise) confirmed the potential of the DRL agent. The present paper continues with this research. The contributions of this paper are: (i) the previous approach is improved to deal with noisy sensor measurements and it is trained to perform well when the vehicle is far from the reference path; (ii) a new improved version of the agent, which is able to compute the optimal velocity of the vehicle depending on the path's shape, is presented; (iii) the resulting agents are implemented and validated in the experimental quadrotor platform; (iv) the approach presented in this paper is a straightforward application of the *DDPG* algorithm and the contribution relies on the definition and formulation of the states, the actions, the reward function and the agent structure and parameters.

## 2 Problem Statement

The aim of this paper is to develop a deep reinforcement learning agent capable of solving the path following problem for a quadrotor vehicle. Moreover, this agent must compute the proper velocity of the vehicle which, according to the defined reward, best adapts to the shape of the given path. This agent will be implemented with the *Deep Deterministic Policy Gradient* algorithm. It will be trained in a simulated environment and tested experimentally.

### 2.1 Path Following Problem

Path following is an approach to solve the trajectory control problem. The objective of path following is to make the vehicle follow a predefined path in space with no preassigned time information. That is, contrarily to the trajectory control approach, any time dependence of the problem is removed. A formal definition of the path following problem [20][21][2] is given in Definition 1.

**Definition 1 Path Following Problem:** Let the desired path be described by a curve in the three-dimensional space $\mathbf{p}_d(\gamma) := [x_d(\gamma), y_d(\gamma), z_d(\gamma)]^T$, parametrized by the virtual arc $\gamma \in [0; \gamma_f]$, where $\gamma_f$ is the total virtual arc length. The control objective is to ensure the convergence of the vehicle's position $\mathbf{p}(t)$ to the path $\mathbf{p}_d(\gamma)$ and $\mathbf{p}(t_f) = \mathbf{p}_d(\gamma_f)$ for a finite time $t_f$.

In this paper, the path following problem is solved by implementing a Separated Guidance and Control (SGC) structure (Fig. 1). That is, a structure based on a separation between translational dynamics and rigid-body rotational dynamics. An inner controller, known as the autopilot, is used to track the attitude, altitude and velocity commands. The path following controller is in charge of computing the altitude command ($z_{cmd}$), the *yaw* angle command ($\psi_{cmd}$) and the longitudinal and lateral velocity commands ($v_{cmd}$ and $u_{cmd}$). More information about quadrotor inputs, states and dynamic equations in Sec. 3.1.

### 2.2 Deep Deterministic Policy Gradient

*Deep Deterministic Policy Gradient* [7] is an actor-critic RL algorithm. It is off-policy since the policy that is being improved is different from the policy that is used to generate the action to compute the loss function. And it is model-free because it makes no effort to learn the dynamics of the environment. Instead, it estimates directly the optimal policy and value function.

Fig. 2 shows a common structure of an actor-critic agent, where the policy (actor) is represented independently from the value function (critic). According

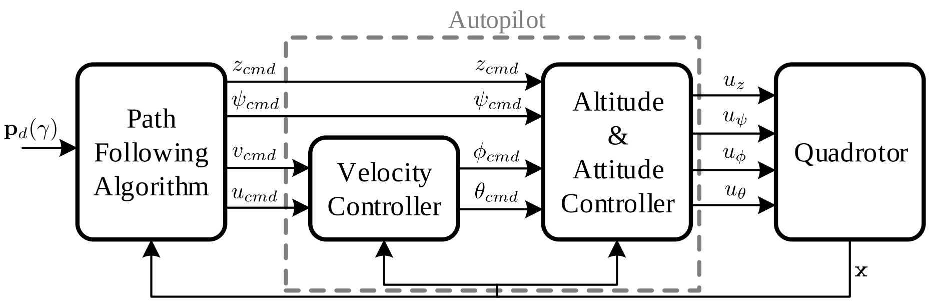

**Fig. 1** Separate Guidance and Control structure.

to the learned policy function ($\mu(s)$), the actor computes the optimal action depending on a state of the environment. The critic estimates the value function ($Q(s, a)$) given the state and the action. The value function gives us information of the expected cumulated future reward for this state-action pair. The critic is also in charge of calculating the temporal-difference error (TD) (i.e. the loss function) that is used on the learning process for both the critic and the actor. In deep reinforcement learning the policy function and the value function, actor and critic, are approximated by neural networks.

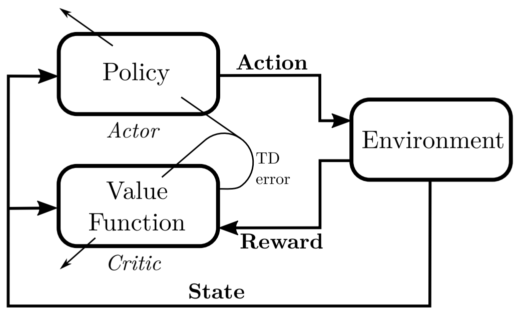

**Fig. 2** Actor-Critic agent structure.

*DDPG* is an improvement of the standard *Deterministic Policy Gradient* [22] algorithm including new concepts of deep learning theory. One of its major advantages is that it is able to provide good performance in large and continuous state-action space environments, which motivated its selection for the particular problem addressed in this paper.

*DDPG* uses two characteristic elements of Deep-Q-Network [6]; the replay buffer and the target networks, which are used to stabilize the learning of the Q-function. A replay buffer is a finite sized memory that stores the transition tuple at each step. This tuple is formed by the current state ($s_i$), the action ($a_i$), the obtained reward ($r_i$), the next state ($s_{i+1}$) and a boolean variable that indicates if the next state is terminal or not ($t_i$). A terminal state is understood as a state where the experiment ends. At each timestep the critic and the actor are trained from a minibatch obtained by sampling random tuples of the replay buffer. This way of training reduces time correlation between learning samples and facilitates convergence in the learning process.

On the other hand, a target network is a network used during the training phase. This network is equivalent to the original network being trained and it provides the target values used to compute the loss function. Once the original network is trained with the set of tuples of the minibatch, the trained network is copied to the target network. Nevertheless, in *DDPG* the target network is modified using a soft update, rather than directly copying the network weights. This means that the target weights are constrained to change slowly. The use of target networks with soft update allows to give consistent targets during the temporal-difference backups and makes the learning process remain stable. Note that *DDPG* requires four neural networks; the actor and the critic and their respective target networks.

When the agent states or actions have different physical units it can be difficult for the neural networks to learn properly and to generalize the solution of the problem. The batch normalization technique [23] is included in the *DDPG* algorithm to avoid this issue. This technique is widely used in deep learning and consists, essentially, on normalizing each dimension of the samples in a minibatch to have zero mean and unit variance.

Eqs. (1 - 2) show the gradient functions used to update the weights of the critic and actor, respectively. $\phi$ are the set of weights of the critic network and $\theta$ the weights of the actor, $\eta_\phi$ and $\eta_\theta$ are the learning rates of the critic and actor, $B$ represents the minibatch of transition tuples and $N$ its size. Target networks are represented with the prime symbol. $y_k$ (Eq. 3) are the target Q-values (Not to be confused with target networks) and are used to compute the loss function. The weights of the critic are updated to minimize this loss function. The discount factor, a value between 0 and 1 that tunes the importance of future rewards to the current state, is represented by $\gamma$. Note that the target Q-Values (Eq. 3) are obtained from the outputs of the actor and critic target networks, following the target network concept.

$$\varDelta\phi = \eta_\phi \nabla_\phi \left( \frac{1}{N} \sum_{i \in B} \Big( Q(s_i, a_i \mid \phi^{Q'}) - y_i \Big)^2 \right) \tag{1}$$

$$\varDelta\theta = \eta_\theta \nabla_\theta \left( \frac{1}{N} \sum_{i \in B} Q(s_i, \mu(s_i \mid \theta^{\mu}) \mid \phi^{Q}) \right) \tag{2}$$

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \phi^{Q'}) \tag{3}$$

Eqs. (4-5) show the update of the weights of the target networks from the trained networks. Parameter $\tau$ indicates how fast this update is carried on. This soft update is made each step after training the main networks.

$$\phi^{Q'} \leftarrow \tau\phi^{Q} + (1-\tau)\phi^{Q'} \tag{4}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \tag{5}$$

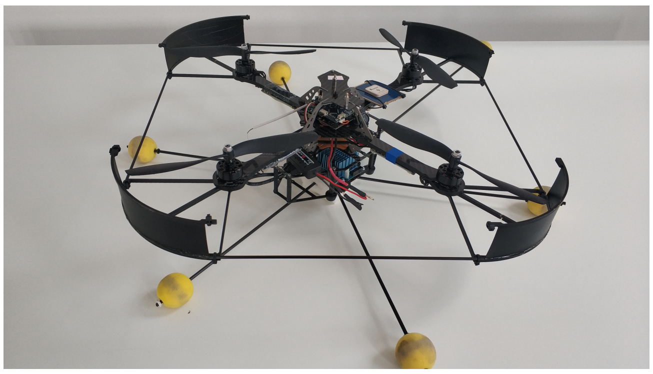

**Fig. 3** Asctec Hummingbird Quadrotor with the Odroid XU4Q on-board PC (center bottom of the UAV).

## 3 Agent Environment

The environment of the agent includes the robot together with the robot's environment [24]. In this paper the robot is the Asctec Hummingbird quadrotor vehicle (Fig. 3). This section gives details of the quadrotor model and the simulation environment wherein the agent is trained.

### 3.1 Quadrotor Model

The quadrotor model has twelve states: the position on each axis in the world frame ($x$, $y$ and $z$), the Euler angles ($\phi$-*roll*, $\theta$-*pitch* and $\psi$-*yaw*), the body velocities ($u$, $v$ and $w$) and the angular velocities ($p$, $q$ and $r$). It has four inputs, that are related to the thrust force ($u_z$) and torques on each axis ($u_\phi$, $u_\theta$ and $u_\psi$). Axis labels and rotational conventions as well as the defined frames of reference of the model are shown in Fig. 4.

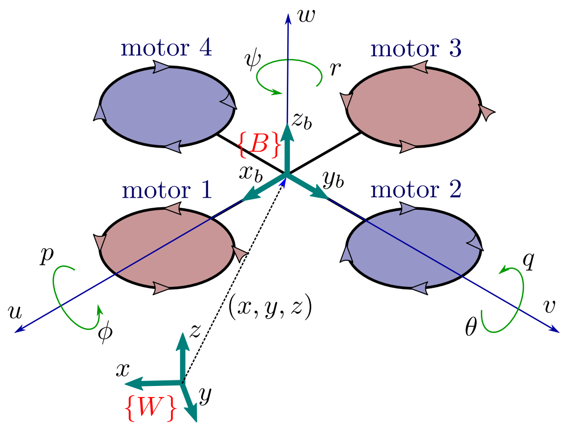

**Fig. 4** Axis labels and conventions.

## 3.2 Training Environment

First training steps of the agent are unpredictable and can become unsafe for the real platform. That is why having a simulated environment is very important in order to maintain the integrity of the experimental platform. Training the agent in a realistic and complete simulated environment will strengthen its effectiveness on real experiments.

In this work a simulation environment was built in the Gazebo-ROS (Robot Operating System) platform, making use of the RotorS simulator [25]. RotorS is a multirotor simulator integrated in Gazebo-ROS which, among the available multirotor models, has a model of the Asctec Hummingbird quadrotor, the vehicle studied in this work. Since certain modifications were made in the real Hummingbird vehicle of our platform, some parameters of the simulation model were updated too (some sensors, pc and other items where placed on the vehicle in such a way that inertias and mass changed). Furthermore, a model of the sensors was included and adjusted to resemble the sensors of the actual quadrotor platform. Nevertheless, simulations assuming ground truth measurements (i.e. ideal sensors) can still be made.

The autopilot was implemented as a package in ROS and it is formed by a set of PID-based controllers. That is, three controllers for the attitude ($\phi$, $\theta$ and $\psi$), one for the altitude ($z$) and two more to control the velocities on the $x$ and $y$ axis ($u$ and $v$). The parameters of these controllers are shown in Table 1. This autopilot was already tested in real experiments with success, and it presents a very similar response on the RotorS simulated environment, thus proving the validity of the model.

**Table 1** Constants of the *PID* controllers of the autopilot.

| | **Kp** | **Ki** | **Kd** |
|---|---|---|---|
| $\boldsymbol{u}$ | 0.32 | - | 0.1 |
| $\boldsymbol{v}$ | 0.32 | - | 0.1 |
| $\boldsymbol{\phi}$ | 4 | 2.2 | 2.4 |
| $\boldsymbol{\theta}$ | 4 | 2.2 | 2.4 |
| $\boldsymbol{\psi}$ | 8 | 1 | 7 |
| $\boldsymbol{z}$ | 4 | 2.2 | 6.6 |

The *DDPG* agent was programmed in python 3.5, using the tensorflow and tflearn libraries to generate and train the neural networks. These libraries permit to save (and restore) the trained nets in order to perform tests or retrain them. Since ROS is only prepared for version 2 of python, the agent was implemented as a regular python script and it communicates with ROS (to subscribe and publish topics) by means of the rospy library.

The scheme of the simulated environment is shown in Fig. 5. The main advantage of building this environment in ROS is that, since the real platform also runs under ROS, the same code of the autopilot and the *DDPG* agent can be transferred to the real quadrotor platform. Thus, the real environment is equivalent to

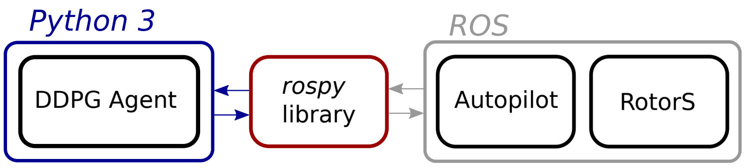

**Fig. 5** Scheme of the training environment.

the one presented in Fig. 5, except that RotorS simulator is substituted by the real vehicle and sensors.

## 4 DDPG for Path Following

This section presents the main characteristics of the DRL agents that are developed in this paper. That is, states, actions and rewards are defined. Other details regarding the structure of the networks or the type of noise added to the actions are introduced as well. Three different approaches are presented. Each approach is an improved version of the preceding one and is implemented using the *Deep Deterministic Policy Gradient* algorithm.

### 4.1 First Approach: Two States

According to the structure of Fig. 1, the path following algorithm must compute four control commands ($z_{cmd}$, $\psi_{cmd}$, $u_{cmd}$ and $v_{cmd}$). Nevertheless, in this first approach the deep reinforcement learning agent is only in charge of computing the reference of the yaw angle ($\psi_{cmd}$). Actually, the action ($a$) produced by this agent is not directly the yaw command but a desired correction ($\psi_{corr,k}$ given in ${}^{rad}/{}_{s}$) over the current yaw angle. Eq. (6) shows how the yaw reference at step $k$ is produced, where $\Delta t$ is the time step. The reason to use the angle correction and not the angle itself as the agent action is to avoid undesired fast angle changes. Moreover, note that the correction is made over the current value of yaw and not over the last yaw reference, which would lead to an incremental control action. Having an incremental control action is equivalent to adding a new integral to the plant, which in this case results in an unstable behaviour. Hence, the selected action achieves a smooth movement while keeping the stability of the system.

$$\psi_{cmd,k} = a_k \, \Delta t + \psi_k \;\;\Big|\;\; a_k = \psi_{corr,k} \tag{6}$$

The other commands defined by the path following controller will depend on the path specifications; $z_{cmd}$ is given by the altitude of the path at the closest point to the vehicle (hereafter named $p_{ct}$, for cross track error point) and $u_{cmd}$ is set to the desired path's velocity. Velocity on $y$ axis ($v_{cmd}$), as in most of the geometrical algorithms, is fixed to 0 ${}^{m}/{}_{s}$.

The basic structure of the *DDPG* algorithm determines that, given a state of the environment, the agent will always choose the best action according to the learned policy. This may not lead to a proper exploration of the action space while training the agent. To enhance the exploration of the agent an Ornstein–Uhlenbeck

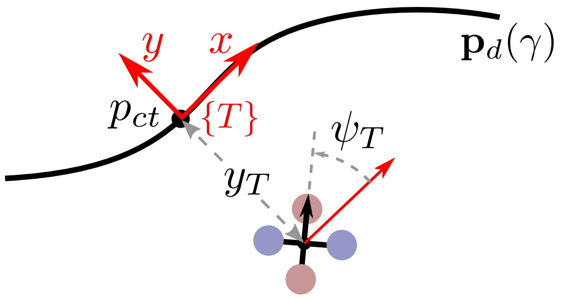

**Fig. 6** States of the agent are with respect of the tangential frame $\{T\}$.

noise (Eq. 7) is added to the action at training time. $n_k$ is the value of the noise at the $k$th iteration, $\theta_n$ is a parameter that defines the speed rate of mean reversion, $\mu_n$ is the drift term which affects the asymptotic mean, $\Delta t$ is the time of a step and $dW_t$ is the standard Wiener process scaled by volatility $\sigma_n$.

Yaw command ($\psi_{cmd}$) including the noise signal is computed as shown in Eq. (8). The exploration rate decreases continuously with the number of training episodes ($j$) in such a way that a smooth transition between exploration and exploitation is achieved while the agent keeps learning. Parameter $\kappa$ indicates the speed of this transition.

$$n_k = n_{k-1} + \theta_n \left(\mu_n - n_{k-1}\right) \Delta t + \sigma_n dW_t \tag{7}$$

$$\psi_{cmd,k} = \left( a_k + \frac{n_k}{{}^{j}\!/\!_{\kappa} + 1} \right) \Delta t + \psi_k \tag{8}$$

In this first approach, the state vector ($s$) is formed by two states (Eq. 9); the distance error ($e_d$) and the angle error ($e_\psi$), both with respect to $p_{ct}$ (Fig. 6). Subscript $T$ is referred to the tangential frame of reference $\{T\}$ that is placed on $p_{ct}$ with $x$ pointing to the path's tangential direction, $z$ pointing up and $y$ pointing to the resultant direction of $x \times z$.

$$\mathbf{s} = \{e_d,\, e_\psi\} \;\mid\; e_d = y_T,\; e_\psi = \psi_T \tag{9}$$

The reward defined for this agent is shown in Eq. (10). This is the reward function that achieves the best performance and fastest convergence among the numerous types of rewards that were evaluated (i.e. continuous or discrete, penalizing bad behaviour or rewarding good path following performance, and mixed strategies). The term $-k_1|e_d|$ penalizes the cross-track error ($e_d$). The term $k_2 v_T$ gives positive reward when the vehicle is moving forward on the path and negative otherwise, where $v_T$ is the velocity of the vehicle projected in the $x$ axis of the tangential frame of reference. $k_1$ and $k_2$ are constants that define the importance of each of the two terms. In this approach these constants take the values of 20 and 10, respectively. Being those the best values amongst several that were evaluated.

$$r = -k_1|e_d| + k_2 v_T \tag{10}$$

The structures of the actor and critic neural networks consist on four layered feed forward networks with 400 neurons in the first hidden layer and 300 neurons in the second one. However, while in the actor's network both the state and the action vectors are connected to the first hidden layer, in the critic networks the action vector is connected directly to the second hidden layer, following the structure of the original algorithm [7]. Making the actions to skip the first layer improves the stability and performance. The neurons of both networks are rectified linear units (ReLU). Batch normalisation technique is used in the two layers of the actor nets, while it is only used in the state input layer in the critic networks.

Table 2 presents the relevant parameters and their values of this first proposed *DDPG* agent.

**Table 2** Parameters of the *DDPG* agent.

| **Symbol** | **Description** | **Value** |
|---|---|---|
| $\eta_\theta$ | Learning rate of actor network. | 0.0001 |
| $\eta_\phi$ | Learning rate of critic network. | 0.001 |
| $\tau$ | Soft target update parameter. | 0.001 |
| $\gamma$ | Discount factor for critic updates. | 0.99 |
| - | Replay buffer size. | 1,000,000 |
| $N$ | Minibatch size. | 64 |
| - | Maximum steps of one episode. | 300 |
| $\Delta t$ | Agent time step. | 0.1 s |
| $\kappa$ | Ratio of exploration-exploitation transition. | 200 |

The agent proposed in this subsection can solve the path following problem properly (see Sec. 6). In fact, it is the best agent setup in terms of PF performance that we were able to obtain among numerous and diverse agent setups that were tested with only two states. However, notice that these two states of the agent (Eq. 9) only provide instantaneous information about the path. In other words, states are computed only from the point $p_{ct}$ in the path and they provide no information about the path shape to come. Therefore, it is not possible for the agent to anticipate the curves of the path. Next subsection presents an improvement over this approach that handles this issue.

## 4.2 Second Approach: Anticipation State

To deal with the anticipation, the issue mentioned in the previous subsection, a new form of the state vector is proposed. The rest of the parameters and structure of the agent of the first approach are maintained. In addition to the two states defined in Eq. (9), another state is included ($\psi_{T_2}$). This state is an angle error between the vehicle's yaw angle and the path's tangential angle. However, in this case the angle error is not computed from the point $p_{ct}$ but in a point that is forward on the path, as represented in Fig. 6. This new state gives information about future orientation of the path with respect to the vehicle and makes it possible for the agent to anticipate curves to come, improving substantially its path following performance

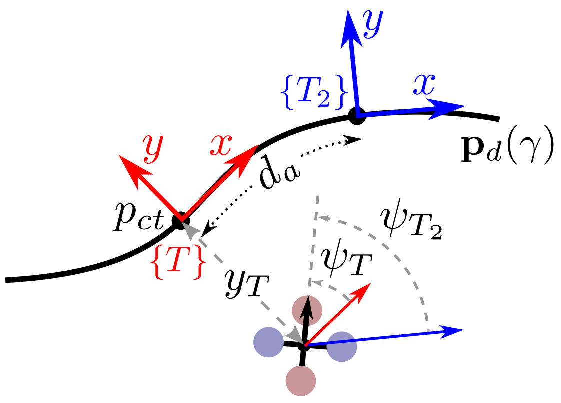

**Fig. 7** States of the second approach; angle error with respect forward tangential frame $\{T_2\}$.

(see Sec. 6). The state vector of this approach is presented in Eq. (11), where $T_2$ subscript indicates that the state is computed from the tangential frame on a point that is forward on the path.

$$\mathbf{s} = \{y_T,\, \psi_T,\, \psi_{T_2}\} \tag{11}$$

The distance at which the second tangential frame, $\{T_2\}$, is placed on the path is named anticipation distance and it is represented by $d_a$. To obtain the best possible performance of the agent, it is necessary to choose a proper anticipation distance. From different tests, it was proven that $d_a$ depends on the velocity of the vehicle on the path. That is, with higher velocities it is necessary to have a larger anticipation distance. For instance, the optimal anticipation distance (according to the obtained PF performance) at a velocity of 1 $^m/_s$ is $0.6m$.

## 4.3 Third Approach: Adaptive Velocity

The main drawback of the previous approaches is that, with the defined structure, the agent can only learn to solve the problem at one specific velocity. That is, if during the training process the velocity of the vehicle is changed every episode, convergence cannot be achieved. In other words, the policy depends on the vehicle's velocity. This subsection presents an improvement that permits the agent to work at different velocities and also makes it capable of computing at each step the velocity of the vehicle that best adapts to the shape of the path according to the defined reward.

In order to have an agent that is resilient to different velocities and path's shapes, the first step is to include the velocity of the vehicle ($\|\mathbf{v}\|$) as a state of the agent. Nevertheless, this modification is not sufficient to accomplish our goal. In *DDPG* it is necessary to define a state vector that fulfils the deterministic property. This means that, knowing the current state vector and action the next state can be estimated. Therefore, since the velocity of the vehicle is an exogenous variable of the system (defined by the user) it is not possible to predict its value, and thus, it is not a deterministic state. To make it deterministic, the action vector must act on the velocity state.

In this third approach, in addition to the yaw correction action defined in Eq. (6), a new action that computes a velocity correction ($u_{corr,k}$) over the current velocity of the vehicle is included. Eq. (12) shows how the velocity command on the $x$ axis is produced from this action (including exploration noise of Eq. 7, only used during the training phase). Again, with the aim of avoiding fast changes on the velocity and to assure the stability of the system, a correction action has been used rather than a velocity action or an incremental action of the command.

$$u_{cmd,k} = \left( u_{corr,k} + \frac{n_k}{j/\kappa + 1} \right) \Delta t + u_k \tag{12}$$

Introducing this new state ($\|\mathbf{v}\|$) and new action ($u_{corr,k}$) to the agent may seem to be enough to solve the problem. However, as mentioned in Sec. 4.2, notice that the path's position where the future angle error state ($\psi_{T_2}$) is computed depends on the velocity of the vehicle. Therefore, having only this state computed with a fixed anticipation distance ($d_a$) does not provide enough information to solve the path following problem at different velocities. Rather than that, two solutions were considered: Adding more future angle error states at different anticipation distances or modifying at each step the anticipation distance at which the angle error is computed in function of the vehicle's velocity.

Including several future angle states at different distances resulted disadvantageous for two reasons: first, having more states makes the training process much slower; second, since at a given velocity only the information of 1 or 2 future angle states is exploited, the remaining states become irrelevant. Having many states that do not provide significant information to solve the problem leads the agent to lose effectiveness. For this reason, in this approach the mentioned issue is solved by having only one future angle state ($\psi_{T_2}$), which is computed with an anticipation distance adapted according to the vehicle's velocity.

Several tests at different velocities were performed in order to find the relation between the velocity of the vehicle and the optimal anticipation distance ($d_{a,opt}$). Optimal in the sense of being the distance that provides more information, and thus, results in a higher performance of the agent. The results obtained from these tests were approximated by the linear piecewise function shown in Eq. (13). This function computes the optimal anticipation distance as a function of the current velocity of the vehicle.

$$d_{a,opt} = \begin{cases} \|\mathbf{v}\| - 0.3 \qquad if \ \ \|\mathbf{v}\| \geq 1 \\ \\ 0.6\|\mathbf{v}\| + 0.1 \ else \end{cases} \tag{13}$$

Summarizing, in this approach the velocity of the vehicle is added as part of the state vector and the future angle state is computed with an adaptive anticipation distance ($d_{a,opt}$). A velocity correction is included in the action vector. Eqs. 14 and 15 present the state and action vectors, respectively. The agent computes the velocity command on the $x$ axis in such a way that it adapts to the path's shape. Velocity on the $y$ axis is still fixed to 0. The reward function of the first approach (Eq. 10) is also maintained. Weights of the reward ($k_1$ and $k_2$) acquire a significant role in this approach, since they define the priority of the trade-off between having small path distance error or travelling at high velocities. All parameters of Table 2 are preserved except for the ratio of the exploration-exploitation transition ($\kappa$), which is set to 1000. This is because the training process of this approach is slower.

$$\mathbf{s} = \{y_T,\ \psi_T,\ \psi_{T_2}(d_{a,opt}),\ \|\mathbf{v}\|\} \tag{14}$$

$$\mathbf{a} = \{\psi_{corr,k},\ u_{corr,k}\} \tag{15}$$

The ingredients that make the agent capable of following a trajectory in space with adaptive velocity have been defined. Nevertheless, it is of utmost importance to design a rich training environment that allows the agent to converge to an efficient and robust solution. Details of this training process are given in Sec. 5.

## 5 Training Process

The training process of the agents has been performed in the training environment detailed in Sec. 3.2. This training environment is integrated in a linux Xubuntu virtual machine with a dedication of 8GB RAM and four 1.80GHz processors (i7-8550U CPU). The training process is performed in real time.

### 5.1 Training of $1^{st}$ and $2^{nd}$ approaches

The first and second approaches followed the same structure in the training phase. That is, the vehicle is required to follow a half lemniscate (8-shaped) path at a constant velocity of 1 $^{m}/_{s}$ in the $x$ body axis ($u_{cmd}$). This path is defined in Eq. (16), where $A$ is the radius of one of the circumferences of the path, fixed to $4m$, and $\gamma_p$ is the virtual arc, which ranges from 0 to $2\pi$ rad. The path is discretized with a precision of 0.01m between each path point.

$$\begin{aligned} x_d\left(\gamma_p\right) &= 2A\cos\left(\gamma_p\right) \\ y_d\left(\gamma_p\right) &= A\sin\left(2\gamma_p\right) \end{aligned} \tag{16}$$

Both agents were trained following the specified path in ideal conditions, meaning that the system uses ground truth measurements and the vehicle starts each episode at the initial position of the path with the *yaw* angle oriented tangentially to it. As denoted in Table 2, each episode has 300 steps of 0.1 seconds. The learning evolution of the first and the second approaches are shown in Figs. 8 and 9, respectively. These figures show. for each episode, the average path distance error ($\overline{|d|}$) and the accumulated reward ($\sum r$) in all the steps of the episode. As the agents keep training the average error decreases and the accumulated reward grows until training converges.

It is important to mention that, as the training process is stochastic, even if the same parameters and structure are maintained, the performance of the trained agents can vary. The agents presented in this paper are the ones that achieve the best performance, in terms of path distance error, among a set of different trained agents that were obtained. In this particular case, the $1^{st}$ approach converged around the $120^{th}$ episode while the $2^{nd}$ approach did it approximately at episode 90.

The resulting agents were tested in the RotorS simulation platform (see Sec. 6.1). They proved to perform well with ground truth measurements. However, if a model

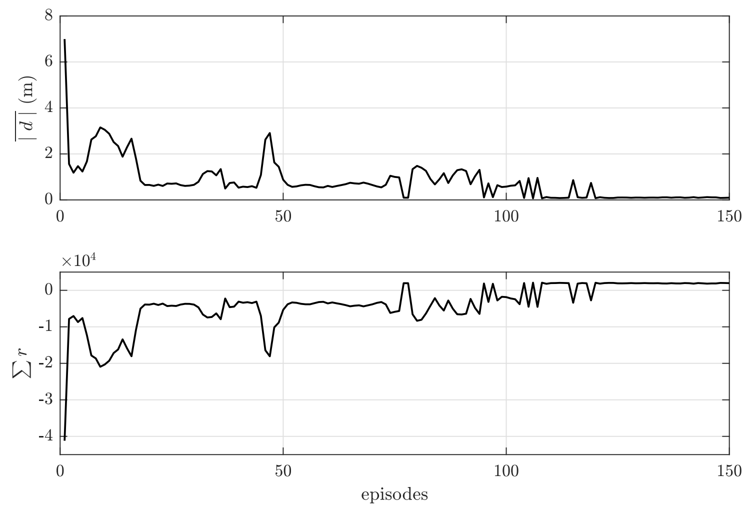

**Fig. 8** Average distance error and accumulated reward on each episode during training phase of $1^{st}$ approach agent (2 states).

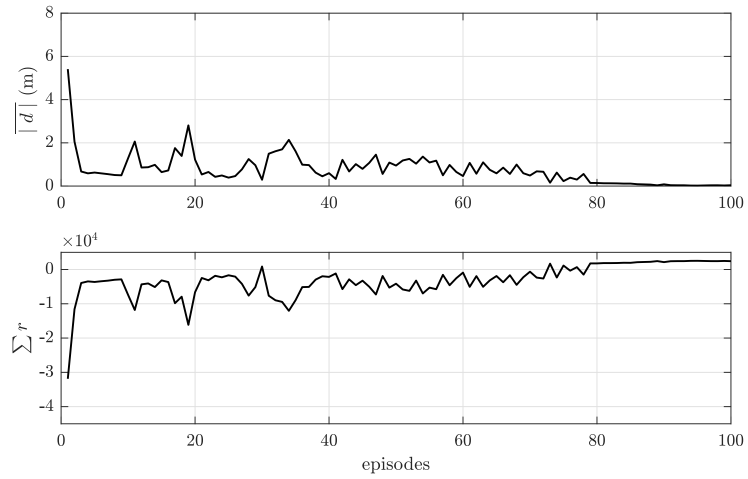

**Fig. 9** Average distance error and accumulated reward on each episode during training phase of $2^{nd}$ approach agent (3 states).

of the sensors is added, the agents present some difficulties to follow the path properly. Particularly, when the vehicle moves far from the path (due to drift or jumps on sensor measurements) and needs to converge back, the vehicle can start loitering around the path without being able to converge to it.

The solution to the mentioned problem could be to train the agent with the model with sensors. However, to capture the dynamics of the system with noisy measurements becomes challenging for the agent and, sometimes, training does not converge in these conditions. Alternatively, this issue is tackled by retraining the agents to learn the policy when the vehicle is far from the path. To do so, the agents are first trained as explained before, and then, they are retrained following the same path but starting at random positions and orientations different from the initial point of the path. In this way, the agents learn how to behave out of the path. Thus, if the vehicle occasionally moves out of the path because of the noisy sensor measurements, the agent will be able of driving the vehicle back to the path.

Both agents (1$^{st}$ and 2$^{nd}$ approaches) were trained 100 more episodes following the specified lemniscate path (Eq. 16) with random initial conditions. That is, in each episode of this training phase the starting position of the vehicle is set at a distance of $-2m$ to $2m$ from the initial position of the path, and the initial orientation is incremented an angle between $-\pi/2$ to $\pi/2$ from the initial path tangential angle. The initial position and angle are selected randomly with a uniform probability distribution in the defined intervals.

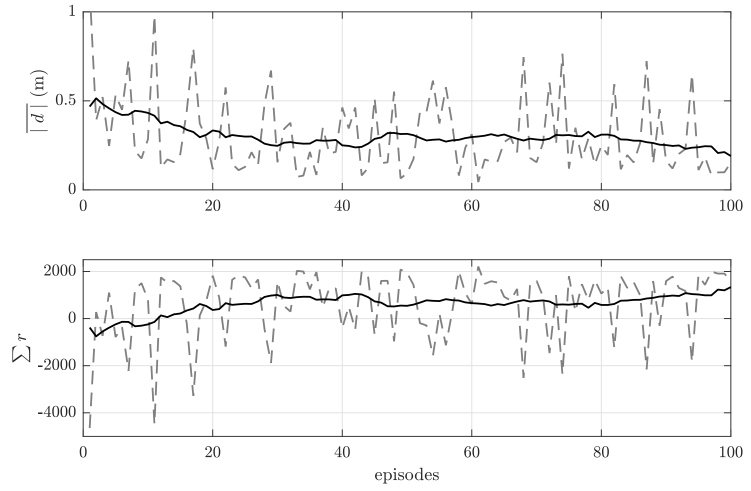

**Fig. 10** Average distance error and accumulated reward on each episode during training phase with non-ideal initial conditions of 2$^{nd}$ approach agent; gray dashed lines are real values and black lines are a 20-episodes moving average.

Fig. 10 shows the learning evolution of the 2$^{nd}$ approach with the 100 new training episodes. Since the initial position and orientation change randomly in each episode, the obtained average distance error and accumulated reward also vary arbitrarily. For this reason, to show better the progression of this learning phase, a 20-values moving average is presented in both plots. That is, episode values are represented with gray dashed lines, while the moving average is represented with solid black lines in Fig. 10. The learning results show how this training phase permits the agent to learn to perform better in diverse initial conditions. This acquired knowledge will notably improve the performance in real experiments, as revealed in Sec. 6.

## 5.2 Training of 3rd approach

The $3^{rd}$ *DDPG* approach developed in this paper requires training in a richer environment than the previous versions. That is because the agent needs to train with different curves in order to learn the optimal vehicle's velocity and the *yaw* angle's policy according to the path radius.

In the training process of this agent the vehicle will be required to follow an asymmetrical half lemniscate path. This is an 8-shaped path where each circle has a different radius. This path is defined in Eq. (17), where $A_1$ and $A_2$ are the radius of each circumference of the path, respectively. The value of this radius is changed every episode, taking a random value between $0.5m$ and $10m$ with a uniform probability distribution. Again, the virtual arc parameter $(\gamma_p)$ ranges from 0 to $\pi/2$ rads, and the path is discretized with a precision of $0.01m$.

$$
\begin{aligned}
x_d(\gamma_p) &= \begin{cases} 2A_1 \cos(\gamma_p) \; if \;\; 0 \le \gamma_p \le \pi/4 \\ 2A_2 \cos(\gamma_p) \; if \; \pi/4 < \gamma_p \le \pi/2 \end{cases} \\
y_d(\gamma_p) &= \begin{cases} A_1 \sin(2\gamma_p) \; if \;\; 0 \le \gamma_p \le \pi/4 \\ A_2 \sin(2\gamma_p) \; if \; \pi/4 < \gamma_p \le \pi/2 \end{cases}
\end{aligned}
\tag{17}
$$

The first training attempts of the agent with the stated environment resulted to be quite unfruitful. Concretely, after hundreds of episodes, the agent just learned that the best way of maximizing the reward (reward function in Sec. 4) was to keep the vehicle static. The reason for this strange behaviour can be explained as follows: since turning around arbitrarily is not penalized when, due to the lack of exploration the policy is not defined yet, whenever the agent starts moving the vehicle forward, as it is rotating, it ends up moving in the opposite direction of the path, receiving a penalty for that policy; therefore the best action is to keep $u_{cmd} = 0$.

A simple but effective solution for such issue is proposed in this paper. It consists on forcing the vehicle to move constantly by establishing a minimum velocity of 0.1 $m/s$. Even if this condition initially produces negative rewards, it ends up promoting the agent to learn the policy of the *yaw* action. At the same time, as soon as the velocity vector of the vehicle starts to be parallel to the path, the agent can start learning that higher velocities lead to greater rewards. Hence, a successful learning process is achieved.

The training results of this agent are shown in Fig. 11. This figure shows the average distance error $(\overline{|d|})$, the average velocity on the $x$ axis $(\overline{u})$ and the accumulated reward $(\sum r)$ on each episode. A 50-episodes moving average is applied to episode values to help the interpretation of each of the three plots. Again, gray dashed lines represent episode values while black lines show the moving average.

It may seem that training converged around episode 400. However, even the average error or reward appear to be constant, evaluating the trained agents with simulation tests showed that they kept learning and improving their performance until around episode 1000. The reason for that is because training more episodes permits to learn the policy on unusual states.

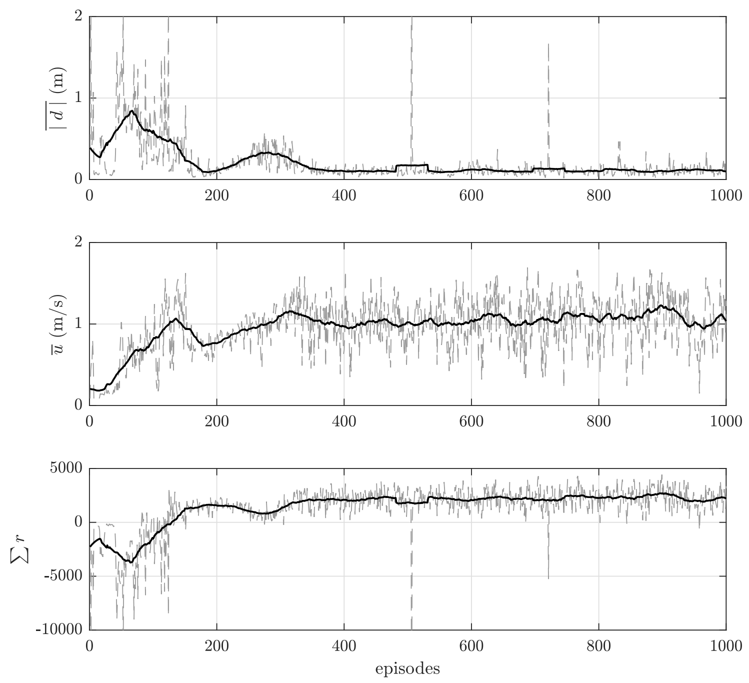

**Fig. 11** Average distance error, average velocity and accumulated reward on each episode during training phase of $3^{rd}$ approach agent; gray dashed lines are real values and black lines are a 50-episodes moving average.

Such long and complex training process allows the agent to learn the policy out of the path. Thus, unlike the $1^{st}$ and $2^{nd}$ approaches, this approach does not need any additional training with diverse initial conditions to improve its performance on experimental results.

## 6 Results

This section presents the results obtained with the three trained agents while following a path in different conditions. The agents were tested in simulation and experimentally with the Asctec Hummingbird platform.

### 6.1 Simulation

The simulations presented in this paper were performed in the same framework where the agents were trained. That is, the RotorS simulator integrated in the ROS-Gazebo platform.

First, the three approaches were tested following a lemniscate path (Eq. 16), the same path used in the training phase. Again, the radius of the path is $A = 4m$. However, this time the vehicle was required to follow a full lemniscate, with the virtual arc parameter, $\gamma_p$, ranging from 0 to $4\pi$ rads. The vehicle started at the initial point on the path with the *yaw* angle oriented tangentially to it.

Table 3 shows the results obtained while following this path with ground truth measurements. That is, in the same conditions used for training. This table shows the average cross-track error ($\overline{d}$), the average velocity ($\overline{\|\mathbf{v}\|}$) and the total time taken to perform a full lap of the path by each agent. Also, to evaluate the 3rd approach agent in the same conditions of the two other agents, another simulation was made with this agent limiting its maximum velocity to 1 $^m/_s$. Note that 1st approach is denoted as *Agent 1* in the table, 2nd approach is *Agent 2* and so on. This nomenclature is maintained hereafter in this section.

**Table 3** Results for one lap of the lemniscate path, simulations with ground truth measurements.

| | $\overline{\mathbf{d}}$ **(m)** | **time (s)** | $\overline{\|\mathbf{v}\|}$ ($^m/_s$) |
|---|---|---|---|
| ***Agent 1*** | 0.1041 | 67.10 | 0.8707 |
| ***Agent 2*** | 0.0398 | 54.79 | 0.8780 |
| ***Agent 3*** | 0.0671 | 39.81 | 1.2276 |
| ***Agent 3*** ($v_{max} = 1$) | 0.0669 | 56.10 | 0.8696 |

The results performing a full lap of the lemniscate path while using the sensor measurements instead of ground truth values, are shown in Table 4. Same parameters and agents of Table 3 are evaluated. The trajectory on the *xy* plane followed by these agents is shown in Fig. 12. Fig. 13 shows the references of *yaw* angle ($\psi_{cmd}$) and velocity in the $x$ axis ($u_{cmd}$) computed by the *Agent 3* and the values of the angle $\psi$ and the velocity $u$ in the same simulation.

**Table 4** Results for one lap on the lemniscate path, simulations with sensor models.

| | $\overline{\mathbf{d}}$ **(m)** | **time (s)** | $\overline{\|\mathbf{v}\|}$ ($^m/_s$) |
|---|---|---|---|
| ***Agent 1*** | 0.1123 | 54.79 | 0.9476 |
| ***Agent 2*** | 0.0895 | 51.70 | 0.9484 |
| ***Agent 3*** | 0.0968 | 40.00 | 1.2338 |
| ***Agent 3*** ($v_{max} = 1$) | 0.0816 | 54.41 | 0.9111 |

As observed in the simulation results following the lemniscate path, the *Agent 2* appears to be the one that obtains the best results in terms of cross-track error. However, it is important to recall that this agent was only trained to perform well at the particular velocity of 1 $^m/_s$. On the other hand, *Agent 3* achieved a similar performance while reducing considerably the time taken to perform a full lap of the lemniscate. That is, this agent computes the optimal velocity at each part of the path, which allows the vehicle to accelerate in the straight lines, arriving at a maximum velocity of 1.82 $^m/_s$. Thus, it was able to increase the average velocity while maintaining almost the same error.

To analyse the performance of the agents while following a different path from the one that was used to train them, a new path was defined. This new path is a

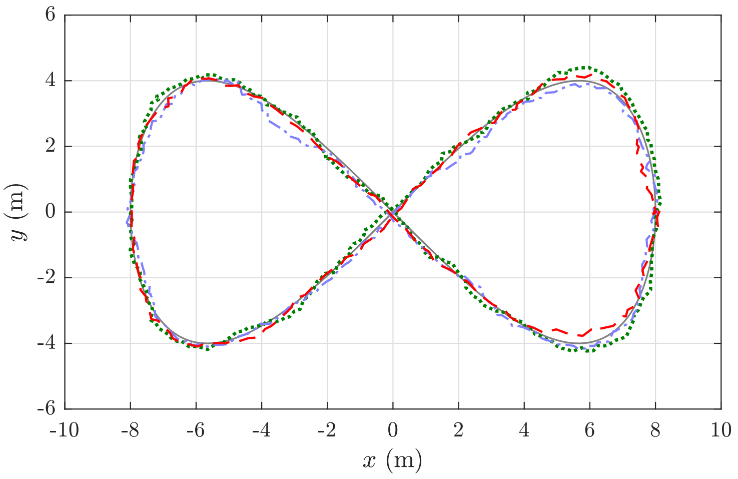

**Fig. 12** Trajectories on $xy$ of lemniscate path, simulation with sensor models: *Agent 1* in green (dotted line), *Agent 2* in blue (dash-dotted line) and *Agent 3* (dashed line) in red.

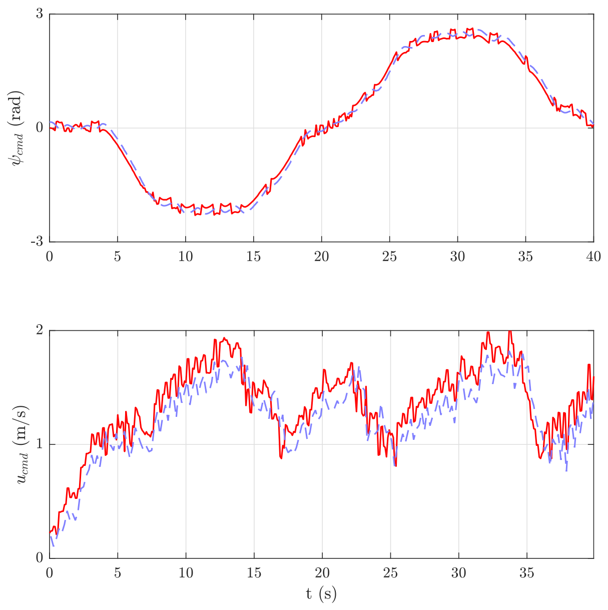

**Fig. 13** Actions of *Agent 3* following a lemniscate path, simulations with sensor models: references computed by agent (angle and velocity) in red and real values in blue (dashed line).

spiral, stated in Eq. (18). This time, parameter $A$ determined the rate at which the radius of the spiral grows and takes a value of 1.25. The virtual arc ($\gamma_p$) ranges from 0 to $2\pi$. Table 5 shows the simulation results obtained while following the spiral path with ground truth measures, while Table 6 presents the results of the agents following the same path with sensors measurements.

$$\begin{aligned} x_d &= -A\gamma_p \cos(\gamma_p) \\ y_d &= A\gamma_p \sin(\gamma_p) \end{aligned} \tag{18}$$

**Table 5** Results for one lap of the spiral path, simulations with ground truth measurements.

| | $\overline{\mathbf{d}}$ **(m)** | **time (s)** | $\overline{\|\mathbf{v}\|}$ (m/s) |
|---|---|---|---|
| ***Agent 1*** | 0.2907 | 34.30 | 0.8860 |
| ***Agent 2*** | 0.1840 | 32.43 | 0.8872 |
| ***Agent 3*** | 0.1418 | 23.52 | 1.2119 |
| ***Agent 3*** ($v_{max} = 1$) | 0.0759 | 32.10 | 0.8530 |

**Table 6** Results for one lap on the spiral path, simulations with sensor models.

| | $\overline{\mathbf{d}}$ **(m)** | **time (s)** | $\overline{\|\mathbf{v}\|}$ (m/s) |
|---|---|---|---|
| ***Agent 1*** | 0.3035 | 32.86 | 0.9448 |
| ***Agent 2*** | 0.2540 | 31.18 | 0.9366 |
| ***Agent 3*** | 0.1677 | 22.62 | 1.2262 |
| ***Agent 3*** ($v_{max} = 1$) | 0.0987 | 30.59 | 0.8830 |

The trajectories in the $xy$ plane of the three agents following the spiral path with sensor measures are shown in Fig. 14. These results correspond to the simulations presented in Table 6. Fig. 15 shows the angle and velocity references obtained by the *Agent 3* and their respective real values during that simulation. In that case the vehicle reached a maxim velocity of 1.71 $^m/_s$.

In the simulation results following the spiral path, the performance obtained by each agent varies more than in the results following the lemniscate path. With these results it is clear that, at least in simulation, the *Agent 3* is able to outperform the other agents, reducing the average cross-track error while travelling at higher velocities.

The simulations results show how the agents, even though having been trained with ground truth measurements following a lemniscate path, can also solve the path following problem with sensor measurements and follow other paths such as the stated spiral path. Next, the agents are tested in the real experimental platform.

## 6.2 Experimental

The results shown in this section were obtained with real experiments in different sessions during a week. Results were obtained with similar wind (aprox. 5-10 $^m/_s$) and GPS coverage (8-10 satellites) conditions. The experimental platform

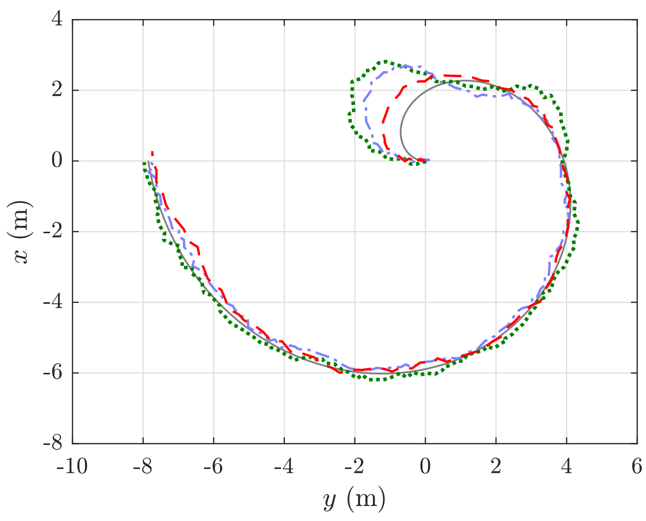

**Fig. 14** Trajectories on $xy$ of spiral path, simulations with sensor models: *Agent 1* in green (dotted line), *Agent 2* in blue (dash-dotted line) and *Agent 3* (dashed line) in red.

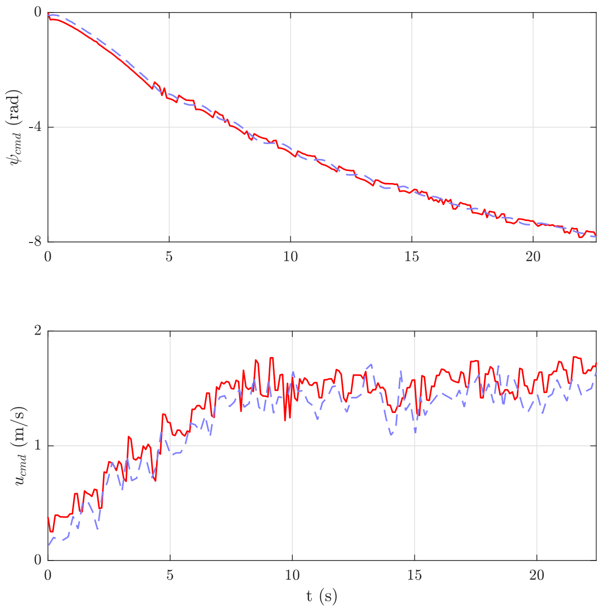

**Fig. 15** Actions of *Agent 3* following a spiral path, simulations with sensor models: references computed by agent (angle and velocity) in red and real values in blue (dashed line).

is the Asctec Hummingbird vehicle with a supplementary on-board PC (Odroid XU4Q) with ROS framework installed. Fig. 16 shows an image of the experimental platform. As can be observed, it is a an outdoors platform. The vehicle is equipped with an IMU sensor which, among other values, provides an estimation of the orientation of the vehicle, with a pressure sensor that provides the altitude measure and with a GPS that provides the position and an estimation of the vehicle's velocity on the $xy$ plane. In this paper, the states are obtained from the raw sensor measurements, without the use of any additional filter. Therefore, the implemented controllers must deal with noisy measures, specially the ones provided by the low-cost GPS sensor, which presents drifts and sometimes jumps in the position measurements. The autopilot ROS package is implemented in the on-board PC (attitude controller runs at 100Hz, velocity controller at 50Hz and altitude controller at 20Hz).

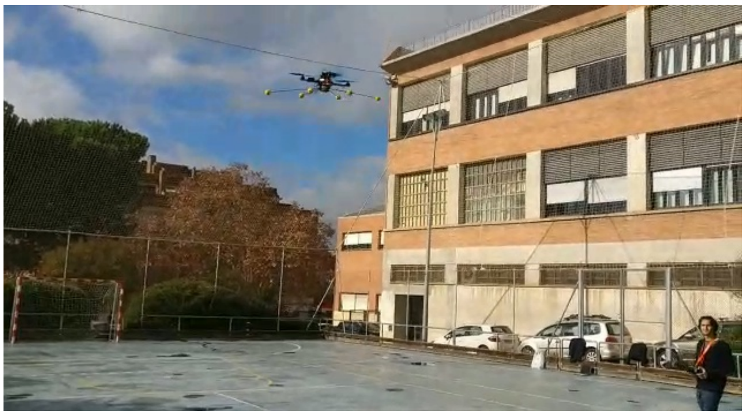

**Fig. 16** Outdoors experimental platform.

Since tensorflow library is required to operate at 64 bits and Odroid XU4Q works at 32 bits, the *DDPG* python agent cannot run in this PC. Instead, this program is executed in a laptop that communicates through Wi-Fi with the ROS master in the on-board PC. The laptop runs with linux Ubuntu with the i7-8550U intel processor and 16GB RAM. The *DDPG* python3 program runs at 10Hz.

The three agents were tested with real experiments following the same paths as in Sec. 6.1. However, although the three agents were able to solve the path following problem correctly, the results were not as good as expected. That is, the trajectory of the vehicle was slightly oscillating around the path. After various tests the authors concluded that this behaviour was due to a slight discrepancy between the simulation model and the real dynamics of the vehicle. Namely, the rotational dynamics around the $z$ axis were a little slower in the real vehicle.

In order to improve the performance of the agents two solutions were considered: the first one consists in training the agents in the experimental platform; the second one is to adjust the parameters of the agents to modify their dynamics. Training the agents during real flights can be harmful for the plant due to the unexpected behaviour of the vehicle. Furthermore, it has been observed that training with noisy measurements reduces the learning effectiveness. On the other hand, apparently, it does not exist any methodology for modifying the dynamics of the agents by changing some of their parameters. Indeed, out of the set of design and training parameters, involved only in the training phase, the *DDPG* algorithm does not have any other parameter to tune. However, in this paper we propose a form of modifying the control dynamics of the agent by adding a new parameter that will scale the output of the agent. That is, since the outputs of the agent are corrections (angle and velocity corrections), this parameter will regulate the speed at which the correction is made, and thus, it ends up regulating the dynamics of the angle and/or velocity reference too.

Since the discrepancy between the two models affects the *yaw* dynamics, only the angle action was scaled with the mentioned parameter. This new parameter, known as the angle correction constant ($k_a$), is apparent in Eq. (19) and it is set experimentally, $k_a = 2$; the value that provided the best performance from the different values that were tested experimentally. This correction constant was included in the three agents that were used to obtain all the experimental results that are presented in this paper. It is important to remember that this constant was just used in the experimental phase to improve the performance of the agents.

$$\psi_{cmd,k} = k_a a_k \Delta t + \psi_k \tag{19}$$

Next, the agents were tested with the same lemniscate of Sec. 6.1 (Eq. 16) with $A = 4$ and $\gamma_p$ ranging from 0 to $4\pi$. The results of the three agents performing a full lap of this path are shown in Table 7. The results of the *Agent 3* with the maximum velocity limited to 1 $^m/_s$ are also included. Again, the table shows the average cross-track error ($\overline{d}$), the total time and the average velocity of the vehicle ($\overline{\|\mathbf{v}\|}$). Fig. 17 presents the trajectory on the $xy$ plane of the agents while following this path. Furthermore, Fig. 18 shows the angle and velocity references computed by the *Agent 3* while following this lemniscate path, where references are shown in red and real values in blue dashed lines.

**Table 7** Experimental results for one lap on the lemniscate path.

| | $\overline{\mathbf{d}}$ **(m)** | **time (s)** | $\overline{\|\mathbf{v}\|}$ **($^m/_s$)** |
|---|---|---|---|
| ***Agent 1*** | 0.1739 | 55.90 | 0.8859 |
| ***Agent 2*** | 0.1140 | 55.01 | 0.9141 |
| ***Agent 3*** | 0.1682 | 39.39 | 1.6311 |
| ***Agent 3*** **($v_{max} = 1$)** | 0.2275 | 59.50 | 0.8829 |

The experimental results following the lemniscate reveal a behaviour that is very similar to the simulation results. The *Agent 2* displays again the best performance in terms of cross-track error, but the *Agent 3* achieves a similar distance error while increasing the average velocity, arriving at a maximum velocity of 2.49 $^m/_s$.

Another important remark from these experimental results is found in the last curve of the trajectory performed by the *Agent 3* (bottom-right curve in Fig. 17), a curve that the agent should clearly undertake better. This behaviour in the fast counter-clockwise curves appeared in all the experiments that we performed with this agent, and was even more evident in the counter-clockwise spiral paths. The cause of that strange pattern was found in the designed training framework. Although the agent was trained with asymmetrical lemniscates with diverse radius, this training framework resulted to be incomplete. The reason is that the agent was trained with a half lemniscate beginning with a counter-clockwise curve and ending

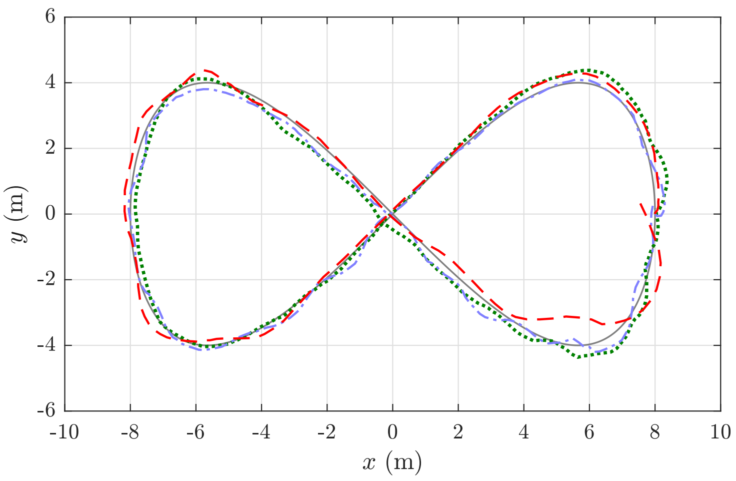

**Fig. 17** Trajectories on $xy$ of lemniscate path, experimental results: *Agent 1* in green (dotted line), *Agent 2* in blue (dash-dotted line) and *Agent 3* (dashed line) in red.

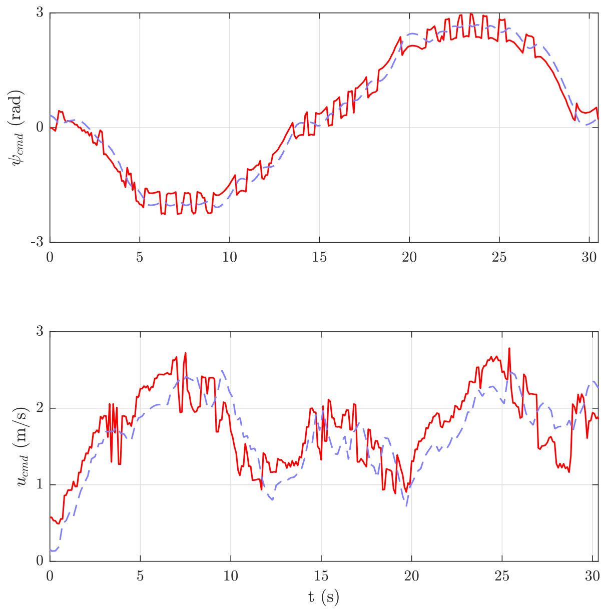

**Fig. 18** Actions of *Agent 3* following a lemniscate path, experimental results: references computed by agent (angle and velocity) in red and real values in blue (dashed line).

in a clockwise curve. This way, the first curve is always slower than the second one and the agent learned to perform fast clockwise curves and slow counter-clockwise curves. Consequently, it resulted in a bad behaviour while following counter-clockwise curves at high velocities. The solution to address this problem consists in changing the training framework of this $3^{rd}$ agent to have both types of curves at slow and fast velocities. It could be done, for instance, with full asymmetrical lemniscates.

Finally, the agents were tested with the spiral path defined in Eq. (18), with $A = 2.5$ and $\gamma_p$ ranging from 0 to $4\pi$. Table 8 presents the results of the three agents plus the *Agent 3* with limited velocity, just as in Table 7. The trajectories of the agents following this spiral path are shown in Fig. 19, and Fig. 20 shows the angle and velocity references computed from the actions of the *Agent 3*.

**Table 8** Experimental results for one lap on the spiral path.

| | $\overline{\mathbf{d}}$ **(m)** | **time (s)** | $\overline{\|\mathbf{v}\|}$ **(m/s)** |
|---|---|---|---|
| ***Agent 1*** | 0.2848 | 28.81 | 1.0503 |
| ***Agent 2*** | 0.2503 | 27.79 | 1.0460 |
| ***Agent 3*** | 0.2257 | 18.59 | 1.5844 |
| ***Agent 3*** ($v_{max} = 1$) | 0.2342 | 33.10 | 0.8826 |

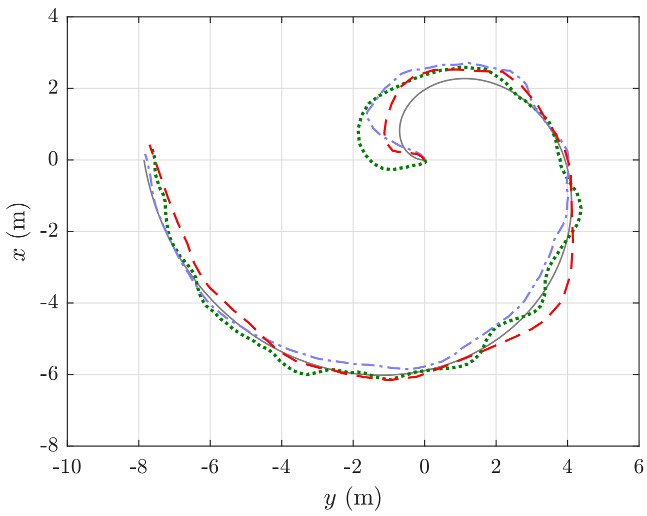

**Fig. 19** Trajectories on $xy$ of spiral path, experimental results: *DDPG v1* in green (dotted line), *DDPG v2* in blue (dash-dotted line) and *Agent 3* (dashed line) in red.

In the experimental results following the spiral path, the *Agent 3* outperforms the other agents by exhibiting a lower cross-track error with higher velocity. Specifically, this agent arrives at a maximum velocity of 2.52 $m/s$.

The initial trajectory of the agents when following the spiral path (Fig. 19) evidences a difference of behaviour between each of the three approaches presented in this paper. That is, the *Agent 1* is required to travel at a constant speed of 1 $m/s$ and only has information of the instantaneous distance and orientation error. Thus, due to the lack of anticipation, in the initial part of the path it starts going forward,

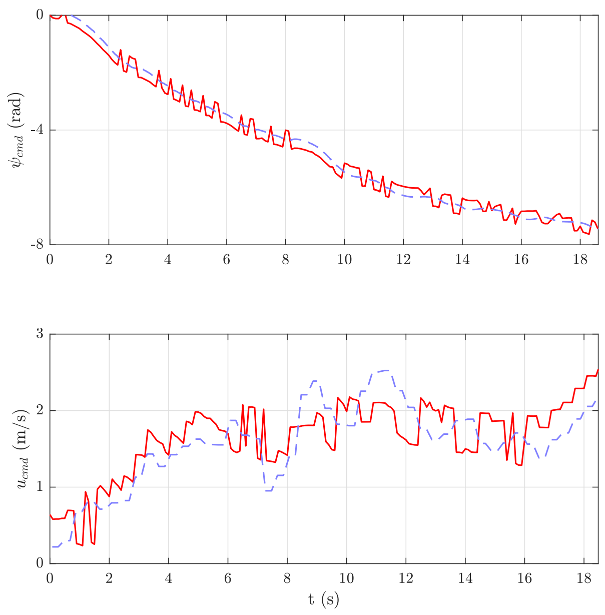

**Fig. 20** Actions of *Agent 3* following a spiral path, experimental results: references computed by agent (angle and velocity) in red and real values in blue (dashed line).

moving out of the path. The *Agent 2* moves also at a constant velocity. However, this agent has information about the upcoming orientation of the path, which allows it to anticipate the curve. Hence, in the initial part of the path, this agent starts moving towards the curve. Finally, the *Agent 3* knows the evolution of the path's curvature in advance and it is able to modify the longitudinal speed. That allows it to command slower speeds at the beginning of the path and turn towards the curve to follow the path as accurately as possible and then, when it is correctly oriented, start increasing the velocity.

## 7 Conclusions

In this paper, a deep reinforcement learning algorithm, the *Deep Deterministic Policy Gradient*, has been proposed to solve the path following problem in a quadrotor vehicle. The path following control computes the references for the velocity, the altitude and the angle in the $z$ axis that are then tracked by the autopilot controller. Three different *DDPG* approaches with different behaviours are presented. The first approach solves the PF problem only with information about the instantaneous position and angle errors. The second approach adds information about the upcoming path. Both approaches work at constant velocity. The third approach permits the agent to compute the optimal vehicle's velocity that adapts better to the shape of the path, according to the defined agent reward.

Each of the proposed agents arises as an improved version of the previous one, that is one of the main strengths of the methodology used in this work. The main structure, common in the three approaches, permits the incorporation of new functionalities (such as having anticipation to curves or adapting the vehicle's velocity) without changing the core of the agent. This is very promising since it means that new functionalities (e.g. wind disturbance rejection) could be straightforwardly integrated to the agent without altering the rest of the functionalities.

The agents were programmed in python using the tensorflow library. The designed training framework integrates the python script with Gazebo-ROS and uses RotorS, a the realistic multirotor simulator. This simulator includes a model of the Asctec Hummingbird, the quadrotor used in the experimental platform. Models of the real sensors of our platform were included in the simulator. The first and second agents were trained with lemniscate paths of fixed radius. They were also trained with different initial conditions to improve their performance in the experimental results. In order to learn the policy of the velocity action with different path's radius, the third agent was trained with asymmetrical lemniscates and changing the radius on each episode. The three agents were trained assuming ground truth measurements. The main advantage of training the agents in ROS is that it facilitates the transition from the simulator to the real plant. Furthermore, since ROS is a standard platform in the robotics field, it is supported by a large community, which can be very useful. The only concern to consider when training in ROS is that, since simulations are made real-time, it may become a time-consuming process.

The three agents were tested in simulations in the RotorS environment with realistic models of the sensors. They were evaluated with a lemniscate path and with a spiral path. Then, the agents were also tested in real experiments with the Asctec Hummingbird quadrotor following the same paths as in simulation. Even thought the agents were able to follow the pre-established paths correctly in the first experiments that were carried out, they performed worse than expected. The authors concluded that this behaviour was due to the small errors of the simulated model, which affected mainly to the yaw dynamics. To improve the performance of the agents a new parameter (angle correction constant, $k_a$) was included. This parameter scales the *yaw* action of the agent. And permits to modify the dynamics of the agent to cope with the model's discrepancy. Training the agents experimentally was dismissed since it can be harmful for the plant due to the unexpected behaviour of the vehicle. Furthermore, it was observed that training with noisy signals was unfruitful.

The agents were tested experimentally including the angle correction constant, which improved significantly their performance. In the lemniscate path, the *Agent 2* achieved the best performance in terms of average cross-track error ($0.114m$), but the *Agent 3* exhibited a similar distance error ($0.168m$) while being able to significantly increase the vehicle's velocity. In the spiral path, the *Agent 3* stands out over the other approaches by achieving the lowest average cross-track error (0.223) while travelling at higher velocities. In conclusion, the experimental results show how the agents are able to successfully follow the spiral path, a different path from the one that they were trained with. And thus, it is proved that the proposed approach is able to find a generalized solution for the path following problem with adaptive velocity.

Nevertheless, a strange pattern was observed in the *Agent 3* while performing counter-clockwise curves at high velocities (Fig. 17). This behaviour was attributed to the design of the training environment. That is, since the agent was trained with half asymmetrical lemniscates, the first curve of the path (counter-clockwise) is followed in the transient part of the experiment (velocity is still increasing). Therefore, the agent ended up training the clockwise curves at faster velocities than the counter-clockwise ones. This fact highlights the importance of having not only a proper structure and parametrization of the agent, but also a rich, complete and adequate training framework. The solution to this issue would be to train with a different path that permits the agent to learn both curves at different velocities.

Future work is to study the effect of the trained path in the performance of the agent and to find the best training environment to exploit the benefits of the agent. Other future work is to improve the presented agent to make it capable of solving other challenging problems such as wind disturbance rejection or reactive obstacle avoidance.

## Conflict of interest

The authors declare that they have no conflict of interest.

## References

1. A. P. Aguiar, J. P. Hespanha, and P. V. Kokotovic, "Performance limitations in reference tracking and path following for nonlinear systems," *Automatica*, vol. 44, no. 3, pp. 598–610, 2008.
2. I. Kaminer, O. Yakimenko, A. Pascoal, and R. Ghabcheloo, "Path generation, path following and coordinated control for time-critical missions of multiple UAVs," in *2006 AMERICAN CONTROL CONFERENCE*, vol. 1-12, pp. 4906+, 2006.
3. B. Rubí, R. Pérez, and B. Morcego, "A Survey of Path Following Control Strategies for UAVs Focused on Quadrotors," *Journal of Intelligent & Robotic Systems*, vol. 98, pp. 241–265, 2019.
4. P. B. Sujit, S. Saripalli, and J. B. Sousa, "Unmanned aerial vehicle path following: A survey and analysis of algorithms for fixed-wing unmanned aerial vehicless," *IEEE Control Systems*, vol. 34, no. 1, pp. 42–59, 2014.
5. G. Heredia and A. Ollero, "Stability of autonomous vehicle path tracking with pure delays in the control loop," *Advanced Robotics*, vol. 21, no. 1-2, pp. 23–50, 2007.
6. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.
7. T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *2016 International Conference on Learning Representations (ICLR)*, 2016.
8. J. C. Caicedo and S. Lazebnik, "Active object localization with deep reinforcement learning," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2488–2496, 12 2015.
9. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016.
10. R. Yu, Z. Shi, C. Huang, T. Li, and Q. Ma, "Deep reinforcement learning based optimal trajectory tracking control of autonomous underwater vehicle," in *2017 36th Chinese Control Conference (CCC)*, pp. 4958–4965, 7 2017.
11. L. P. Tuyen and T. Chung, "Controlling bicycle using deep deterministic policy gradient algorithm," in *2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pp. 413–417, 6 2017.
12. L. Li, Y. Lv, and F. Wang, "Traffic signal timing via deep reinforcement learning," *IEEE/CAA Journal of Automatica Sinica*, vol. 3, pp. 247–254, 7 2016.

13. A. Rodriguez-Ramos, C. Sampedro, H. Bavle, P. de la Puente, and P. Campoy, “A deep reinforcement learning strategy for uav autonomous landing on a moving platform,” *Journal of Intelligent & Robotic Systems*, vol. 93, pp. 351–366, 2 2019.
14. K. Kang, S. Belkhale, G. Kahn, P. Abbeel, and S. Levine, “Generalization through simulation: Integrating simulated and real data into deep reinforcement learning for vision-based autonomous flight,” in *2019 International Conference on Robotics and Automation (ICRA)*, 2019.
15. W. Koch, R. Mancuso, R. West, and A. Bestavros, “Reinforcement learning for uav attitude control,” *ACM Trans. Cyber-Phys. Syst.*, vol. 3, no. 2, 2019.
16. N. O. Lambert, D. S. Drew, J. Yaconelli, R. Calandra, S. Levine, and K. S. J. Pister, “Low level control of a quadrotor with deep model-based reinforcement learning,” *IEEE Robotics and Automatic Letters*, vol. 4, no. 4, 2019.
17. D. Mittall, K. Kumar, S. N. Hashmi, P. Kumar, A. Nanda, and S. Chandra, “Performance comparison of deep and shallow network for quadcopter automation,” in *2018 IEEE 13th International Conference on Industrial and Information Systems (ICIIS)*, pp. 143–147, 2018.
18. R. Polvara, M. Patacchiola, S. Sharma, J. Wan, A. Manning, R. Sutton, and A. Cangelosi, “Toward end-to-end control for uav autonomous landing via deep reinforcement learning,” in *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 115–123, 2018.
19. B. Rubí, B. Morcego, and R. Pérez, “A Deep Reinforcement Learning Approach for Path Following on a Quadrotor,” in *2020 European Control Conference*, 2020.
20. A. P. Aguiar and J. P. Hespanha, “Trajectory-tracking and path-following of underactuated autonomous vehicles with parametric modeling uncertainty,” *IEEE Transactions on Automatic Control*, vol. 52, pp. 1362–1379, 8 2007.
21. D. Cabecinhas, R. Cunha, and C. Silvestre, “Rotorcraft path following control for extended flight envelope coverage,” in *Proceedings of the 48th IEEE Conference on Decision and Control, held jointly with the 28th Chinese Control Conference (CDC/CCC)*, pp. 3460–3465, 2009.
22. D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic policy gradient algorithms,” in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, pp. I–387–I–395, JMLR.org, 2014.
23. S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *ArXiv*, vol. abs/1502.03167, 2015.
24. R. S. Sutton and A. Barto, *Reinforcement Learning: An Introduction.* The MIT Press, 1992.
25. F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, *RotorS—A Modular Gazebo MAV Simulator Framework*, pp. 595–625. Cham: Springer International Publishing, 2016.