Reinforcement Learning based approach for quadrotor landing

I. Context and Challenges

Introduction

The autonomous landing of a quadrotor drone on a moving platform is a complex task that has traditionally been fully handled by PID controllers, which regulate roll, pitch, yaw, and thrust angles. However, these approaches present several limitations, including the need for an accurate system model and a parameter optimization phase that is often labor-intensive and sensitive to operating conditions.

In recent years, Deep Reinforcement Learning has been explored as an alternative for learning control policies based exclusively on interaction with the environment. Although DRL offers a high potential for generalization and adaptability to complex scenarios, it requires long training times and computationally expensive hardware, making it less practical for real-time applications or devices with limited resources.

The objective of this approach is to demonstrate the effectiveness of a Reinforcement Learning-based method without resorting to deep learning techniques, significantly reducing training time and dependence on advanced hardware. This is made possible through a discrete formulation of the problem and the use of advanced state-space discretization techniques. The proposed approach enables control of the drone's roll and pitch angles, improving maneuverability and stability during landing. The code can be found at the following link: Github repository

II. Proposed Approach

The proposed approach addresses the highlighted issues through the following strategies:

1) **Decoupling of dynamics:** The control of roll and pitch angles is handled separately based on the assumption that the two movements are independent. In a quadrotor drone, the dynamics are inherently nonlinear and coupled; however, under the assumption of small angular variations and linearization around hover conditions, the drone's dynamics can be modeled as four independent subsystems along the longitudinal, lateral, vertical directions, and around the yaw axis. This simplification is commonly adopted in the literature and helps reduce the complexity of the problem.
2) **Reuse of the RL agent:** A single RL agent is trained to control pitch and subsequently replicated for roll, avoiding the need for additional training.
3) **Double Q-Learning algorithm:** The adoption of a tabular approach ensures memory efficiency, eliminating the need for computationally expensive hardware. Moreover, it helps improve the stability and convergence of training compared to neural network-based methods.
4) **Curriculum Learning and Multiresolution Discretization:** The former structures the learning process into a sequence of progressively more challenging tasks, while the latter allows the state space to be represented with variable resolution, increasing precision in critical regions of the task. In addition to accelerating algorithm convergence, this mitigates the issue of high dimensionality (*curse of dimensionality*), improving learning effectiveness. A more detailed explanation of the approach will be provided later.

III. Simulation Environment and Reference Frame

The simulation environment is modeled in Gazebo and consists of the following elements:

A. *Landing Platform*

The landing platform consists of six fixed joints: four for the wheels, one for the contact bumper, and one as a reference for the *world frame*. The platform's motion is described by a second-order differential equation that defines a one-dimensional sinusoidal movement:

$$\ddot{\mathbf{er}}_{mp} = -\left(\frac{v_{mp}^2}{r_{mp}}\right)\begin{bmatrix}\sin(\omega_{mp}t)\\0\\0\end{bmatrix}, \tag{1}$$

$$\omega_{mp} = \frac{v_{mp}}{r_{mp}}. \tag{2}$$

B. *Drone*

The drone used in the simulation is a symmetric quadrotor configured in an **X** arrangement, with the following physical parameters:

- **Mass:**

$$m = 0.716 \text{ kg} \tag{3}$$

- **Inertia Matrix:**

$$\mathbf{J} = \begin{bmatrix}0.007 & 0.0 & 0.0\\0.0 & 0.007 & 0.0\\0.0 & 0.0 & 0.012\end{bmatrix} \tag{4}$$

- **Rotor Configuration:**
- **Rotation Direction:**
  - Rotors 1 and 3 ($i = 1, 3$): clockwise.
  - Rotors 2 and 4 ($i = 2, 4$): counterclockwise.

Rotor	Arm Length [m]	C_f [N·s^2]	C_m [N·m·s^2]
i	0.17	8.54×10^{-6}	1.6×10^{-2}

TABLE I: Quadrotor rotor configuration

C. *Fly-Zone*

The allowed flight area is a cube with dimensions $9 \times 9 \times 9$ m.

D. *Reference Frame*

To simplify the control problem, the **stability frame** is adopted. This reference frame is anchored to the drone's Z-axis and remains constantly parallel to the ground.

This approach allows:

- Simplifying transformations from the drone body frame and platform body frame to the stability frame.
- Focusing exclusively on relative differences (position, velocity, acceleration, and orientation) between the drone and the platform, without considering inclinations along x and y, which do not directly affect the landing task.
- Isolating the drone's dynamics from the platform's dynamics, enabling the RL model to learn a control policy based on relative variations, thus reducing the complexity of the overall problem.

1) *Computation of the Relative State in Frame s*

To express the relative dynamics between the drone and the platform, the following initial conditions are considered:

- The drone and platform start in a *hover* state, with initial angular velocities set to zero:

$$\dot{s\boldsymbol{\phi}}_{mr,0} = \dot{s\boldsymbol{\phi}}_{mp,0} = \mathbf{0}, \quad s\boldsymbol{\phi}_{mr,0} = s\boldsymbol{\phi}_{mp,0} = \mathbf{0}.$$

- The initial conditions of the translational part, expressed in the terrestrial frame e, are transformed into the stability frame s, where frame s is defined on the drone.

The equations of relative dynamics are defined as:

$$\ddot{s\boldsymbol{\phi}}_{\text{rel}} = \ddot{s\boldsymbol{\phi}}_{mp} - \ddot{s\boldsymbol{\phi}}_{mr}, \tag{5}$$

$$\ddot{\mathbf{s}}_{\text{rel}} = \ddot{\mathbf{s}}_{mp} - \ddot{\mathbf{s}}_{mr}. \tag{6}$$

From these relations, the **relative state vector** used by the agent to select actions is defined as:

$$p_c = [p_{c,x}, p_{c,y}, p_{c,z}]^T = s\mathbf{r}_{\text{rel}}, \tag{7}$$

$$v_c = [v_{c,x}, v_{c,y}, v_{c,z}]^T = s\dot{\mathbf{r}}_{\text{rel}}, \tag{8}$$

$$a_c = [a_{c,x}, a_{c,y}, a_{c,z}]^T = s\ddot{\mathbf{r}}_{\text{rel}}, \tag{9}$$

$$\phi_c = [\phi_{\text{rel}}, \theta_{\text{rel}}, \psi_{\text{rel}}]^T = s\boldsymbol{\phi}_{\text{rel}}. \tag{10}$$

Where:

- p_c represents the relative position of the drone with respect to the platform.
- v_c is the relative velocity.
- a_c is the relative acceleration, computed as the numerical derivative of velocity and filtered using a first-order Butterworth filter to reduce noise.
- ϕ_c represents the relative orientation, expressed in terms of roll, pitch, and yaw angles in frame s.

IV. The Node Architecture

When developing a Reinforcement Learning-based approach, the most natural practical choice is to use Python. This offers several advantages in terms of flexibility and implementation speed but also presents limitations, particularly regarding simulation frameworks that are fully integrated with Python, which are rather scarce. To address this issue, Gazebo was chosen as the simulation environment, making it necessary to integrate a communication layer between the training logic and the simulator using ROS.

The system architecture is based on five main nodes, described below.

Refer to Figure 1 to understand the hierarchy of components.

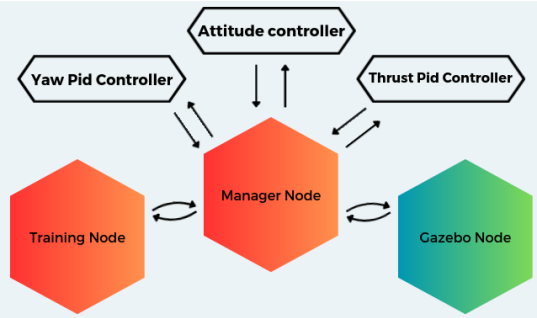


Fig. 1: Node Architecture

A. *Main Node*

The **Main Node** serves as the core of the architecture, responsible for managing communication between different modules and synchronizing the simulated environment. Specifically, it performs the following tasks:

- Initializes and computes the platform's motion equation, updating its state and publishing it at each simulation step.
- Retrieves the state of the drone and the platform from Gazebo with respect to the *world frame*.
- Computes and publishes in the `ROS transformation tree` the transformation required to convert the state of the drone and the platform into the *stability frame*.
- Computes the relative state between the drone and the platform and publishes the observation for the training node.
- Receives and publishes setpoints for the PID controllers, interfacing with the Attitude Controller for managing the forces and moments generated.
- Continuously communicates with the simulation node to ensure the correct update of states.

B. *Training Node*

The training node contains the modeling and learning logic for Reinforcement Learning-based control. Specifically, the node:

- Models the problem as a *Markov Decision Process*.
- Discretizes the state space according to the adopted technique.
- Computes the *reward* associated with the current state.
- Determines the optimal discrete action based on the learned policy.
- Sets and communicates the required setpoints to the controllers.

C. Yaw and Thrust Controller Nodes

The Yaw and Thrust nodes implement two independent PID controllers responsible for regulating the yaw angle and vertical thrust.

D. Attitude Controller Node

This node translates the received inputs into angular velocities for the rotors.

Having defined the general architecture of the nodes, we now delve into the details of its components, starting with the controllers.

V. Yaw and Thrust Controller

In the proposed system, the control of roll and pitch angles is managed by the Reinforcement Learning algorithm, while the regulation of **yaw** and **thrust** is handled separately through two independent PID controllers.

A. Operation of the PID Controllers

These nodes receive the setpoints generated by the training node and return the necessary *effort* to correct the drone's trajectory.

- **Yaw Controller:** Regulates the drone's orientation relative to the Z-axis, ensuring that it follows the desired direction.
- **Thrust Controller:** Controls the drone's vertical thrust, ensuring that it can compensate for altitude variations and manage the approach phase to the platform.

B. Mathematical Formulation of PID Control

The PID controllers generate a control input based on the error between the desired *setpoint* and the current measured observation. The continuous control law is expressed as:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{11}$$

where:

- $e(t) = s(t) - y(t)$ error between the setpoint $s(t)$ and the measured value $y(t)$.
- K_p is the proportional gain.
- K_i is the integral gain.
- K_d is the derivative gain.

1) Discrete Implementation

In practice, PID control is implemented in discrete form, with periodic updates based on a sampling interval Δt, synchronized with the simulation update rate:

$$u[k] = K_p e[k] + K_i \sum_{j=0}^{k} e[j]\Delta t + K_d \frac{e[k] - e[k-1]}{\Delta t} \tag{12}$$

This formulation enables the implementation of PID control in a digital environment with discrete-time updates.

2) Algorithmic Implementation

Below is the implementation of the PID controller, including **anti-windup** management for the integral term.

Algorithm 1: PID Controller for Yaw and Thrust

Data: Setpoint s, Current State y, PID Parameters K_p, K_i, K_d
Result: Control effort u

Initialize integral error: $e_{\text{int}} \leftarrow 0$;
Initialize previous error: $e_{\text{prev}} \leftarrow 0$;
while *the system is active* **do**
  Acquire current state y ;
  Current time: $t_{\text{now}} \leftarrow$ current time ;
  Compute time interval: $\Delta t \leftarrow t_{\text{now}} - t_{\text{prev}}$;
  $e \leftarrow s - y$;
  `// Compute proportional term`
  $P \leftarrow K_p \cdot e$;
  `// Compute integral term with anti-windup`
  $e_{\text{int}} \leftarrow e_{\text{int}} + e \cdot \Delta t$;
  $e_{\text{int}} \leftarrow \text{clip}(e_{\text{int}}, e_{\text{min}}, e_{\text{max}})$;
  $I \leftarrow K_i \cdot e_{\text{int}}$;
  `// Compute derivative term`
  $D \leftarrow K_d \cdot \frac{e - e_{\text{prev}}}{\Delta t}$;
  `// Compute control effort`
  $u \leftarrow P + I + D$;
  Publish control effort u ;
  $e_{\text{prev}} \leftarrow e$;
  $t_{\text{prev}} \leftarrow t_{\text{now}}$;

VI. Attitude Controller

The *Attitude Controller* is responsible for regulating the drone's orientation. After the yaw and thrust controllers have generated their respective *control efforts* and the setpoints for roll and pitch have been defined by the training node, the **Attitude Controller** computes the required torque moments to correct the attitude.

These moments, together with the total thrust, are then converted into rotor angular velocities using the inverse of the drone's allocation matrix.

A. *Mathematical Formulation*

The implemented controller follows the model proposed by Lee et al. for geometric control on $SO(3)$, leveraging the representation through rotation matrices.

1) Attitude Error

The error between the current rotation R and the desired rotation R_d is defined as:

$$e_R = \frac{1}{2}(R_d^T R - R^T R_d)^\vee \tag{13}$$

where the symbol $\vee$ represents the operation that converts a skew-symmetric matrix into a three-dimensional vector.

2) Angular Velocity Error

The angular velocity error is defined as:

$$e_\omega = \omega - R^T R_d \omega_d \tag{14}$$

3) Control Law

The control moment is defined by the following equation:

$$M = -K_R e_R - K_\omega e_\omega + \omega \times J\omega \tag{15}$$

where:

- K_R is the gain matrix for attitude correction.
- K_ω is the gain matrix for angular velocity correction.
- The term $\omega \times J\omega$ compensates for gyroscopic forces.

4) Allocation Matrix

The torque moments M and the total thrust T are converted into rotor angular velocities ω using the allocation matrix F:

$$\omega^2 = F^{-1} \begin{bmatrix} M_x \\ M_y \\ M_z \\ T \end{bmatrix} \tag{16}$$

where:

- T is the required total thrust.
- M_x, M_y, M_z are the torque moments along their respective axes.
- F is the allocation matrix, which depends on the rotor configuration and allows the distribution of control commands to individual propellers.

B. *Algorithmic Implementation*

Below is the implementation of the attitude control algorithm in $SO(3)$:

Algorithm 2: Attitude Control in $SO(3)$

Data: Odometry $odom$, Current orientation R, Current state $(\phi, \theta, \dot{\psi}, T)$
Result: Angular velocities for the rotors

Acquire current state: ω, R ;
Acquire desired commands: $(\phi, \theta, \dot{\psi}, T)$;
`// Construction of the desired rotation matrix`
Extract current yaw angle: $\psi \leftarrow \text{atan2}(R_{1,0}, R_{0,0})$;
Construct $R_{\text{yaw}} \leftarrow$ rotation around $(0, 0, 1)$ with angle ψ ;
Construct $R_{\text{roll}} \leftarrow$ rotation around $(1, 0, 0)$ with angle ϕ ;
Construct $R_{\text{pitch}} \leftarrow$ rotation around $(0, 1, 0)$ with angle θ ;
Compute desired rotation: $R_d \leftarrow R_{\text{yaw}} R_{\text{roll}} R_{\text{pitch}}$;
`// Compute attitude error`
$e_R \leftarrow \frac{1}{2}(R_d^T R - R^T R_d)^\vee$;
`// Compute desired angular velocity`
$\omega_d \leftarrow [0, 0, \dot{\psi}]^T$;
`// Compute angular velocity error`
$e_\omega \leftarrow \omega - R^T R_d \omega_d$;
`// Compute control moment`
$M \leftarrow -K_R e_R - K_\omega e_\omega + \omega \times J\omega$;
`// Convert moment and total thrust into rotor angular velocities`
Compute $\omega^2 \leftarrow F^{-1}[M_x, M_y, M_z, T]^T$;
`// Publish motor command`
Publish motor command ω ;

VII. Reinforcement Learning Setting

The reinforcement learning (RL) task in this work is to learn a policy that enables a multi-rotor vehicle to land on a moving platform. The assumption of decoupled dynamics allows us to treat this problem as a one-dimensional (1D) control task, significantly simplifying the learning process while preserving the core challenge of autonomous landing.

In reinforcement learning, approaches are broadly classified into *model-free* and *model-based* categories, depending on whether the agent explicitly assumes or learns a transition model.

- **Model-based RL** explicitly incorporates a transition model, which is either learned from data or assumed a priori. This model is then used to simulate imagined rollouts, optimize actions before execution, or improve sample efficiency by leveraging known system dynamics.
- **Model-free RL** does not use an explicit transition model. Instead, it learns a policy or a value function purely from experience by interacting with the environment. The agent updates its policy or value function based solely

on observed rewards, without requiring knowledge of the transition dynamics.

The approach taken in this work can be considered a hybrid between model-free and model-based RL. The RL agent itself does not rely on an explicit platform dynamics model for predicting the next state during training or execution. However, the environment is designed using a known kinematic model of the platform. This prior knowledge informs critical design choices, such as state space discretization, curriculum learning structure, and hyperparameter selection. By incorporating structured biases through this knowledge, the learning process becomes more efficient and stable than a purely model-free approach with random exploration.

Since RL builds upon the Markov Decision Process (MDP) framework, it is essential to define its fundamental components:

- **State space** (S): The set of observations that describe the environment under which the RL agent operates.
- **Action space** (A): The discrete set of possible actions the agent can take at any given state.
- **Transition model** ($P(s'|s,a)$): Defines the probability of transitioning from state s to state s' after taking action a. While a well-defined MDP typically requires an explicit transition model, in this work, the transition dynamics exist but are unknown to the agent.
- **Reward function** ($R(s,a)$): Determines the immediate reward received by the agent for taking an action in a particular state, shaping the learning process through positive and negative incentives.

To ensure interpretability of hyperparameters and ease of training, the authors argue for a tabular reinforcement learning approach. A suitable candidate in this context is **Q-learning**, an off-policy algorithm that updates its policy using another policy—the greedy max policy. However, this greedy selection often leads to overestimation of Q-values, which can slow down learning and introduce instability. To address this issue, **Double Q-learning** is used, which mitigates overestimation by maintaining two separate Q-value estimates and selecting actions based on one while updating with the other. This results in a more stable and reliable learning process, making it the preferred choice for this application.

A. *Action discretization*

In tabular Q-learning, the size of the action space significantly impacts the learning process due to the curse of dimensionality. To ensure efficient learning, we adopt a minimal yet effective action space that consists of three discrete actions: increasing, decreasing, or maintaining the pitch angle of the UAV. This choice reduces the number of Q-values the agent needs to learn, making training both faster and more stable.

The action space is discretized using fixed pitch angle increments. The maximum pitch angle is set as:

$$\theta_{\max} = 21.37723^\circ \tag{17}$$

and since the number of pitch angle intervals is fixed at: $n_\theta = 3$, this leads to a step size of:

$$\Delta\theta = \frac{\theta_{\max}}{n_\theta} = 7.12574^\circ \tag{18}$$

B. *State discretization*

Similar to the action space, the state space discretization follows a three-bin structure. Increasing the number of bins would lead to a larger Q-table, making learning less sample-efficient. The three-bin structure provides a balance between granularity and computational feasibility.

As we previously mentioned, the problem is structured as a curriculum learning task, the state discretization dynamically evolves as the agent progresses through different curriculum steps. The core idea behind curriculum learning is to introduce the agent to progressively harder versions of the landing problem by refining the definitions of success and failure. This is achieved through two key parameters:

- **Goal Region**: Defines how close the UAV needs to be to the platform for the agent to consider it "on target." It starts large (allowing loose precision) and shrinks progressively as the curriculum advances, requiring finer control.
- **Limit Region**: Defines the outermost boundary beyond which the UAV is considered to have failed. If the UAV exceeds these limits, it receives a negative reward, and the episode terminates. Like the goal region, this limit starts large (allowing more mistakes in early learning stages) and shrinks over successive curriculum steps to enforce stricter precision.

Both the goal and limit values are determined from the kinematic modeling of the moving platform, ensuring that the discretization aligns with the physical constraints of the problem.

The goal region is shrunk exponentially over successive curriculum steps. This approach allows the agent to first learn coarse control before being required to refine its policy for more precise landings. Given this structure, the discretization function effectively encodes how far the current state is from the goal and the direction in which the agent should adjust its behavior.

Each state variable—position, velocity, acceleration, and pitch—is classified into one of three bins:

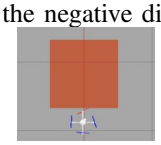
Since the goal region shrinks over successive curriculum steps, this three-bin structure allows for a natural and progressive refinement of what the RL agent considers as "good behavior."

C. *Reward shaping*

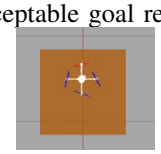
A fundamental challenge in reinforcement learning is the **credit assignment problem**: when an agent receives a reward only at the end of an episode, it struggles to determine which past actions were responsible for that reward. This issue becomes particularly pronounced in long-horizon tasks like landing a UAV, where delayed rewards make it difficult for the agent to learn effective policies.

To mitigate this, a **reward shaping** technique is employed. Instead of only rewarding the agent upon success, incremental

Left of Target (Negative bin, 0): The UAV is positioned too far in the negative direction.



On Target (Goal region, 1): The UAV is within the acceptable goal region.



Right of Target (Positive bin, 2): The UAV is positioned too far in the positive direction.

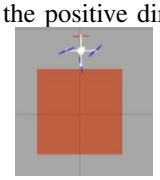


Fig. 2: UAV position relative to the target.

rewards are provided at each step, guiding the learning process by associating actions with their immediate consequences. The reward function is structured to encourage desirable behavior while penalizing inefficient or incorrect actions. The shaping terms in the reward function include:

- r_p: Reward for reducing the position error.
- r_v: Reward for reducing the velocity error.
- r_θ: Reward for minimizing unnecessary pitch changes.
- r_{dur}: Negative reward for taking too long to land.

Since the problem is designed as a curriculum learning task, each step progressively increases the difficulty of the landing challenge. As the UAV advances through the curriculum:

- The allowed error margins for position and velocity shrink, requiring more precise control.
- The limits on failure conditions become stricter, increasing the likelihood of termination if the agent makes mistakes.

If training were restricted only to the hardest curriculum step, the agent would face several issues:

- It would start learning in an unforgiving environment where mistakes are severely penalized, making exploration difficult.
- Learning could stagnate or fail entirely due to excessive early failures.
- The average reward would be significantly lower, providing less meaningful feedback for improvement.

Instead, continuing training across multiple curriculum steps ensures:

- **Encouraged exploration**: Earlier curriculum steps provide higher rewards, allowing the agent to explore effective strategies before facing stricter conditions.
- **Retention of useful behaviors**: Training across easier steps prevents the agent from forgetting beneficial strategies learned in earlier stages.
- **Stabilized learning**: If the agent struggles in the hardest step, it can still improve by refining behaviors in less punishing environments, similar to experience replay.

An additional consideration is that as rewards shrink throughout the curriculum, Q-values (expected cumulative rewards) naturally decrease due to the increasing difficulty of the task. To counteract this effect, two mechanisms are implemented:

- **Knowledge transfer**: The agent's learned policy is propagated to the next curriculum step, preventing it from having to relearn from scratch.
- **Reward scaling**: The reward function is adjusted to account for the reduced reward magnitudes, ensuring that the agent continues to receive meaningful feedback at every step.

These strategies collectively enhance sample efficiency, stabilize training, and improve policy generalization, ultimately leading to a more robust landing behavior.

D. Training workflow

At each curriculum step, a new environment is initialized with updated constraints reflecting the current difficulty level. The agent progresses through the training process using a structured reinforcement learning pipeline designed for incremental learning and stability.

For each episode:

- The agent initializes in a given state and selects an action based on an exploration strategy (*epsilon-greedy policy*).
- The agent interacts with the environment, receiving a reward and transitioning to the next state.
- The Double Q-learning update is applied, refining the policy by updating one of the two Q-tables based on the observed reward and next state.
- The process repeats until the episode terminates, either in success or failure.

The agent's performance is continuously evaluated. If the success rate surpasses the predefined threshold (96

To maintain learning efficiency and prevent catastrophic forgetting, **transfer learning** is applied between curriculum steps. The learned Q-values from the previous step are scaled and propagated forward, allowing the agent to retain knowledge and adapt to the increasing complexity of the task without starting from scratch.

Throughout training, key performance metrics—including success rate, exploration rate, and learning rate—are logged to monitor stability and convergence. Periodic saving of training progress ensures that the learning process remains robust and can be resumed if needed. This structured workflow allows the agent to incrementally refine its policy, effectively managing the growing difficulty of the landing task.