Reinforcement Learning to fly a Flapping Wing Drone

Jack Leckert and Nicolas Samy

Abstract— **The Aerobat, a rotor-based micro-air vehicle, is able to simulate bat-like flapping dynamics, incorporating elbow flexion and extension during flight to minimize negative lift and maximize aerodynamic efficiency. The aerodynamics of the flapping wing MAV are modeled using lifting line theory and Wagner's function. Model-Based controllers (PID and MPC) offer good performance for hovering and flying a square trajectory with the help of four propellers to maintain stability. A learning-based controller utilizing proximal policy optimization (PPO) is later trained by reducing slowly the action of propellers. This controller learns the dynamics of flapping wings, for which explicit equations may be unknown, thus offering the potential to rely less on propellers for flight control. The investigation covers two operational modes: hovering and linear flight. While initial results are promising, further design iterations are necessary to achieve stable flight across both operational modes.**

I. Introduction

The deployment of flapping-wing Micro Air Vehicles (MAVs) offers a novel alternative to quadcopters to navigate challenging settings, such as confined tunnels. Traditional rotor-based drones often struggle within such confines due to strong downdrafts and adverse surface effects, which can disrupt stable flight. In contrast, our flapping wing MAV design based on study [1] potentially harnesses these aerodynamic complexities to its advantage, promoting enhanced maneuverability and stability in turbulent airflow typical of tunnel-like structures. In the realm of flapping wing robotics, the predominant research has traditionally centered on insect-inspired designs that mimic the biomechanics of natural flyers. Our approach diverges significantly from conventional models by introducing a tailless configuration with morphing wings. This innovative design not only distinguishes our MAV from others, but also optimizes its adaptability and efficiency in confined spaces, where precision and flexibility in movement are crucial. The structural innovation lies in its kinetic sculpture, a computational framework that controls the wing's motion divided into proximal and distal segments. This unique system enables the Aerobat to simulate bat-like flapping dynamics, incorporating elbow flexion and extension during flight to minimize negative lift and maximize aerodynamic efficiency. Despite its advanced design, the Aerobat currently faces significant challenges in terms of control systems. Existing controllers are inadequate for handling the intricate dynamics of flapping wing flight, particularly in the unstructured and unpredictable environments it is designed to navigate.

The primary focus of our project is twofold: to develop a comprehensive mathematical model of the MAV, and to engineer robust and advanced control systems capable of stable flight. Flight trajectories will be limited to hovering and linear flight and different model-based and learning-based controllers performance will be compared for each trajectories. By achieving these objectives, we aim to significantly enhance the capabilities of MAVs in complex environments, paving the way for new applications in industrial inspection, search and rescue operations, and environmental monitoring.

II. Simulation Environment and Drone Design

The simulation of the flapping wing MAV uses a hybrid approach through MuJoCo, a robot physics simulator. The simulation for Flappy is composed of three parts: kinematics, dynamics, and aerodynamics. The MuJoCo directly handles the kinematics and dynamics of the simulation while the aerodynamics are computed using a custom Python function. In this section, the first two parts directly using MuJoCo, kinematics and dynamics, are discussed. The aerodynamics computation will be closely discussed in the next section.

A. MuJoCo Environment

MuJoCo, which stands for Multi-Joint dynamics with Contact, is a physics engine designed for simulating complex robotic systems and biological organisms in a physically accurate and computationally efficient manner [3]. Developed to facilitate research and development in robotics, biomechanics, and other fields requiring detailed simulation of physical interactions, MuJoCo offers advanced features such as stable and fast simulation of contact dynamics, kinematic chains, and actuator dynamics. Additionaly, its API allows users to build, customize, and extend models programmatically. For all these reasons, MuJoCo has been used as the simulation environment for many other robotics research, especially for reinforcement learning. MuJoCo is selected as the major environment for our simulation to facilitate the modeling process, which also path the way for conducting reinforcement learning in the simulation.

An URDF-type XML file is created to define the model for the usage of MuJoCo. The CAD design files of each mechanical part of the drone are first converted into STL files on Solidworks, which can then be used by MuJoCo. The XML file defines model geometries, kinematic relations (joint positions, dependency, etc.), dynamic properties (inertial properties, damping coefficients, etc.), model features (actuator properties, etc.), and some other simulation features (contact model, solver type, etc.). The full design is depicted on Figure 1.

B. Kinematics and Dynamics

The Aerobat's kinetic sculpture is designed with a complex structure that includes 10 joints and 7 linkages,

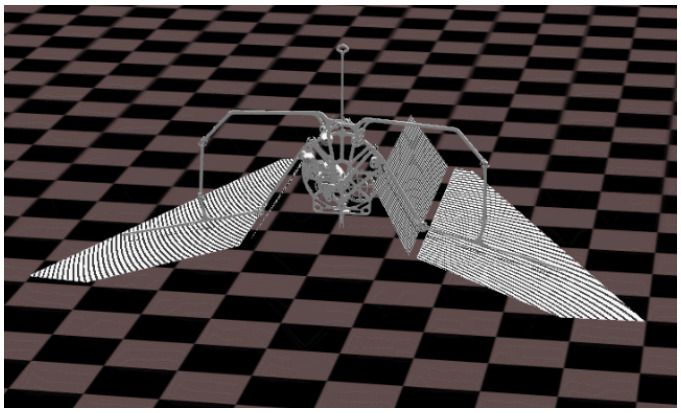


Fig. 1: Flapping wing drone design flying in the MuJoCo environment.

as shown in 2. The first joint functions as pivotal gear connected to motor in driving the linkages that generates the flapping motion. Specifically, joints 5 and 6 serve as the shoulder and elbow joints of the wing, respectively. The wing structures are connected to linkages 3 and 7, which correspond to the proximal and distal segments of the wings, respectively. Note that the kinetic sculpture is a closed kinematic chain, with joint J1, J5, and J8 mounted on the fuselage.

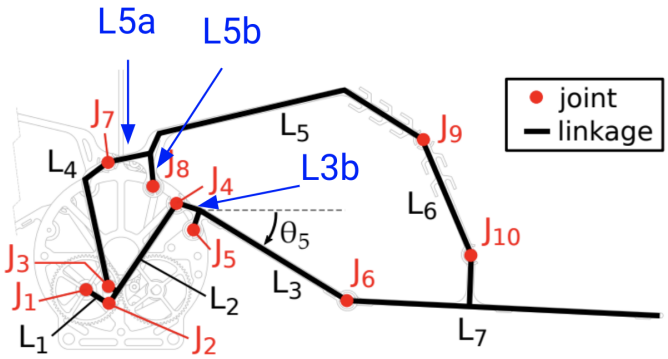


Fig. 2: The kinetic sculpture of the Aerobat.

Initially, all links are directly modeled in the simulation. As MuJoCo does not directly address closed kinematic chain, the model is first defined as 3 open chains on each side and then close the open chains by applying equality constraints. However, this approach leads to instability of simulation as well as violation of constraints at big time step. To improve the precision and stability of the simulation, a significantly small simulation time step $(2e-5s)$ was chosen to allow relatively accurate and stable simulations. However, this leads to a slow simulation speed (slower than the real time), which tends to be not only not ideal and but even impractical for reinforcement learning.

To address this issue, an approach using open kinematic chain was finally implemented. This approach eliminates all links except L3 and L7 and applies the computed values directly to θ_5 and θ_6 to simulate the closed-loop kinematic chain. This approach allows a much larger time step (0.001 second), which enables faster-than-real-world simulation speed and effectuate training process for reinforcement learning.

Dynamics are directly computed by the MuJoCo engine using multi-rigid body dynamics. Note that the external forces such as aerodynamic force (section 2) was applied under each time step to simulate the aerodynamic effect.

III. Aerodynamics Modeling

Flapping wing aerodynamics is very complex to capture due to their unsteady nature. They need to be computed locally before being generalized at the center of mass, as they strongly depend on the local fluid-structure interaction between the wing and the air.
Two different models can be adopted to compute the flapping wing aerodynamics as described in [2]: a quasi-steady model, which approximates the aerodynamics into separate blades of the wing, or an unsteady model which is more accurate but computationally more expensive. Considering the lack of computing resources and the time required to optimize control strategies described in a later section, it is preferable to adopt a hybrid model of quasi-steady and unsteady aerodynamics.

Quasi-steady aerodynamics modeling

The wings surface is divided into discrete blade elements, on which a quasi-steady aerodynamic function is computed locally. Wings are divided each in 2 parts: proximal and distal segments divided each respectively into 2 and 6 blades, which makes in total 16 blades. It is assumed that blade elements are rigid, continuous, and unsegmented so that the lifting line theory is licit. Figure 3 shows the blade elements of the distal wings and its locally computed aerodynamic force.

Given a blade element k, the steady local lift and drag aerodynamic forces are calculated as follows:

$$f_{L,k} = \frac{1}{2}\rho|v_k|^2 C_L(\alpha_k) c_k \Delta s_k \tag{1}$$

$$f_{D,k} = \frac{1}{2}\rho|v_k|^2 C_D(\alpha_k) c_k \Delta s_k \tag{2}$$

$$\mathbf{f_{a,k}} = f_{L,k}\hat{\mathbf{e}}_{\mathbf{L,k}} + f_{D,k}\hat{\mathbf{e}}_{\mathbf{D,k}} \tag{3}$$

,where ρ is the density of air, c_k, Δs_k and α_k are the chord length, span width and angle of attack of the blade element. $\hat{e}_{L,k}$ and $\hat{e}_{D,k}$ are the lift and drag forces directions, respectively parallel and perpendicular to v_k. C_L and C_D are the lift and drag coefficients. Their relation to the angle of attack α was derived experimentally by Dickinson [4], and are given by following equations:

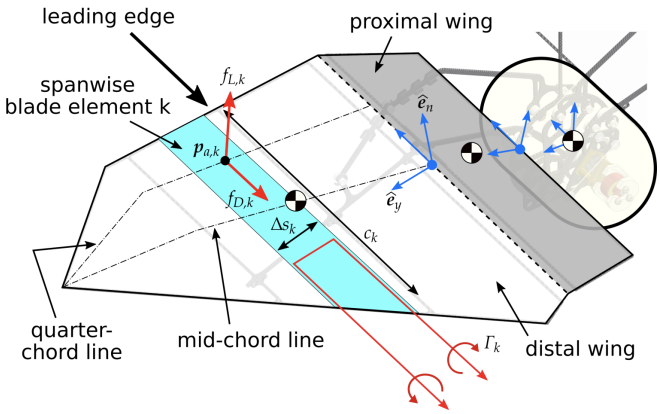


Fig. 3: Distal wing blade elements including its local lift and drag forces, chord lengths and vortex circulation

$$C_L(\alpha) = 0.225 + 1.58\sin(2.13\alpha - 7.2^\circ) \tag{4}$$

$$C_D(\alpha) = 1.92 - 1.55\cos(2.04\alpha - 9.82^\circ) \tag{5}$$

The local forces can be expressed at the center of mass in the body frame using the transformation matrix B_k and following operation:

$$\mathbf{u_{a,k}} = \mathbf{B_k f_{a,k}} \tag{6}$$

Taking the sum of the aerodynamic force for each blade, we finally obtain the generalized aerodynamic force of the flapping wing expressed at the center of mass.

Unsteady aerodynamics using Wagner's function

To address the unsteady nature of the aerodynamics involved in flap-wing flight, the current model integrates Wagner's function into the model. Wagner's function provides a theoretical framework to account for the time-dependent buildup of circulation around the wing after a change in angle of attack, which is particularly relevant for the high angles of attack and rapid maneuvers characteristic of bat flight. The unsteady lift component is modeled using a series expansion of sine functions across the wingspan, representing the circulation distribution. This approach, combined with the Kutta-Joukowski theorem, allows for the calculation of the sectional lift coefficient that varies with time and position along the wingspan.

The approach described in [2] boils down to these equations:

1) **Circulation distribution on the wing:**
The circulation distribution along the wing is represented by the Fourier series, where the circulation $\Gamma(t, y)$ is given by:

$$\Gamma(t, y) = \frac{1}{2} a_0 c_0 U \sum_{n=1}^{m} a_n(t) \sin(n\theta(y))$$

Here, a_0 represents the slope of the angle of attack, c_0 is the chord length at the wing's axis of symmetry, U is the free stream airspeed, and $a_n(t)$ are the Fourier coefficients.

2) **Additional downwash induced by vortices:**
The downwash induced by vortices, $w_y(t, y)$, can be derived from the circulation distribution $\Gamma(t, y)$ and is given by:

$$w_y(t, y) = -\frac{a_0 c_0 U}{4S} \sum_{n=1}^{m} n a_n(t) \sin(n\theta) \sin(\theta)$$

Here, S denotes the total wingspan.

3) **Application of the Kutta-Joukowski theorem: lift coefficient:**
Applying the Kutta-Joukowski theorem, the sectional lift coefficient $C_L(t, y)$ is expressed as:

$$C_L(t, y) = \frac{a_0}{U} \left(\frac{c_0}{c(y)} a_n(t) + \frac{c_0}{U} \dot{a}_n(t) \right) \sin(n\theta)$$

Here, $c(y)$ represents the chord length at position y.

4) **Wagner function for transient response:**
The transient response of the lift coefficient $C_L(t, y)$ to a step change in downwash is captured by the Wagner function $\Phi(t)$, given by:

$$\Phi(t) = 1 - \psi_1 e^{-\epsilon_1 t} - \psi_2 e^{-\epsilon_2 t}$$

The lift coefficient can then be expressed as:

$$C_L(t, y) = \frac{a_0}{U} \Delta w(t, y) \Phi(t)$$

where $\Delta w(t, y)$ represents the change in downwash at position y and time t.

5) **Relation between sectional lift coefficient equations:**
The comprehensive expression for the sectional lift coefficient $C_L(t, y)$ combines the contributions from the Wagner function and the aerodynamic states $z_1(t, y)$ and $z_2(t, y)$. These aerodynamic states reflect the transient effects of downwash on the lift coefficient over time. The combined expression is given by:

$$C_L(t, y) = \frac{a_0}{U} \left(w(t, y)\Phi(0) + z_1(t, y) + z_2(t, y) \right)$$

Here, $z_1(t, y)$ and $z_2(t, y)$ represent the aerodynamic states derived from the integral in the previous equation.

Incorporating Wagner's function into lifting line theory facilitates simulation of lift generation over time. This method models the aerodynamic forces more accurately than quasi-steady assumptions, especially under conditions of rapid wing movement (high frequency) and high angles of attack.

IV. CONTROLLERS

For the controller design, we took an iterative approach. The actual flapping wing design lacks the degrees of freedom to control and track a flight trajectory by itself. It is therefore necessary to modify the design by adding 6 propellers on a cage around the flapping wing body and reduce the burden on the wings for generating lift and maintain hovering. The flapping wing design is depicted on Figure 4.

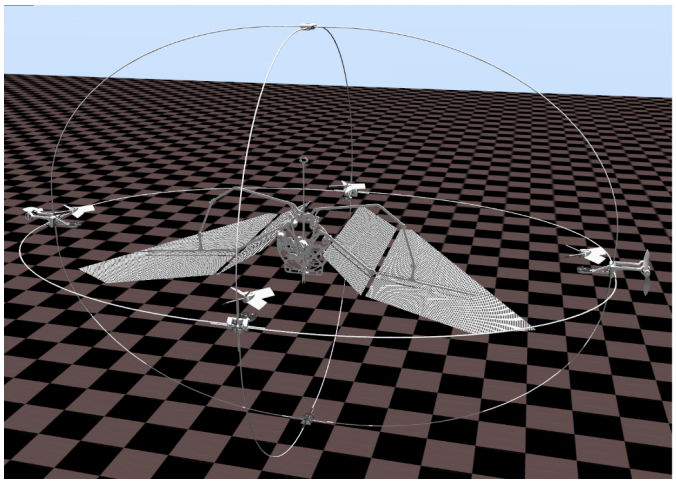


Fig. 4: Flapping wing drone design with 6 propellers flying in the MuJoCo environment.

A. Model-based controllers

We designed two controllers for the setup with propellers and they are - i) PID and ii) Model Predictive Control (MPC). Those model-based controllers will be then useful to discuss the performance of the trained RL policy.

1) PID: A Proportional-Integral-Derivative controller for the flapping wing MAV is developed to control the position (x,y,z) and attitude (roll,pitch,yaw). The general equation of the PID controller is given by equation (7), where $e(t)$ is the error between the reference and the actual value. We used a PID controller for roll, pitch, yaw, x, y and z.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{7}$$

2) MPC: Three MPC controllers are combined together to track the altitude, attitude, and position, respectively. This architecture simplifies the optimization problem and offers overall better performance.

For each MPC controller, traditional 1st order dynamics of a quadcopter are used to predict future states over a fixed horizon. Then, the finite-time optimal control problem is formulated by defining an objective function (8) subject to dynamics, state, and input constraints.

$$\min_{u_i, x_i} (x_N - x_d)^T P (x_N - x_d) + \sum_{i=1}^{N-1} (u_i)^T R u_i + (x_i - x_d)^T Q (x_i - x_d) \tag{8}$$

The above equation represents the general format of the cost function used in the three MPC controllers. The cost is in a quadratic form and includes the stage cost, input cost, and terminal cost subject to the dynamics and other constraints on the state and input constraints.

- **Altitude Controller:** In order to maintain the altitude, the discretized dynamics are shown below.

$$\begin{bmatrix} z(k+1) \\ \dot{z}(k+1) \end{bmatrix} = \begin{bmatrix} z(k) \\ \dot{z}(k) \end{bmatrix} + \begin{bmatrix} \dot{z}(k) \\ \frac{u(k)}{m} - g \end{bmatrix} dt$$

The control variable for this MPC controller is the thrust (u) and constraints are imposed on the state and inputs.

$$\begin{aligned} -\infty \leq z \leq 0 \\ -20 \leq \dot{z} \leq 20 \\ 0 \leq u \leq 15 \end{aligned} \tag{9}$$

- **Attitude Controller:** This controller is composed of three control variables and six state variables and the dynamics for the same are given below.

$$\begin{bmatrix} \phi(k+1) \\ \theta(k+1) \\ \psi(k+1) \\ \dot{\phi}(k+1) \\ \dot{\theta}(k+1) \\ \dot{\psi}(k+1) \end{bmatrix} = \begin{bmatrix} \phi(k) \\ \theta(k) \\ \psi(k) \\ \dot{\phi}(k) \\ \dot{\theta}(k) \\ \dot{\psi}(k) \end{bmatrix} + \begin{bmatrix} \dot{\phi}(k) \\ \dot{\theta}(k) \\ \dot{\psi}(k) \\ \frac{\dot{\theta}\dot{\psi}(I_y - I_z) + \tau_\phi}{I_x} \\ \frac{\dot{\phi}\dot{\psi}(I_z - I_) + \tau_\theta}{I_y} \\ \frac{\dot{\phi}\dot{\theta}(I_x - I_y) + \tau_\psi}{I_z} \end{bmatrix} dt$$

The boundary state conditions for this controller are:

$$\begin{aligned} -\pi/2 \leq \phi \leq \pi/2 \\ -\pi/2 \leq \theta \leq \pi/2 \\ -\infty \leq \psi \leq \infty \\ -\pi/6 \leq \dot{\phi} \leq \pi/6 \\ -\pi/6 \leq \dot{\psi} \leq \pi/6 \\ -\pi/6 \leq \dot{\theta} \leq \pi/6 \end{aligned} \tag{10}$$

- **Position Controller:** To maintain the x and y coordinates of the drone, the following dynamics are used:

$$\begin{bmatrix} x(k+1) \\ y(k+1) \\ \dot{x}(k+1) \\ \dot{y}(k+1) \end{bmatrix} = \begin{bmatrix} x(k) \\ y(k) \\ \dot{x}(k) \\ \dot{y}(k) \end{bmatrix} + \begin{bmatrix} \dot{x}(k) \\ \dot{y}(k) \\ \frac{u * sin(\theta)}{m} \\ \frac{-u * sin(\phi)}{m} \end{bmatrix} dt$$

The input and state boundary conditions are shown below:

$$\begin{aligned} -20 \leq \dot{x} \leq 20 \\ -20 \leq \dot{y} \leq 20 \\ -1 \leq \dot{\phi} \leq 1 \\ -1 \leq \dot{\theta} \leq 1 \end{aligned} \tag{11}$$

The code for MPC is written with the casADi library, being chosen as the fastest framework when resolving the optimization problem.

B. Learning-based controllers

In order to design a controller for the flappy drone without the propellers, learning-based methods are required. In fact, the dynamics of the flapping wings are not known which makes the use of model-based control impossible. By using reinforcement learning, we hope to train a model that will learn the dynamics and be able to fly the drone.

In the training process, we will start with the propellers on and gradually reduce the thrust for the model to be able to adapt. This way, the propellers act as training wheels for the model to learn to fly on its own.

The training algorithm utilizes Proximal Policy Optimization (PPO) inspired by [5]. The PPO algorithm makes multiple gradient updates with different data batches, making it more data-efficient and faster than regular gradient policy methods. However, it can lead to instabilities. To address this, a constraint is added to maximize a surrogate function $L(\theta)$, which is the expected value of the advantage function multiplied by the ratio of probabilities of actions under the new policy to the old policy.

$$L(\theta) = E[r(\theta)A(s,a)]$$

Here, $r(\theta)$ represents the probability ratio.
PPO introduces a clipping mechanism to prevent excessive policy divergence. It clips the probability ratio to be within a range of $[1-\epsilon, 1+\epsilon]$, ensuring the policy update does not deviate too much.

As a result, PPO maximizes a new objective function which takes the minimum between the original and clipped objective functions:

$$L(\theta) = E\left[\min(r(\theta)A(s,a), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)A(s,a))\right]$$

This approach stabilizes the training process by preventing the policy from moving towards new policies with parameters resulting in high probability ratios $r(\theta)$.

The reward function is generally expressed as follows but small modifications are added depending on the trajectory considered (hovering or linear flight). The trained policy is therefore task-specific:

$$\mathrm{R} = w_x e^{-1.5\|\mathbf{s}_x \cdot (\mathbf{x} - \mathbf{x}_{\text{des}})\|} + w_v e^{-0.7\|\mathbf{s}_x \cdot (\mathbf{v} - \mathbf{v}_{\text{des}})\|}$$
$$+ w_\theta e^{-0.9(|\theta_{\text{x}}| + |\theta_{\text{z}}|)} + w_\omega e^{-|\omega_{\text{y}}|} + w_f \tanh\left(\frac{f}{8}\right)$$

where:

- $\mathbf{s}_x = [1,1,1]$ represents the position delta scaling.
- $\mathbf{s}_v = [1,1,4]$ represents the position delta scaling.
- $\mathbf{x}$ and $\mathbf{x}_{\text{des}}$ are the current and desired positions, respectively.
- $\mathbf{v}$ and $\mathbf{v}_{\text{des}}$ represent the current and desired velocities.
- θ_{x} and θ_{z} are the first and third components of the normalized orientation.
- ω_{y} is the y-component of angular velocity.
- f denotes the flapping frequency.
- $w_x, w_v, w_\theta, w_\omega, w_f$ corresponds to weights. position and orientation being the most important ones.

The each term of the reward as well as the reward itself is at maximum equal to 1 for a given timestep. This enables to efficiently compare the performance of the policy between different runs.

The policy can be trained with or without propellers by setting the maximum range of thrust inputs for each propellers. In both cases, timestep is fixed to 0.001 and episodes last a maximum of 10s if no bound conditions are reached before that. Bounds conditions are expressed for wing collisions, orientation and distance to reference position.

RL framework with propellers

When the policy is trained with propellers, the action state space includes 6 inputs for the 6 propellers and 1 input for the flapping frequency.
For the observation history, the 15 last timesteps are chosen and include the position vector (3 linear positions and 4 quaternions), the velocity vector (3 linear velocities and 3 angular velocities), as well as the driving gear angle θ_1.

RL framework without propellers

Removing now the propellers, the actual design lacks the degrees of freedom (DoFs) to control roll and yaw. Feedback driven components are therefore added to the design, which enable to choose the length of specific linkage (L3b, L5a and L5b, see Figure 2). This design modification adds 3 additional inputs for each wing, so 6 additional DoFs for the flapping wing drone, increasing the possibilities to stabilize the flight. Also, a necessary tail is included as the back of the drone which enhance the stability as well.
When the policy is trained without propellers, the action state space includes 6 inputs for the 6 FDC lengths, 1 input for the flapping frequency and 1 input for the tail angle.
For the observation history, the 50 last timesteps are chosen and include the position vector, the velocity vector and the driving gear angle θ_1.

V. Results and Discussion

A. Model-based controllers with propellers

The PID controller is developed to (1) hover and (2) fly a square trajectory. As observed in Figure 5, the controller shows good tracking performance, particularly when hovering.

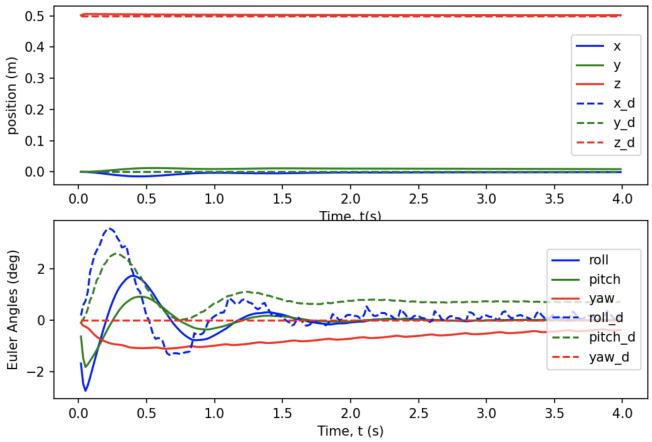


Fig. 5: PID controller performance for hovering

However, when switching to a square trajectory, the drone needs some additional time to adjust its attitude angles in order to follow the trajectory, such as depicted in Figure 6. This therefore motivates the need to develop a stronger, model predictive, controller.

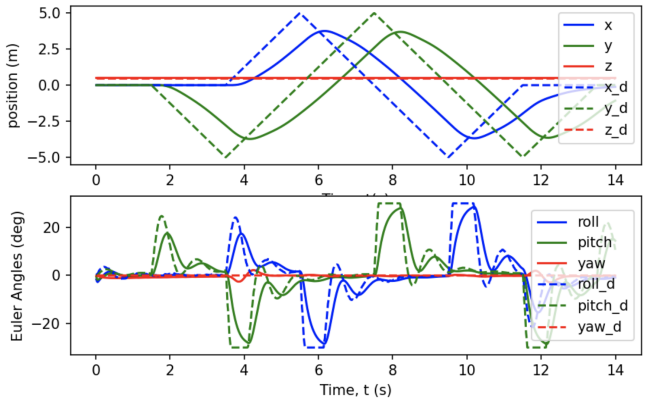
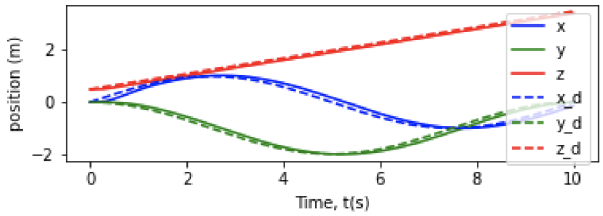


Fig. 6: PID results for a square trajectory

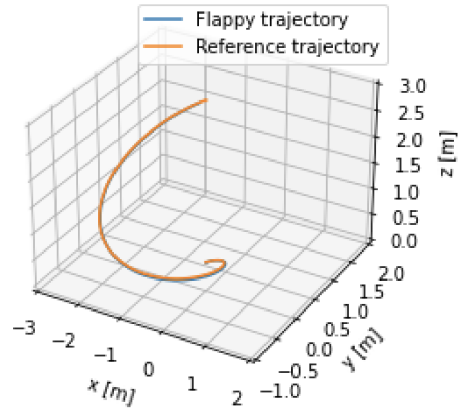
By implementing the 3 MPC controllers as described in the previous section, the drone is able to track any feasible trajectory by using its propellers. Its performance is first tested on a curved trajectory using propellers without any flapping.

The model predictive controller achieves a very strong result where the drone is flying at most 10 cm away from the reference trajectory. When adding flapping, some additional disturbance needs to be controlled but the overall performance stays the same.

The PID controller for hovering and the MPC controllers for flying complex trajectories with propellers achieve therefore remarkable performance. These results constitute a baseline to be reached by the trained RL policy when flying without propellers.

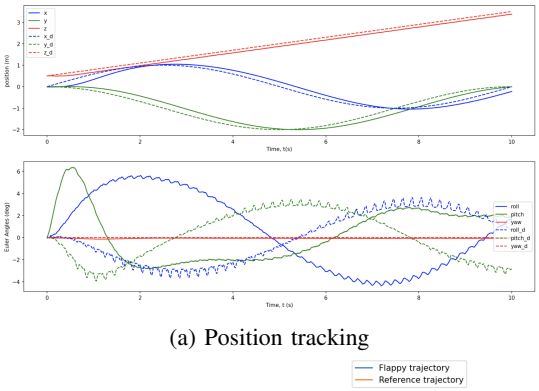


(a) Position tracking

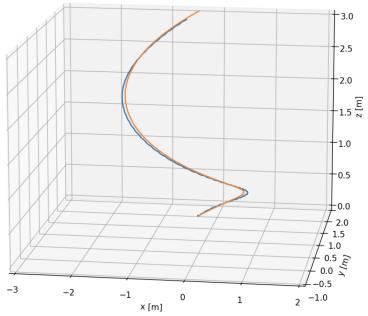


(b) 3D trajectory and reference, which are almost overlapping

Fig. 7: MPC performance on tracking a curved trajectory



(a) Position tracking



(b) 3D trajectory and reference, which little gap

Fig. 8: MPC performance on tracking a curved trajectory with flapping wings

B. *Learning-based controllers with propellers*

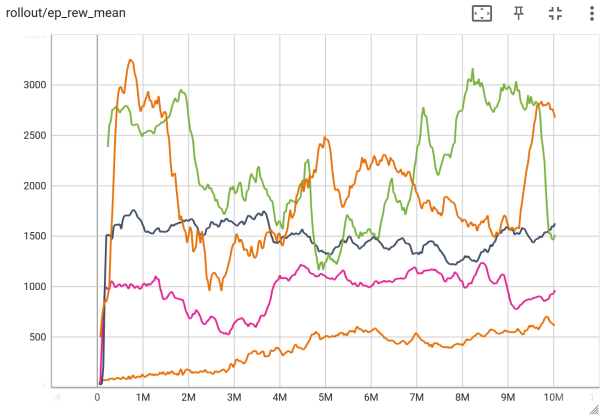
1) Hovering: Considering the weight of the drone, a thrust of 0.35 N is needed from the propellers to sustain the flight. The thrust on the propellers are adjustable control parameters of the model. When training, we set a maximum limit on the value they can take. We began training limiting the thrust to 0.7 N and gradually reduced it every 10M training steps.

When hovering, birds put their body at an angle of 45 degree. Some adjustments were therefore made on the reward function to make the drone adopt this angle. In fact, the results obtained before this change showed that the drone would pitch down and try to hover upside down. That is what led us to make that change.

At each run, the reward function was fined tuned. The desired position was constant and equal to the initial position, and the weights were changed to put the emphasis on the orientation, the angular velocity and the position. Here is a summary of tensorboard logs of the training. We can see the impact of training on top of a previously trained model in the higher value in reward and episode length.

Color	Thrust Available (N)	Max Reward	Max Episode length (s)
Orange	0.7	965	1.79
Pink	0.35	1237	2.23
Blue	0.30	1628 3	2.59
Green	0.20	3168	4.68
Orange 2	0.20	3256 5	4.8

TABLE I: Train History



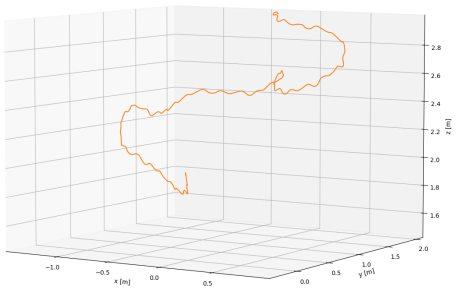
(a) Training History: Reward function

Fig. 9: Training history of successive runs of 10 Million steps

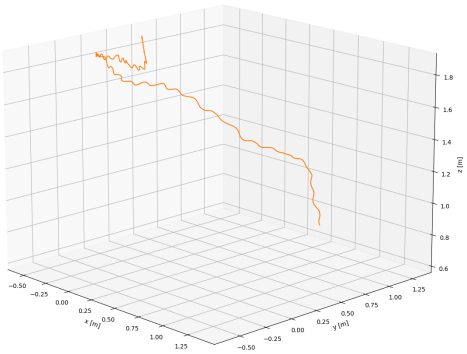
Using the most trained model, the results obtained in the simulation when trying to hover are shown in the plots in Figure 10a,10b and in the Table II

	Thrust Available (N)	Δz from origin (m)	Time (s)
1	0.35	+0.6	1.66
2	0.25	+0.4	1.66
3	0.20	+0.9	5.04
4	0.15	-1.3	3.8
5	0.10	-1.7	2.75
6	0.05	-2.0	1.88

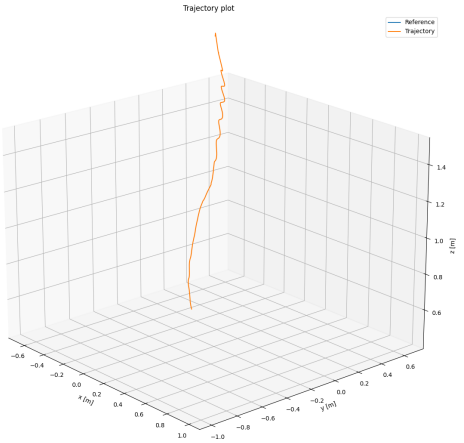
TABLE II: 3D plot Summary of Results



(a) Thrust Available 0.20N, crash: orientation issue, went up. 5.04 seconds



(b) Thrust Available 0.15N, crash: orientation issue, close to ground, went down. 3.80 seconds



(c) Thrust Available 0.05N, crash: touched the ground, went down

Fig. 10: 3D Plots of the trajectory of the drone for different value of thrust available on the latest trained model.

Discussion

What can be observed from these, is that the model is not yet able to fly without propellers. The optimal point for thrust seems to be between 0.2N and 0.15N. The best results were obtained for 0.2N as the drone managed to sustain it's flight for 5 seconds staying within a reasonable distance from the point of origin. When looking at the simulation video, we could observe that in some phase of the flight, the drone manage to fly in the right angle of 45 degree. However, the

drone is not yet stable enough to be able to fly longer. Some perturbation occur that change the orientation of the drone which the controller can't adapt to.

Further training is needed with values for the propeller around 0.2N. The results we have show that it could be possible to achieve good enough results with some amount of thrust available from the propellers. However, the tests made without the propellers show that we are no way near a good outcome to fly without propellers. The drone is clearly falling down, slower than gravity, but it is falling down. A design change might be needed in order to achieve hovering.

2) Flying in a straight line: Considering now a linear trajectory, the propeller thrust is fixed to 0.35N and is not modified during the whole training. The training is run for 50M steps.

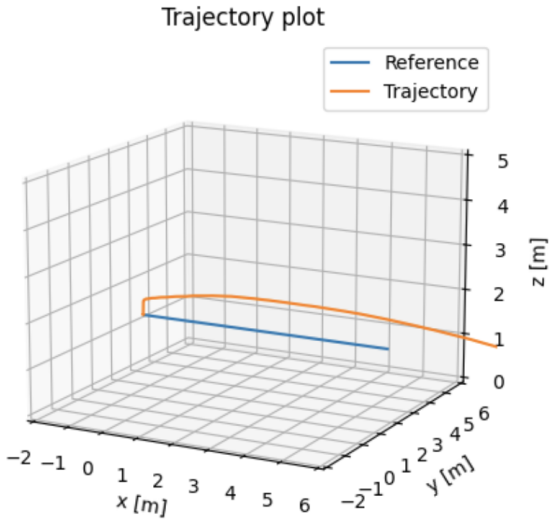


Fig. 11: Flying a RL trained linear trajectory with propellers.

Discussion

Flying a linear trajectory with propellers by using a trained RL policy achieves very satisfying results. The drone flies with a little offset over the reference but it is overall able to track the trajectory. It is therefore easier to train an RL policy to fly a linear trajectory then to hover.

C. Learning-based controllers without propellers

In this final section, the aim is to train directly the RL policy to fly the drone using only its own wings. By implementing the FDCs, it adds degrees of freedom to stabilize roll and yaw, however it still lacks enough degrees of freedom to hover. The RL policy is therefore only trained to fly a linear trajectory of $3m/s$ in the x direction.

To do so, the flapping wing drone is supposed to be flying in air already, removing the complex task of taking off. An initial velocity of $3m/s$ is therefore fixed during the first 100 timesteps (0.1s). The policy is trained for 27M steps before reaching a plateau.

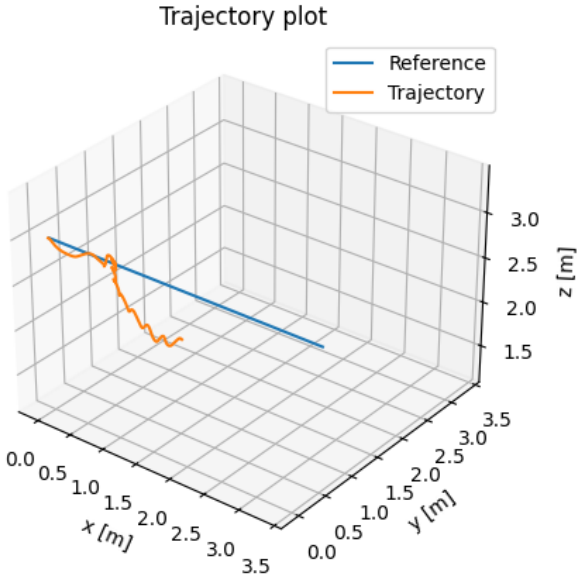


Fig. 12: Flying a RL trained linear trajectory without propellers.

Figure 12 shows the flight trajectory of the flapping wing drone (in orange). The drone flies 8 flapping cycles over 2.5 meters and during 2.6 seconds before hitting a bound condition on orientation. In fact, the drone slowly drifts towards the left and the roll angle diverges. This stability problem happens during each episode, whether it is at the beginning or the end of the training. Even the addition of FDCs did not fix the problem.

looking closer again to the design of the drone, it seems that the left arm is not exactly aligned with the right arm (see Figure 13. The resulting aerodynamic force produced by both wings therefore generate a moment around the x axis, therefore destabilizing the drone.

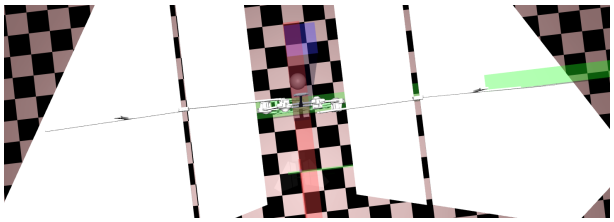


Fig. 13: Zoom on flapping wing drone design.

Although the results obtained with the trained RL policy are far behind model-based controllers performance, those are encouraging results which can be further improved.

VI. Conclusion and Further Improvements

To conclude, this project was conducted with the aim to develop a reinforcement learning based controller to fly a flapping wing drone in a linear trajectory. After building the simulation framework on MuJoCo, model-based controllers

were implemented in order to fly the drone with additional propellers. This gave a baseline to compare later results obtained by the training of a PPO policy. Reinforcement learning based controllers achieved overall worse performance than model based controllers but are encouraging. The flapping wing drone was able to fly about 3 meters by fully relying on its own wings.
To further achieve sustainable flight, a change in the flapping wing drone design would be required. First to make sure the aerodynamic force does not create a moment around the x axis, and second to assure that the center of mass is aligned with the x-axis and in a stable position.

ACKNOWLEDGMENT

Jack worked on the Mujoco environment and design of the drone. He designed the MPC controller and focused on flying in straight line for the RL controller.

Nicolas worked on the aerodynamic modeling. He desgined the PID controller and focused on hovering for the RL controller.

REFERENCES

[1] E. Sihite and A. Ramezani, "Enforcing nonholonomic constraints in aerobat, a roosting flapping wing model," in 2020 59th IEEE Conference on Decision and Control (CDC), pp. 5321–5327, IEEE, 2020.
[2] E. Sihite, P. Ghanem, A. Salagame, and A. Ramezani, "Unsteady aerodynamic modeling of aerobat using lifting line theory and wagner's function," arXiv preprint arXiv:2207.12353, 7 2022.
[3] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 5026–5033, 2012.
[4] M. H. Dickinson, F.-O. Lehmann, and S. P. Sane, "Wing rotation and the aerodynamic basis of insect flight," Science, vol. 284, no. 5422, pp. 1954–1960, 1999.
[5] F.I. Hsiao, C.M. Chiang, A. Hou, "Reinforcement Learning Based Quadcopter Controller", 2019.