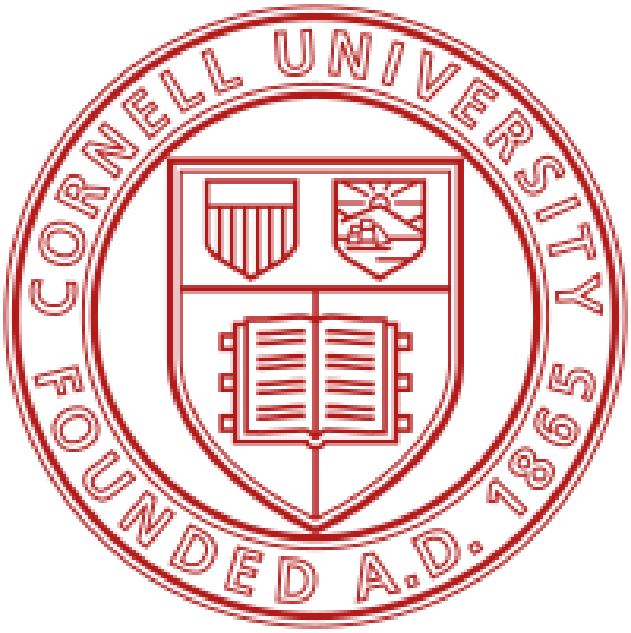
Reinforcement Learning for Determining Wing Kinematics for Hovering Trajectory Following

Albert Addo

December 2025



1 Abstract

Flapping-wing flight is governed by highly nonlinear, unsteady aerodynamic interactions that make analytical modeling and control challenging. This project develops a reduced-order dynamic model of a flapping system with an ellipsoidal body and rigid wings capable of symmetric flapping and pitching. Aerodynamic forces are derived using thin-body approximations inspired by Andersen–Pesavento–Wang models and evaluated numerically through spanwise integration. These forces are coupled with inertial effects to form equations of motion that are integrated in real time within a reinforcement learning environment.

A Proximal Policy Optimization (PPO) agent is trained to adjust wing kinematics—parameterized by amplitudes, phase offsets, and wingbeat frequency—to reduce deviation from a target trajectory. Smooth transitions between actions are enforced to maintain kinematic continuity. Results show that PPO can learn structured and physically plausible flapping motions and improve trajectory performance over training, although stable hovering was not achieved. This work demonstrates the potential of reinforcement learning for studying unsteady aerodynamics and bio-inspired flight while highlighting limitations in reward design and policy structure.

2 Acknowledgments

I would like to express my sincere gratitude to Professor Jane Wang for serving as my advisor on this project and for her guidance and support throughout its development. I would also like to thank Professor Rajesh Bhaskaran for granting access to the computational resources in the Swanson Simulation Laboratory, which was helpful at times for conducting the reinforcement learning simulations. Additionally, I am grateful to Matt Ulinski and Judy Thoroughman for their assistance with project logistics and for their continued support and guidance throughout the duration of this work and my M. Eng.

Contents

3 Introduction

Flapping-wing flight remains one of the most challenging regimes in aerodynamics and vehicle dynamics due to its inherently unsteady, nonlinear, and tightly coupled nature. Unlike conventional fixed-wing or rotary-wing aircraft, flapping-wing systems rely on time-varying wing kinematics to generate lift and thrust simultaneously, often operating in low-Reynolds-number regimes where viscous effects and unsteady flow phenomena dominate. These characteristics make both analytical modeling and control design for flapping-wing vehicles particularly difficult, motivating the development of simplified models and data-driven approaches to better understand the relationship between wing motion and resulting vehicle dynamics.

This project addresses the problem of determining wing kinematics capable of producing a desired translational motion of a flapping-wing body. Rather than focusing on biological fidelity or high-fidelity flow simulation, the emphasis is placed on constructing a reduced-order dynamic model that captures the essential coupling between wing motion and body translation, while remaining computationally tractable for control optimization. Such an approach is particularly well-suited for reinforcement learning (RL), where repeated simulation rollouts are required to explore the control space efficiently.

The system considered consists of a rigid body modeled as an ellipsoid, representing the main body, with two identical wings rigidly attached at fixed hinge points. The wings execute symmetric flapping motions, each characterized by a flapping angle $\phi(t)$ and a pitching angle $\psi(t)$. No spanwise flexibility or multiple articulation points are considered; instead, each wing is constrained to rotate about a single attachment location. To further reduce model complexity, the orientation of the body is assumed to be fixed, and rotational dynamics of the body are neglected. The analysis therefore focuses exclusively on the translational motion of the body's center of mass for a prescribed, constant body orientation θ.

An equation of motion governing the translational dynamics of the system is derived based on the specified kinematics and force models. Aerodynamic forces generated by the wings are functions of the instantaneous flapping and pitching angles, and these forces act on the body to produce acceleration of the center of mass. Given this formulation, the control problem can be posed as follows: **for a desired trajectory of the body's center of mass, determine the time-varying wing kinematics $\phi(t)$ and $\psi(t)$ that cause the system to follow that trajectory.** This paper focuses only on following a hover trajectory.

To solve this problem, reinforcement learning is employed as a model-based control strategy. The desired trajectory is discretized in time and represented as a sequence of target states. The RL agent interacts with the simulated environment by selecting wing kinematics at each time step, observes the resulting state of the system, and receives a reward based on how closely the actual motion matches the prescribed trajectory. Through repeated training episodes, the agent learns a control policy that maps system states to wing motions capable of producing the desired translational behavior.

Beyond this specific system, the approach explored in this work has broader relevance to robotics and bio-inspired flight. Reinforcement learning has become a powerful tool in robotic control, particularly for systems with complex dynamics, high-dimensional control spaces, and limited analytical solutions. Flapping-wing vehicles fall squarely within this class of problems: the relationship between actuator motion and resulting forces is highly nonlinear and strongly dependent on timing, phase, and coupling between degrees of freedom. RL offers a framework for discovering effective control strategies without requiring explicit inversion of the system dynamics or hand-designed control laws.

From a bio-inspired perspective, birds and insects achieve remarkable agility, efficiency, and robustness through coordinated wing motions that remain difficult to replicate using traditional engineering approaches. Simplified models such as the one developed here provide a testbed for investigating how particular patterns of wing kinematics give rise to specific translational behaviors. By framing trajectory generation as a learning problem, this work contributes toward the long-term goal of replicating aspects of biological flight in engineered systems, including small-scale aerial robots capable of maneuvering in cluttered environments, performing precise hovering, or adapting to uncertain aerodynamic conditions. While the present model neglects many features of real biological flyers, it represents a step toward integrating physics-based modeling with learning-based control in the study of flapping-wing flight.

4 Background & Literature Review

4.1 Andersen, Pesavento & Wang: Thin-Body Models and Unsteady Aerodynamics in Fluttering and Tumbling Systems

The paired studies by Andersen, Pesavento, and Wang (2005) provide a comprehensive exploration of the unsteady aerodynamics governing simple lifting surfaces such as falling cards and fluttering plates. Despite their geometric simplicity, these systems exhibit rich dynamical behavior driven by strong coupling between rotation, translation, and vortex shedding. Across both papers, the authors show that oscillating or rotating thin bodies immersed in a fluid generate aerodynamic forces dominated by unsteady mechanisms that cannot be captured by steady-state lift or drag models.

A central theme in both works is the use of a thin-body approximation—idealizing each structure as a plate or thin elliptical cross section. This simplification enables the derivation of compact expressions for added-mass forces, unsteady pressure loads, and aerodynamic moments that depend on local translational and rotational velocities. In the study of fluttering and tumbling plates, this thin-body formulation reveals how relationships between rotation and translation govern transitions between stable fluttering motions, autorotating tumbling, and other aerodynamic regimes. Similarly, in the falling-card study, the authors show that varying non-dimensional parameters such as the moment of inertia or thickness-to-width ratio produces bifurcations between periodic flutter, continuous tumbling, and steady descent. In both cases, small variations in geometry, inertia, or timing lead to large qualitative changes in aerodynamic response.

These findings are directly relevant to the aerodynamic modeling of the flapping-wing system developed in the present work. Here, each wing is approximated as a sequence of thin elliptical cross sections along the span, and the local aerodynamic forces are computed using principles inspired by the thin-body formulations presented in Andersen et al. The forces at each spanwise station depend on the instantaneous translational and rotational components induced by the wing's flapping angle $\phi(t)$ and pitching angle $\psi(t)$, after which they are integrated to obtain the net load on the ellipsoidal body. This modeling approach mirrors the philosophy of the plate and falling-card studies: reduce geometric complexity while retaining the essential unsteady aerodynamic mechanisms that arise from vortex formation and rotation–translation coupling.

Furthermore, the strong sensitivity observed in fluttering and tumbling systems underscores why analytic control design is difficult for flapping-wing flight. Andersen et al. demonstrate that transitions between aerodynamic regimes occur through bifurcations driven by subtle changes in timing or system parameters. In the present study, reinforcement learning (RL) is used to navigate this nonlinear landscape by discovering wing-kinematic patterns that produce desired trajectories of the center of mass. RL serves as an adaptive control method capable of coping with the same kinds of sensitivity and nonlinear vortex-driven behaviors that make fluttering and tumbling systems challenging to model analytically.

In summary, the Andersen–Pesavento–Wang studies provide both the conceptual foundation and the aerodynamic modeling framework for the simplified flapping-wing model used in this project. Their thin-body force formulations motivate the spanwise sectional approach employed here, while their analyses of unsteady transitions highlight the necessity of data-driven methods such as RL for controlling systems governed by nonlinear, vortex-dominated aerodynamics.

4.2 Review of Wang (2005): *Dissecting Insect Flight*

Wang's 2005 review, *Dissecting Insect Flight*, provides a foundational perspective on the aerodynamic mechanisms that enable insects to generate efficient, controllable flight despite operating in highly unsteady, low Reynolds number regimes. The paper emphasizes that insect flight is governed not by steady aerodynamic principles but by vortex-dominated, time-dependent interactions between the wings and surrounding fluid. Mechanisms such as leading-edge vortex (LEV) attachment, wake capture, clap-and-fling, and unsteady lift enhancement are identified as central to understanding how insects generate forces far exceeding those predicted by classical steady-state airfoil theory.

A major contribution of Wang's work is the synthesis of experimental, computational, and theoretical studies to illustrate how wing kinematics—particularly wing rotation, pitching, and reversing—directly shape aerodynamic force production. The paper argues that the "wing is the engine," meaning that control authority in flapping flight arises almost entirely from the ability to prescribe time-varying kinematics rather

than from changes in body attitude or external control surfaces. Additionally, Wang highlights the importance of simplified models that retain key physical mechanisms while enabling tractable analysis of flapping dynamics, noting that such models are essential for linking observed biological motions to force production and flight behavior.

The relevance of this work to the this project lies in its articulation of the tight coupling between wing kinematics and translational body motion. Although the current study adopts a simplified dynamic model—an ellipsoidal body with two rigid wings executing symmetric flapping ($\phi(t)$) and pitching ($\psi(t)$) motions while body orientation remains fixed—the underlying goal aligns with the conceptual framework introduced by Wang. Specifically, this project seeks to determine how prescribed wing kinematics generate forces that shape the motion of the center of mass, echoing Wang's emphasis on the central role of unsteady kinematics in governing insect flight.

Furthermore, Wang's review underscores the complexity and high dimensionality of the control problem inherent to biological flapping systems, suggesting a natural connection to modern data-driven control methods such as reinforcement learning (RL). Insects achieve flight control through subtle modulation of wing phase, amplitude, and pitch—precisely the types of control inputs explored in this study. By using RL to discover wing-kinematic patterns that produce target trajectories, the present work can be viewed as a computational extension of the ideas introduced in Wang's paper: leveraging simplified physical models to investigate how variations in flapping and pitching motions map to aerodynamic forces and, ultimately, to controlled flight.

In summary, Wang (2005) provides both the biological motivation and the physical justification for the modeling and control approach explored here. While the current model does not attempt to replicate the full richness of insect aerodynamics, it is directly inspired by the principle that unsteady wing motions are the primary driver of flight forces. This connection situates the present RL-based trajectory generation framework within the broader body of research seeking to understand and replicate the remarkable maneuverability of biological flyers.

5 Theory

5.1 System

Figure 1: A simple CAD model of the flapping system for demonstration.

5.1.1 Definition of Frames, Points, and Coordinate Systems

As shown in **Figure 2**, I defined a total of 3 reference frames. The first reference frame is the inertial frame, $\mathcal{I} \equiv (O, \hat{e}_x, \hat{e}_y, \hat{e}_z)$. **Point O** is a fixed arbitrary point in the inertial frame, which, in this case, is the lab frame of reference.

The next reference frame, $\mathcal{B} \equiv (B, \hat{b}_1, \hat{b}_2, \hat{b}_3)$, is fixed to the body. This frame, under our assumptions and simplifications, is allowed to rotate only in the xy-plane. **Point B** marks the center of mass of the body. Drawn above **Point B** in figure 1, I have defined **Point G**, which represents the overall center of mass of the system, but it is drawn at an arbitrary position in-between the 3 individual center of masses (body and wings).

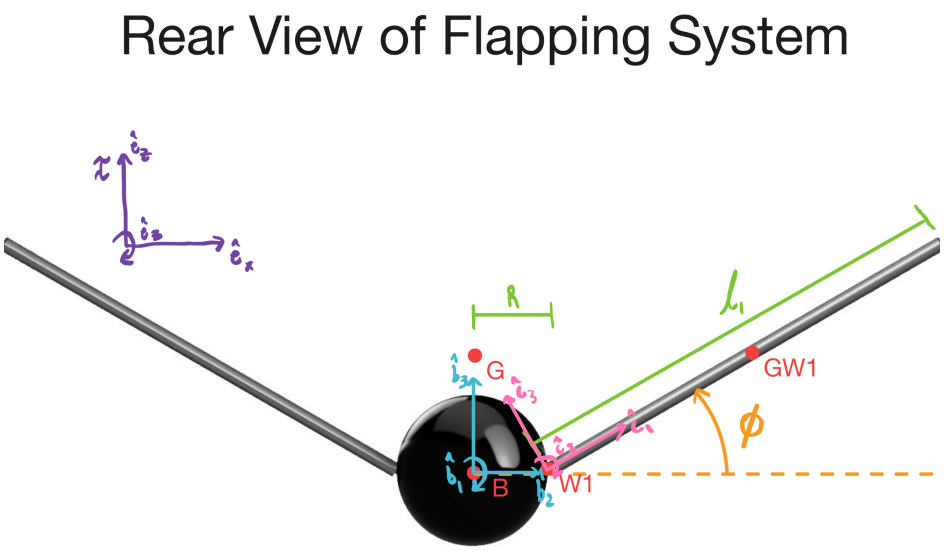


Figure 2: Rear view of flapping system showcasing the $\mathcal{I}, \mathcal{B}, \mathcal{C}$ frames and relevant dimensions.

The last reference frame of reference shown in **Figure 2** is the intermediate frame: $\mathcal{C} \equiv (W_1, \hat{c}_1, \hat{c}_2, \hat{c}_3)$ for the right wing (which this report refers to as "Wing 1"). **Point $\mathbf{W_1}$** is located at the joint of wing 1 and the body. $\hat{\mathbf{c}}_\mathbf{1}$ is always aligned with the length of the wing, and $\hat{\mathbf{c}}_\mathbf{3}$ and $\hat{\mathbf{c}}_\mathbf{1}$ are always in the xz-plane. ϕ is defined as positive counterclockwise from horizontal and is the wing beat angle. The $\mathcal{C}$ frame acts as intermediary frame to the wing-fixed frame, $\mathcal{D}$, defined in **Figure 3** below.

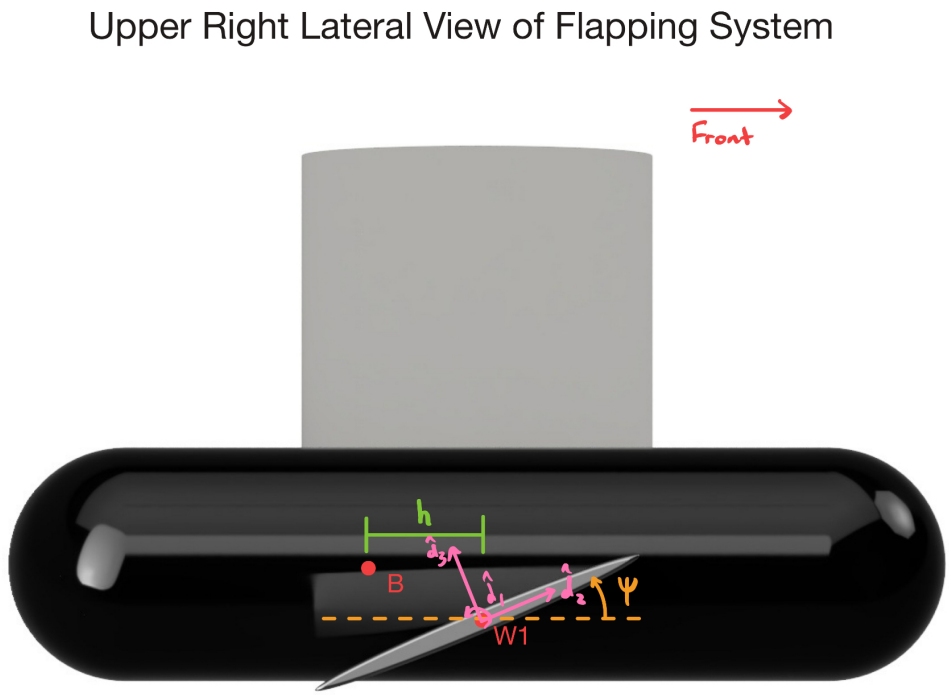


Figure 3: Upper-right lateral view of flapping system showcasing the $\mathcal{D}$ frame and h.

I defined frame $\mathcal{D}$, to be a wing-fixed frame. I defined the angle ψ to measure the pitch of the wing relative to the alignment of the body frame. l_1 is the outward length of the wing and l_2 is the side length of the wing.

5.1.2 Direction Cosine Matrices

To make the vector math simpler quicker, I went ahead and defined Direction Cosine Matrices (DCM's) for switching between the rotational frames and the inertial frames. The DCM used to switch from the inertial frame to the $\mathcal{B}$ frame is:

$$^{\mathcal{B}}\underline{\underline{N}}^{\mathcal{I}} = \begin{bmatrix} \hat{b}_1 \cdot \hat{e}_x & \hat{b}_1 \cdot \hat{e}_y & \hat{b}_1 \cdot \hat{e}_z \\ \hat{b}_2 \cdot \hat{e}_x & \hat{b}_2 \cdot \hat{e}_y & \hat{b}_2 \cdot \hat{e}_z \\ \hat{b}_3 \cdot \hat{e}_x & \hat{b}_3 \cdot \hat{e}_y & \hat{b}_3 \cdot \hat{e}_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{1}$$

Similarly, the DCM used to switch from the $\mathcal{B}$ frame to the intermediary wing frame, $\mathcal{C}$, is:

$$^{\mathcal{C}}\underline{\underline{N}}^{\mathcal{B}} = \begin{bmatrix} \hat{c}_1 \cdot \hat{b}_1 & \hat{c}_1 \cdot \hat{b}_2 & \hat{c}_1 \cdot \hat{b}_3 \\ \hat{c}_2 \cdot \hat{b}_1 & \hat{c}_2 \cdot \hat{b}_2 & \hat{c}_2 \cdot \hat{b}_3 \\ \hat{c}_3 \cdot \hat{b}_1 & \hat{c}_3 \cdot \hat{b}_2 & \hat{c}_3 \cdot \hat{b}_3 \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix} \tag{2}$$

From the $\mathcal{C}$ frame to the $\mathcal{D}$ frame is:

$$^{\mathcal{D}}\underline{\underline{N}}^{\mathcal{C}} = \begin{bmatrix} \hat{d}_1 \cdot \hat{c}_1 & \hat{d}_1 \cdot \hat{c}_2 & \hat{d}_1 \cdot \hat{c}_3 \\ \hat{d}_2 \cdot \hat{c}_1 & \hat{d}_2 \cdot \hat{c}_2 & \hat{d}_2 \cdot \hat{c}_3 \\ \hat{d}_3 \cdot \hat{c}_1 & \hat{d}_3 \cdot \hat{c}_2 & \hat{d}_3 \cdot \hat{c}_3 \end{bmatrix} = \begin{bmatrix} 0 & \cos\psi & \sin\psi \\ 1 & 0 & 0 \\ 0 & -\sin\psi & \cos\psi \end{bmatrix} \tag{3}$$

The inverses of the inertial to rotational DCM's are simply the transposes of the above matrices. This will give you DCM's to transform from the rotational frames to the inertial frame:

$$^{\mathcal{I}}\underline{\underline{N}}^{\mathcal{C}} = [^{\mathcal{C}}\underline{\underline{N}}^{\mathcal{I}}]^T \tag{4}$$

$$^{\mathcal{I}}\underline{\underline{N}}^{\mathcal{D}} = [^{\mathcal{D}}\underline{\underline{N}}^{\mathcal{I}}]^T \tag{5}$$

5.2 Deriving the Equation of Motion for the Flapping System

5.2.1 Assumptions and Simplifications

The following assumptions are made for the analysis:

- The wing is approximated as an extrusion of a thin ellipse structure.
- There is no wind.
- The body has radius R.
- Symmetric Wing Beating (ϕ_1=ϕ_2=ϕ).
- When $\phi = 0$, the wings are aligned with the center of the body.

5.2.2 Free-Body Diagram

I identified 3 separate forces acting on the flapping system, the overall weight force and aerodynamic forces from each individual wing. $\vec{F}_g$ acts at **point G** and will always be in the $-\hat{e}_z$ direction. I have drawn the aerodynamic forces as $\int_0^{l_1} \vec{f}_{aero}(s)ds$. **Despite the forces being drawn as though they are orthogonal to their respective wings, they can be drawn in any arbitrary direction.**

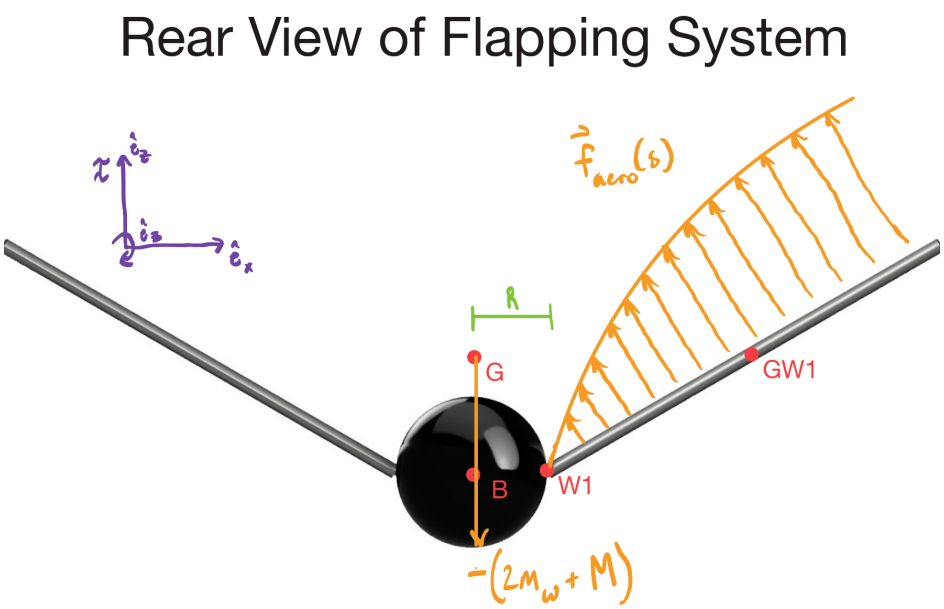


Figure 4: Free-body diagram showing the combined weight force and the aerodynamic wing force distribution.

5.2.3 Newton's Second Law

Newton's 2nd law tells that the net force acting on a system of bodies is equal to the total mass of that system times the acceleration of the system's center of mass. My free-body diagram shows that there are only 3 total forces acting on the system: gravity and the aerodynamic forces felt by each wing. Let $\vec{r}_G \equiv < x_G(t), x_G(t), z_G(t) >$ be a vector representing the position of the system's center of mass relative to some arbitrary **point O** fixed to the lab frame, and let $\vec{r} \equiv < x(t), y(t), z(t) >$ be a vector representing the position of the body at **point B** relative to **point O**. Applying Newton's 2nd Law, we get:

$$(2m_w + M) \cdot^{\mathcal{I}} \vec{a}_G = -(2m_w + M) \cdot g \cdot \hat{e}_z + \int_0^{\ell_1} \vec{f}_{aero,1}(s)ds + \int_0^{\ell_1} \vec{f}_{aero,2}(s)ds \tag{6}$$

where ${}^{\mathcal{I}}\vec{a}_G$ is the acceleration of $\vec{r}_G$ in the inertial frame. However, because we are only concerned about the vertical and horizontal motion of the body, we only need to analyze the $\hat{e}_y$ and $\hat{e}_z$ components of this equation, giving us for the z-direction:

$$\left[(2m_w + M) \cdot^{\mathcal{I}} \vec{a}_G = -(2m_w + M) \cdot g \cdot \hat{e}_z + \int_0^{\ell_1} \vec{f}_{aero,1}(s)ds + \int_0^{\ell_1} \vec{f}_{aero,2}(s)ds\right] \cdot \hat{e}_z \tag{7}$$

$$(2m_w + M) \cdot \ddot{z}_G = -(2m_w + M) \cdot g + \int_0^{\ell_1} \vec{f}_{aero,1}(s)ds \cdot \hat{e}_z + \int_0^{\ell_1} \vec{f}_{aero,2}(s)ds \cdot \hat{e}_z \tag{8}$$

Utilizing symmetry, we get...

$$(2m_w + M) \cdot \ddot{z}_G = -(2m_w + M) \cdot g + 2 \cdot \left[\int_0^{\ell_1} \vec{f}_{aero,1}(s)ds \cdot \hat{e}_z\right] \tag{9}$$

And for the y-direction:

$$[(2m_w + M) \cdot^{\mathcal{I}} \vec{a}_G = -(2m_w + M) \cdot g \cdot \hat{e}_z + \int_0^{\ell_1} \vec{f}_{aero,1}(s)ds + \int_0^{\ell_1} \vec{f}_{aero,2}(s)ds] \cdot \hat{e}_y \tag{10}$$

$$(2m_w + M) \cdot \ddot{y}_G = \int_0^{\ell_1} \vec{f}_{aero,1}(s)ds \cdot \hat{e}_y + \int_0^{\ell_1} \vec{f}_{aero,2}(s)ds \cdot \hat{e}_y \tag{11}$$

Utilizing symmetry, we get...

$$(2m_w + M) \cdot \ddot{y}_G = 2 \cdot \left[\int_0^{\ell_1} \vec{f}_{aero,1}(s)ds \cdot \hat{e}_y \right] \tag{12}$$

Now, to write these equations of motion in terms of z(t) and y(t), we must look at the definition of the center of mass.

5.2.4 $\ddot{z}_G(t)$ in Terms of $\ddot{z}(t)$

Representing the vertical positions of the centers of mass of wings 1 and 2 as $Z_w(t)$, the z-position of the center of mass is defined as:

$$z_G(t) = \frac{M \cdot z(t) + 2m \cdot z_{w1}(t)}{2m + M} \tag{13}$$

Because each wing individually only has two degrees of freedom as a result of their pinned constraints with the body of the flapping system, we can apply the following constraint equations to write their positions in terms of z(t):

$$z_w(t) = z_{w/W_1}(t) + z_{W_1/G'}(t) + z_{G'/O}(t) = \frac{\ell_1}{2} \cdot \sin(\phi) + h \cdot \sin(\theta) + z(t) \tag{14}$$

This turns the equation to:

$$z_G(t) = \frac{M \cdot z(t) + 2m \cdot \left(\frac{\ell_1}{2} \cdot \sin(\phi) + h \cdot \sin(\theta) + z(t)\right)}{2m + M} \tag{15}$$

In order to apply this to the equation of motion, the second time derivative must be found:

$$\frac{d}{dt} \cdot \left[z_G(t) = \frac{M \cdot z(t) + 2m \cdot \left(\frac{\ell_1}{2} \cdot \sin(\phi) + h \cdot \sin(\theta) + z(t)\right)}{2m + M} \right] \tag{16}$$

$$\dot{z}_G(t) = \dot{z}(t) + \frac{2m}{2m + M} \left(\tfrac{\ell_1}{2} \cos\phi\, \dot{\phi} + h \cos\theta\, \dot{\theta} \right) \tag{17}$$

$$\ddot{z}_G(t) = \frac{d}{dt} \cdot [\dot{z}_G(t)] = \ddot{z}(t) + \frac{2m}{2m + M} \left(\tfrac{\ell_1}{2} \left(-\sin\phi\, \dot{\phi}^2 + \cos\phi\, \ddot{\phi} \right) + h \left(-\sin\theta\, \dot{\theta}^2 + \cos\theta\, \ddot{\theta} \right) \right) \tag{18}$$

Plugging this back into equation 9, we get:

$$\ddot{z}(t) = -\frac{2m}{2m + M} \left(\frac{l}{2} \left(-\sin\phi\, \dot{\phi}^2 + \cos\phi\, \ddot{\phi} \right) + h \left(-\sin\theta\, \dot{\theta}^2 + \cos\theta\, \ddot{\theta} \right) \right) - g + \frac{2 \cdot \left[\int_0^{\ell_1} \vec{f}_{aero,1}(s)ds \cdot \hat{e}_z \right]}{2m + M} \tag{19}$$

That is the equation of motion without defining the aerodynamic forces just yet. Note that the first term of equation 18, $-\frac{2m}{2m+M} \left(\frac{l}{2} \left(-\sin\phi\, \dot{\phi}^2 + \cos\phi\, \ddot{\phi} \right) + h \left(-\sin\theta\, \dot{\theta}^2 + \cos\theta\, \ddot{\theta} \right) \right)$, represents the acceleration of the center of mass of the body due solely to the movement of the wings per conservation of linear momentum.

5.2.5 $\ddot{y}_G(t)$ in Terms of $\ddot{y}(t)$

Representing the y-positions of the centers of mass of wings 1 and 2 as $y_w(t)$, the y-position of the center of mass is defined as:

$$y_G(t) = \frac{M \cdot y(t) + 2m \cdot y_{w1}(t)}{2m + M} \tag{20}$$

Because each wing individually only has two degrees of freedom as a result of their pinned constraints with the body of the flapping system, we can apply the following constraint equations to write their positions in terms of y(t):

$$y_w(t) = y_{w/W_1}(t) + y_{W_1/G'}(t) + y_{G'/O}(t) = h \cdot \cos(\theta) + y(t) \tag{21}$$

This results in an equation for the y position of the systems center of mass as:

$$y_G(t) = \frac{M \cdot y(t) + 2m \cdot (h \cdot \cos(\theta) + y(t))}{2m + M} \tag{22}$$

Following the same process as done with z:

$$\dot{y}_G(t) = \frac{M\dot{y}(t) + 2m\left(\dot{y}(t) - h\dot{\theta}\sin(\theta)\right)}{2m + M} \tag{23}$$

$$\ddot{y}_G(t) = \ddot{y}(t) + \frac{2m\left(-h\ddot{\theta}\sin(\theta) - h\dot{\theta}^2\cos(\theta)\right)}{2m + M} \tag{24}$$

Plugging back into equation 12 gives us:

$$\ddot{y}_G = -\frac{2m\left(-h\ddot{\theta}\sin(\theta) - h\dot{\theta}^2\cos(\theta)\right)}{2m + M} + \frac{2}{2m + M}\left[\int_0^{\ell_1} \vec{f}_{aero,1}(s)ds \cdot \hat{e}_y\right] \tag{25}$$

5.2.6 The Aerodynamic Forces Using Blade Element Method

In this model, I use blade element method in order to determine the aerodynamic force on the system. I first begin by analyzing the force contribution from an infinitesimally wide element of the wing with width ds. Viewing the cross-section of the element below gives us the following relationship with the forces:

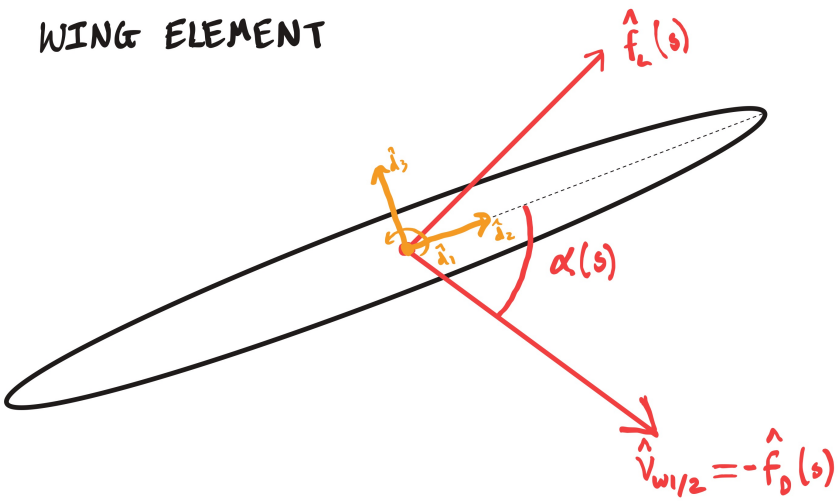


Figure 5: A diagram showing the drag and lift forces with respect to the velocity of the wing element and the wing's geometry.

In order to find the force per unit length that each element along the blade contributes, I first found ${}^{\mathcal{I}}\vec{v}_{dw/O}(s)$, the velocity of the wing elements for both relative to the air as a function of the distance along the wing from the flapping system's body. "s" denotes the distance of an element. We can find ${}^{\mathcal{I}}\vec{v}_{dw/O}(s)$ through the following steps:

First, note that the position of the the center of each blade element, dw, can be represented as a function of s, the distance along the wing from the body.

$$\vec{r}_{dw/O}(s) = \vec{r}_{dw/W1}(s) + \vec{r}_{W1/G'} + \vec{r}_{G'/O} = s\hat{c}_1 + h\hat{b}_2 + R\hat{b}_1 + y\hat{e}_y + z\hat{e}_z \tag{26}$$

In order to take the inertial time derivative of the position we must define the angular velocity vector of the wing's frame, $\mathcal{D}$, relative to the inertial frame, ${}^{\mathcal{I}}\vec{\omega}^{\mathcal{D}}$, as well as the angular velocity of the body's fram with respect to the inertial frame, ${}^{\mathcal{I}}\vec{\omega}^{\mathcal{B}}$, and also ${}^{\mathcal{I}}\vec{\omega}^{\mathcal{C}}$:

$$ {}^{\mathcal{I}}\vec{\omega}^{\mathcal{D}} = \dot{\theta}\hat{b}_1 - \dot{\phi}\hat{c}_2 + \dot{\psi}\hat{d}_1 \tag{27} $$

$$ {}^{\mathcal{I}}\vec{\omega}^{\mathcal{C}} = \dot{\theta}\hat{b}_1 - \dot{\phi}\hat{c}_2 \tag{28} $$

$$ {}^{\mathcal{I}}\vec{\omega}^{\mathcal{B}} = \dot{\theta}\hat{b}_1 \tag{29} $$

Taking advantage of the fact that $\hat{c}_2 = \hat{b}_2$ and $\hat{d}_1 = \hat{c}_1$ We can write ${}^{\mathcal{I}}\vec{\omega}^{\mathcal{D}}$ and ${}^{\mathcal{I}}\vec{\omega}^{\mathcal{C}}$ as:

$$ {}^{\mathcal{I}}\vec{\omega}^{\mathcal{D}} = \dot{\theta}(\cos\phi\hat{c}_1 - \sin\phi\hat{c}_3) - \dot{\phi}\hat{c}_2 + \dot{\psi}\hat{c}_1 \tag{30} $$

$$ {}^{\mathcal{I}}\vec{\omega}^{\mathcal{C}} = \dot{\theta}(\cos\phi\hat{c}_1 - \sin\phi\hat{c}_3) - \dot{\phi}\hat{c}_2 \tag{31} $$

We can then apply these angular velocities to the computation of the inertial time derivatives of each element dw.

$$ {}^{\mathcal{I}}\vec{v}_{dw/O} = \frac{{}^{\mathcal{I}}d}{dt}[\vec{r}_{dw1/O}] = s({}^{\mathcal{I}}\vec{\omega}^{\mathcal{C}} \times \hat{c_1}) + {}^{\mathcal{I}}\vec{\omega}^{\mathcal{B}} \times (h\hat{b}_2 + R\hat{b}_1) + \dot{y}\hat{e}_y + \dot{z}\hat{e}_z \tag{32} $$

$$ = -s\dot{\theta}\sin\phi\hat{c}_2 + s\dot{\phi}\hat{c}_3 + h\dot{\theta}\hat{b}_3 + \dot{y}\hat{e}_y + \dot{z}\hat{e}_z \tag{33} $$

$$ [{}^{\mathcal{I}}\vec{v}_{dw/O}]_{\mathcal{D}} = \begin{bmatrix} \cos\psi\left(-s\dot{\theta}\sin\phi + \dot{y}\cos\theta + \dot{z}\sin\theta\right) + \sin\psi\left(s\dot{\phi} + h\dot{\theta}\cos\phi + (-\dot{y}\sin\theta + \dot{z}\cos\theta)\cos\phi\right) \\ h\dot{\theta}\sin\phi + (-\dot{y}\sin\theta + \dot{z}\cos\theta)\sin\phi \\ -\sin\psi\left(-s\dot{\theta}\sin\phi + \dot{y}\cos\theta + \dot{z}\sin\theta\right) + \cos\psi\left(s\dot{\phi} + h\dot{\theta}\cos\phi + (-\dot{y}\sin\theta + \dot{z}\cos\theta)\cos\phi\right) \end{bmatrix}_{\mathcal{D}} \tag{34} $$

$$ [{}^{\mathcal{I}}\vec{v}_{dw/O}]_{\mathcal{I}} = \begin{bmatrix} -s\dot{\phi}\sin\phi \\ -s\dot{\phi}\cos\phi\sin\theta - h\dot{\theta}\sin\theta - s\dot{\theta}\sin\phi\cos\theta + \dot{y} \\ s\dot{\phi}\cos\phi\cos\theta + h\dot{\theta}\cos\theta + \dot{z} - s\dot{\theta}\sin\phi\sin\theta \end{bmatrix} \tag{35} $$

Using this definition of the velocity vector in coordinates of the wing's frame, I have decided to define a vector, ${}^{\mathcal{I}}\vec{U}_{dw/O}$, which will define the projection of the velocity of each element of the wing, dw, onto $\hat{d}_2$-$\hat{d}_3$ plane: The aerodynamic forces on each element of width ds can be represented as:

$$ \vec{U}_{dw}(s) = ({}^{\mathcal{I}}\vec{v}_{dw/O} \cdot \hat{d}_2)\hat{d}_2 + ({}^{\mathcal{I}}\vec{v}_{dw/O} \cdot \hat{d}_3)\hat{d}_3 \tag{36} $$

$$ U_{dw}(s)^2 = ({}^{\mathcal{I}}\vec{v}_{dw/O} \cdot \hat{d}_2)^2 + ({}^{\mathcal{I}}\vec{v}_{dw/O} \cdot \hat{d}_3)^2 \tag{37} $$

With that definition of ${}^{\mathcal{I}}\vec{U}_{dw/O}$, we can proceed to define the aerodynamic forces using the formulas provided by Andersen et al based on numerical simulations conducted by Pesavento and Wang (2004) and experiments conducted by Andersen et al. (2005).

Lift

The lift is conventionally computed using the following equation:

$$ |\vec{F}_L| = \rho_{air} U_\infty \Gamma \tag{38} $$

And the direction is perpendicular to the direction of movement relative to the air. Positive lift is defined to be in the direction of a 90 degree counter clockwise rotation of the vector of the wing element's motion relative to the ambient air. using the variables and parameters I have defined, the model used states that the circulation, Γ, can be defined as:

$$ \Gamma(s) = -\frac{C_T \ell_2 (\vec{U}_{dw}(s) \cdot \hat{d}_2)(\vec{U}_{dw}(s) \cdot \hat{d}_3)}{|\vec{U}_{dw}(s)|} + \frac{C_R \ell_2^2}{2}({}^{\mathcal{I}}\vec{\omega}^{\mathcal{D}} \cdot \hat{d}_1) \tag{39} $$

Where C_T and C_R are constants. The lift force vector per unit length along the wing can then be written as:

$$\vec{f}_{lift} = \rho_{air}\left(-\frac{C_T\ell_2(\vec{U}_{dw}(s)\cdot\hat{d}_2)(\vec{U}_{dw}(s)\cdot\hat{d}_3)}{|\vec{U}_{dw}(s)|} + \frac{C_R\ell_2^2}{2}({}^{\mathcal{I}}\omega^{\mathcal{D}}\cdot\hat{d}_1)\right)\cdot\underline{\underline{R}}\vec{U}_{dw}(s) \tag{40}$$

Where $\underline{\underline{R}} = \begin{bmatrix}1 & 0 & 0\\ 0 & 0 & -1\\ 0 & 1 & 0\end{bmatrix}$.

Drag

The drag contribution per unit length to the net aerodynamic force can be described using the standard drag model:

$$\vec{f}_{drag}\cdot ds = -\frac{1}{2}C_D(\alpha(s))\rho\ell_2 U_{dw}(s)^{\mathcal{I}}\vec{U}_{dw/O}(s)\cdot ds \tag{41}$$

where, for the range of intermediate to high Reynolds number, $C_D = A - B\frac{(\vec{U}_{dw}(s)\cdot\hat{d}_2)^2-(\vec{U}_{dw}(s)\cdot\hat{d}_3)^2}{\vec{U}_{dw}(s)\cdot\vec{U}_{dw}(s)}$, and A and B are dimensionless constants (Andersen et. al. 2005). The dependence on the angle of attack, $\alpha(s)$, is implicit.

5.2.7 Final Form of Aerodynamic Force

Now that the lift and drag contributions to the aerodynamic force have been defined, we can write the net aerodynamic force vector experienced by the right wing as:

$$\vec{F}_{aero} = \rho_{air}\int_0^{\ell_1}\left(\Gamma(s)\underline{\underline{R}} - \frac{C_D(s)\ell_2 U_{dw}(s)}{2}\underline{\underline{I}}^{3\times3}\right)\ {}^{\mathcal{I}}\vec{U}_{dw}(s)ds \tag{42}$$

Taking advantage of symmetry between the total aerodynamic force terms in the $\hat{e}_y$ and $\hat{e}_z$ directions respectively are $2(\vec{F}_{aero}\cdot\hat{e}_y)$ and $2(\vec{F}_{aero}\cdot\hat{e}_z)$

5.2.8 Final Translational Equations of Motion with Blade Element Method

Putting all of these elements together, we can write the final translational equations of motion in the following dimensional forms:

$$\ddot{y} = -\frac{2m\left(-h\ddot{\theta}\sin(\theta) - h\dot{\theta}^2\cos(\theta)\right)}{2m+M} +$$
$$\frac{2\rho_{air}\ell_2}{2m+M}\left[\int_0^{\ell_1}\left(\left(-\frac{C_T(\vec{U}_{dw}(s)\cdot\hat{d}_2)(\vec{U}_{dw}(s)\cdot\hat{d}_1)}{|\vec{U}_{dw}(s)|} + \frac{C_R\ell_2}{2}({}^{\mathcal{I}}\omega^{\mathcal{D}}\cdot\hat{d}_1)\right)\underline{\underline{R}} - \frac{C_D(s)U_{dw}(s)}{2}\underline{\underline{I}}^{3\times3}\right)\ {}^{\mathcal{I}}\vec{U}_{dw}(s)ds\right]\cdot\hat{e}_y$$

$$\ddot{z}(t) = -\frac{2m}{2m+M}\left(\frac{\ell_1}{2}\left(-\sin\phi\,\dot{\phi}^2 + \cos\phi\,\ddot{\phi}\right) + h\left(-\sin\theta\,\dot{\theta}^2 + \cos\theta\,\ddot{\theta}\right)\right) - g +$$
$$\frac{2\rho_{air}\ell_2}{2m+M}\left[\int_0^{\ell_1}\left(\left(-\frac{C_T(\vec{U}_{dw}(s)\cdot\hat{d}_2)(\vec{U}_{dw}(s)\cdot\hat{d}_1)}{|\vec{U}_{dw}(s)|} + \frac{C_R\ell_2}{2}({}^{\mathcal{I}}\omega^{\mathcal{D}}\cdot\hat{d}_1)\right)\underline{\underline{R}} - \frac{C_D(s)U_{dw}(s)}{2}\underline{\underline{I}}^{3\times3}\right)\ {}^{\mathcal{I}}\vec{U}_{dw}(s)ds\right]\cdot\hat{e}_z$$

5.2.9 Dimensionless Forms:

$$\Pi_1 = \frac{2m}{2m+M} \quad \Pi_2 = \frac{2\rho\ell_2\ell_1^2}{2m+M} \quad C_T, C_R, C_D = \text{Dimensionless Coefficients} \quad \pi_1 = \frac{s}{\ell_1} \quad \pi_2 = \frac{\ell_2}{\ell_1} \tag{43}$$

$$\pi_3 = \frac{r}{\ell_1} \qquad \pi_4 = \frac{h}{\ell_1} \tag{44}$$

$$\tilde{\ddot{y}} = -\Pi_1 \left(-\pi_4 \tilde{\ddot{\theta}} \sin\left(\tilde{\theta}\right) - \pi_4 \tilde{\dot{\theta}}^2 \cos\left(\tilde{\theta}\right)\right) +$$
$$\Pi_2 \left[\int_0^1 \left(\left(-\frac{C_T(\tilde{\vec{U}}_{dw}(\pi_1)\cdot\hat{d}_2)(\tilde{\vec{U}}_{dw}(\pi_1)\cdot\hat{d}_1)}{|\tilde{\vec{U}}_{dw}(\pi_1)|} + \frac{C_R\pi_2}{2}({}^{\mathcal{I}}\tilde{\vec{\omega}}^{\mathcal{D}}\cdot\hat{d}_1)\right)\underline{\underline{R}} - \frac{C_D(\pi_1)\tilde{U}_{dw}(\pi_1)}{2}\underline{\underline{I}}^{3\times3}\right)\ {}^{\mathcal{I}}\tilde{\vec{U}}_{dw}(\pi_1)d\pi_1\right]\cdot\hat{e}_y$$

$$\tilde{\ddot{z}}(t) = -\Pi_1 \left(\frac{1}{2}\left(-\sin\tilde{\phi}\,\tilde{\dot{\phi}}^2 + \cos\tilde{\phi}\,\tilde{\ddot{\phi}}\right) + \pi_4\left(-\sin\tilde{\theta}\,\tilde{\dot{\theta}}^2 + \cos\tilde{\theta}\,\tilde{\ddot{\theta}}\right)\right) - 1+$$
$$\Pi_2 \left[\int_0^1 \left(\left(-\frac{C_T(\tilde{\vec{U}}_{dw}(\pi_1)\cdot\hat{d}_2)(\tilde{\vec{U}}_{dw}(\pi_1)\cdot\hat{d}_1)}{|\tilde{\vec{U}}_{dw}(\pi_1)|} + \frac{C_R\pi_2}{2}({}^{\mathcal{I}}\tilde{\vec{\omega}}^{\mathcal{D}}\cdot\hat{d}_1)\right)\underline{\underline{R}} - \frac{C_D(\pi_1)\tilde{U}_{dw}(\pi_1)}{2}\underline{\underline{I}}^{3\times3}\right)\ {}^{\mathcal{I}}\tilde{\vec{U}}_{dw}(\pi_1)d\pi_1\right]\cdot\hat{e}_z$$

All the linear position, speed, and acceleration terms have ℓ_1, $\sqrt{g\ell_1}$, and g factored out of them, respectively. All the angular speed and acceleration terms have $\sqrt{\frac{g}{\ell_1}}$ and $\frac{g}{l}$ factored out of them, respectively.

5.2.10 Limitations of this Model

5.3 Validating the Equations of Motion

5.3.1 Varying Parameters to Find Equilibria

In order to identify equilibria, I chose dimensionless coefficients that would correlate with a housefly. For my testing I used: $C_T : 2.5, C_R : \pi, A_{CD} : 1.4, B_{CD} : 1.0, \Pi_1 : 0.23, \Pi_2 : 4.5, \pi_2 : 0.37, , \pi_3 : 0.2, \pi_4 : 0.25$. These based on rough estimation of the dimensions of a typical housefly, but can be varied for other things as well. The code modified to do this in **Appendix A.1** .

The primary part I examined to determine if I could find equilibria was the input wing behavior for ϕ and ψ. With ϕ being the input for the wing beat angle, I expressed phi with the following time dependent function, playing around with the amplitude, the frequency, the center of the beating:

$$\phi(t) = A_\phi \cos(\omega_\phi t) + \phi_c \tag{45}$$

For ψ I chose my functional parameters based on ψ(t). From initially playing around with a constant ψ, I came to find that changing ψ between the upstroke and the downstroke would be allow for better control of translational motion. So my input for ψ shared the same frequency as ϕ but is a sine function instead of a cosine one due to the fact I wanted max ψ for the upstroke and for the downstroke (i.e., different ψ's when $\dot{\phi}$ is negative or positive):

$$\psi(t) = (\psi_+ - \psi_-)\sin(\omega_\phi t) + \psi_c \tag{46}$$

where $\psi_c = \frac{\psi_+ + \psi_-}{2}$ and +/- denote up and down stroke phi values. Also inputting the respective derivatives, I get my general input functions:

```
def phi_input(t):
    return A1 * np.cos(w1*t) +np.deg2rad(30);

def phi_d_input(t):
    return -A1 * (w1*np.sin(w1*t))

def phi_dd_input(t):
    return -A1 * w1**2 * np.cos(w1*t)

def psi_input(t):
    return -A_phi * np.sin(w1*t) + phic;

def psi_d_input(t):
    return -A_phi * (w1*np.cos(w1*t))

def psi_dd_input(t):
    return A_phi * w1**2 * np.sin(w1*t)
```

This notion of my phi and psi relationship is supported by the Wang 2005 submission in the Annual Review of Fluid Mechanics in 2005 [8]. I eventually found approximately hovering flight for the following wing input for the parameters above:

$$A_\phi = 60^\circ \qquad \phi_c = 30^\circ \qquad \omega_\phi = 30.04 \text{ Hz} \qquad \psi_+ = 87^\circ \qquad \psi_- = 4^\circ \qquad \psi_c = 45.5^\circ \tag{47}$$

Below are the resulting y and z positions over time as well as the trajectories.

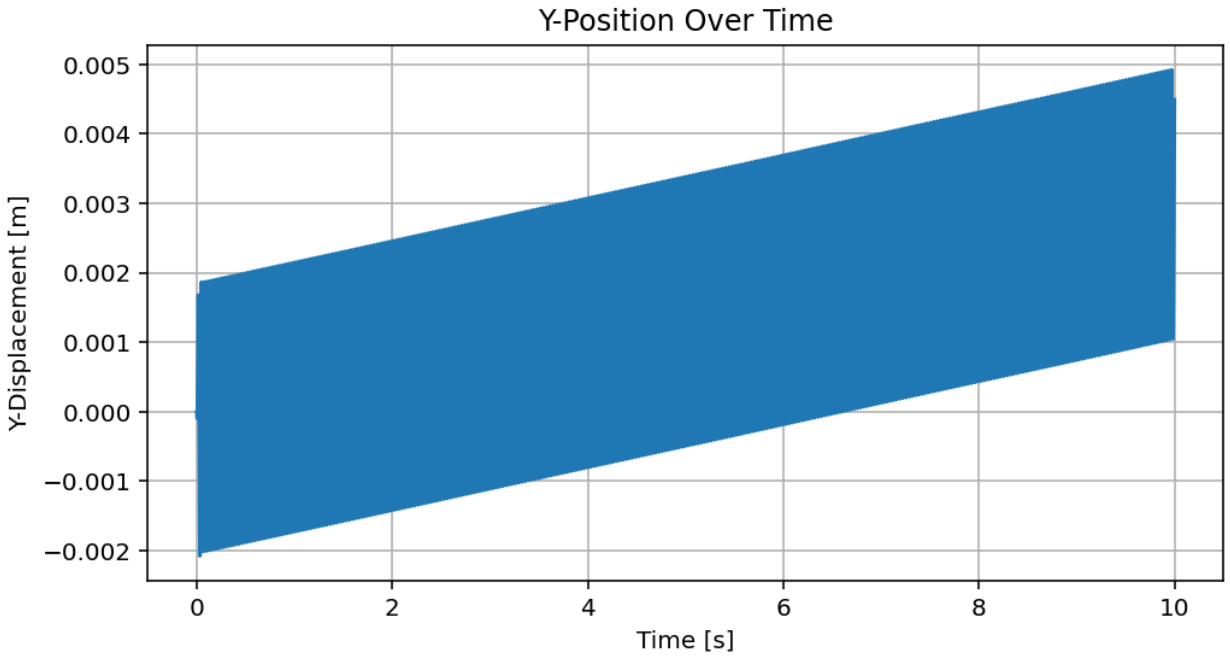


Figure 6: In this result, y only varies by 3 mm within 10 seconds.

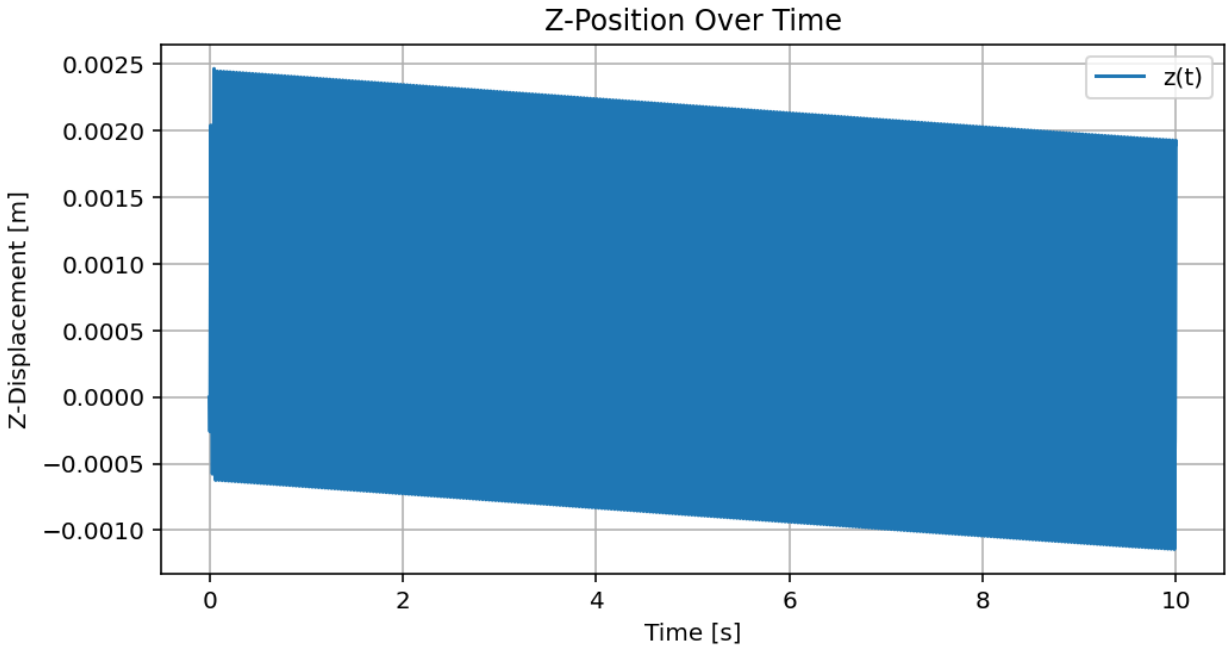


Figure 7: Although z falls slightly in this time interval, z only changes by 1.12 mm in the span of 10 seconds.

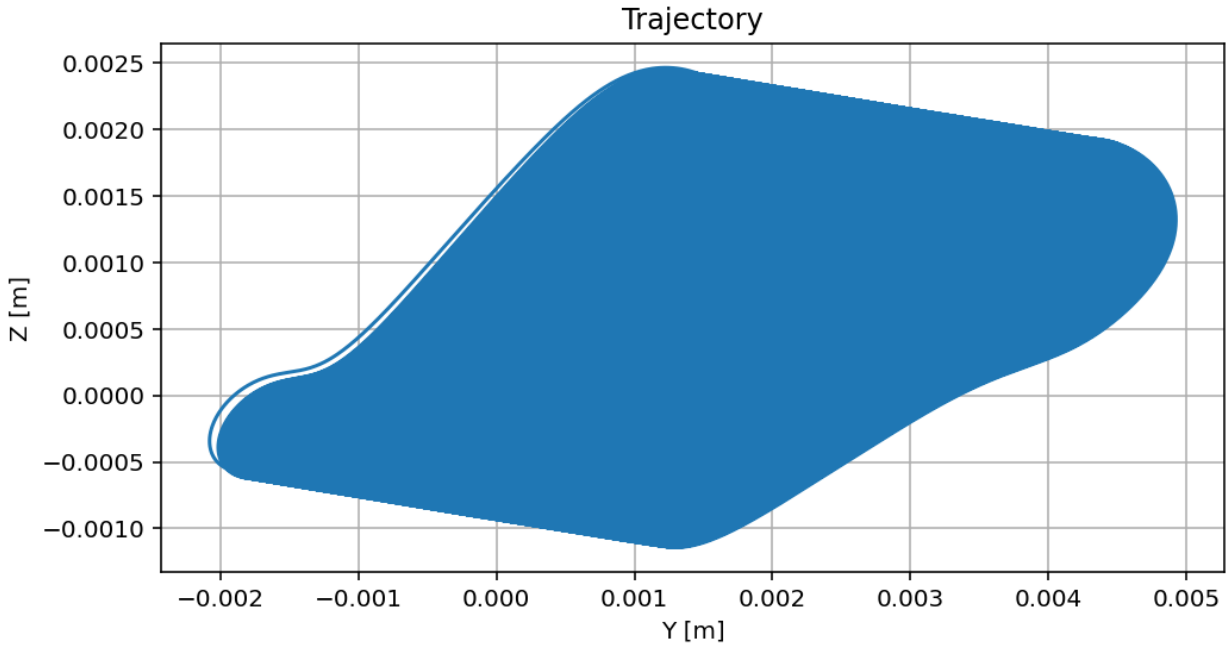


Figure 8: The resulting trajectory only inhabits a small amount of space over the span of 10 secs, so i considered this to be near a hover. I tried fine tuning the parameters as much as I could to

6 Simulation and RL Design and Decision Criteria

6.1 Aerodynamic Model Construction

The aerodynamic model is constructed symbolically using `sympy` and then compiled to efficient NumPy functions using `lambdify`. The force experienced by a differential wing section in the D-frame is a function of:

- the instantaneous wing velocity (including rotation-induced components),
- the unsteady aerodynamic circulation term Γ,
- and the nonlinear drag coefficient C_D depending on relative motion.

The model follows the thin-body, thin-elliptical section approximation employed in Andersen–Pesavento–Wang studies. This simplification allows the wing to be treated as a spanwise distribution of 2D cross-sections whose aerodynamic loads are locally proportional to velocity, circulation, and drag-like damping.

A symbolic expression for the aerodynamic force vector $\mathbf{f}_I(t, s)$ is generated, where $s \in [0, 1]$ parametrizes the wing span. The final expressions for the vertical and lateral accelerations due to aerodynamic forces are implemented via lambdified functions:

$$f_y(t, s) = f_{\text{aero},y}(t, \phi, \dot{\phi}, \ddot{\phi}, \psi, \dot{\psi}, \ddot{\psi}, y, \dot{y}, z, \dot{z}),$$

$$f_z(t, s) = f_{\text{aero},z}(t, \phi, \dot{\phi}, \ddot{\phi}, \psi, \dot{\psi}, \ddot{\psi}, y, \dot{y}, z, \dot{z}).$$

6.1.1 Gauss–Legendre Integration of Aerodynamic Forces

At every simulation step, the aerodynamic force must be integrated along the span:

$$F_y(t) = \int_0^1 f_y(t, s)\, ds, \qquad F_z(t) = \int_0^1 f_z(t, s)\, ds.$$

Rather than discretizing into a coarse mesh, the environment uses SciPy's adaptive Gaussian integration routine `quad`, which is based on Gauss–Legendre quadrature. This technique adaptively refines evaluation points to track strong nonlinearities in the integrand. The advantage is twofold:

- It minimizes computational cost by concentrating evaluations where the wing-induced velocities or aerodynamic terms change rapidly.
- It provides symbolically accurate spanwise integration for the highly nonlinear velocity and circulation expressions.

This is critical for flapping wings, where rotation-induced velocities create sharp gradients near the wing root or tip.

6.2 Equation of Motion and ODE Integration

The rigid-body translational motion of the body is governed by:

$$\ddot{y} = \ddot{y}_{\text{na}} + F_y(t), \qquad \ddot{z} = \ddot{z}_{\text{na}} + F_z(t),$$

where $\ddot{y}_{\text{na}}$ and $\ddot{z}_{\text{na}}$ are non-aerodynamic inertial terms derived from the flapping kinematics. These are determined symbolically and lambdified for numerical evaluation.

The environment integrates:

$$\frac{d}{dt}\begin{bmatrix} y \\ z \\ \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} \dot{y} \\ \dot{z} \\ \ddot{y} \\ \ddot{z} \end{bmatrix},$$

using the RK45 integrator provided by `solve_ivp`. RK45 is well-suited because:

- The aerodynamic forces are smooth within each transition interval.
- Adaptive stepping improves stability for stiff motion segments near stroke reversals.
- It provides an efficient balance between accuracy and computation time for RL rollouts.

6.3 Reinforcement Learning Setup and Algorithmic Structure

The design of the reinforcement learning framework for identifying flapping-wing kinematics required a series of modeling choices, numerical strategies, and structural decisions in the implementation of the training environment. This section outlines the reasoning behind these design choices and explains the underlying algorithms used in the FlappingEnv, Wings kinematic class, rollout collection utilities, and PPO training pipeline. The goal is to ensure that the simulated environment remains physically plausible, numerically stable, and compatible with reinforcement learning algorithms that operate through repeated time integration.

6.3.1 Flapping Environment

The `FlappingEnv` class defines a continuous-time, physics-based environment that follows the OpenAI Gym interface. Each RL action specifies a full parameterization of the wing kinematics:

$$\mathbf{a} = [A_\phi,\ \omega,\ \phi_c,\ \delta_\phi,\ A_\psi,\ \psi_c,\ \delta_\psi]\,.$$

These parameters determine the flapping and pitching motion of each wing over a short interval of time. The environment integrates the rigid-body translational dynamics of a flapping-wing system from the current time t_k to t_{k+1} using `solve_ivp`.

The observation vector at time t is

$$\mathbf{o}(t) = [\,y,\ z,\ \dot{y},\ \dot{z},\ (t-t_0)/(t_f-t_0)\,],$$

allowing the agent to make decisions based on current position, velocity, and episode progress. The episode horizon is determined directly from the temporal discretization of a reference trajectory.

Time Discretization and Environment Steps

The simulation is defined over a fixed nondimensional time interval

$$t \in [t_0, t_f].$$

This interval is discretized into at most $N_{\max}$ segments ("steps") by choosing a step size

$$\Delta t = \frac{t_f - t_0}{N_{\max}}.$$

At the beginning of each episode, the environment time and state are initialized as

$$t_0 \ \text{ given}, \qquad t_{\text{curr}} = t_0, \qquad \mathbf{x}(t_0) = \big(y(t_0),\, z(t_0),\, \dot{y}(t_0),\, \dot{z}(t_0)\big)^\top.$$

At each environment step $k = 0, 1, 2, \ldots$, PPO supplies an action vector

$$\mathbf{a}_k = \begin{bmatrix} A_\phi \\ \omega \\ \phi_c \\ \Delta\phi \\ A_\psi \\ \psi_c \\ \Delta\psi \end{bmatrix},$$

which parametrizes the wing kinematics over the next time segment

$$t \in [t_k,\, t_{k+1}], \qquad t_k = t_{\text{curr}}, \quad t_{k+1} = \min\{t_k + \Delta t,\, t_f\}.$$

Given $\mathbf{a}_k$, the environment:

1. Updates the wing parameters (with a smooth transition, described below).
2. Integrates the state ODE
$$\dot{\mathbf{x}}(t) = \mathbf{f}\big(t, \mathbf{x}(t); \mathbf{a}_k\big)$$
from t_k to t_{k+1} using a time integrator (e.g. `solve_ivp`).
3. Uses the resulting trajectory on $[t_k, t_{k+1}]$ to compute a scalar reward r_k.
4. Updates the current time and state:
$$t_{\text{curr}} \leftarrow t_{k+1}, \qquad \mathbf{x}(t_{k+1}) \text{ becomes the new state.}$$

The episode terminates when either $t_{\text{curr}} \geq t_f$ or the step count reaches $N_{\max}$. Thus, during a single episode PPO selects a *sequence* of actions $\{\mathbf{a}_0, \mathbf{a}_1, \dots\}$, one per time segment, and observes the resulting rewards $\{r_0, r_1, \dots\}$.

Importantly, PPO does *not* fully optimize the parameters for one segment before moving on to the next. Instead, it repeatedly:

1. Runs rollouts over the entire time horizon by sampling actions at each step from the current policy.
2. Uses the total returns from these full episodes to update the policy parameters.

Over many iterations, this process discovers policies that choose better actions at each segment so that the *cumulative* reward over the whole interval $[t_0, t_f]$ is maximized.

6.3.2 Wing Kinematics and Smooth Parameter Transitions

The `Wings` class provides smooth, differentiable expressions for the wing angles $\phi(t)$ and $\psi(t)$, along with their first and second derivatives. Baseline motions are sinusoidal:

$$\phi_{\text{base}}(t) = \phi_c + A_\phi \cos(\omega t + \delta_\phi), \qquad \psi_{\text{base}}(t) = \psi_c + A_\psi \sin(\omega t + \delta_\psi).$$

When an RL action updates these parameters, direct replacement would cause discontinuities in $\phi, \dot{\phi}, \ddot{\phi}$ and similarly for ψ. Discontinuities create numerical instability in the ODE integration and yield unphysical instantaneous changes in aerodynamic forces. To prevent this, the environment uses a quintic transition polynomial.

Smooth Transitions Between Sinusoidal Wing Parameters

For $\phi(t)$, the total angle is
$$\phi(t) = \phi_{\text{base}}(t) + q_\phi(t),$$
where $q_\phi(t)$ is a quintic polynomial in a normalized time variable
$$s = \frac{t - t_k}{T_{\text{tr}}}, \qquad s \in [0, 1],$$
over a transition interval of length T_{tr} starting at t_k. We write
$$q_\phi(t) = \sum_{i=0}^{5} b_i s^i,$$
with coefficients b_i chosen to enforce continuity of ϕ, $\dot{\phi}$, and $\ddot{\phi}$ at both ends of the transition. Specifically, at $t = t_k$ (i.e. $s = 0$) we require
$$\begin{aligned}
\phi_{\text{old}}(t_k) &= \phi_{\text{base}}(t_k) + q_\phi(t_k), \\
\dot{\phi}_{\text{old}}(t_k) &= \dot{\phi}_{\text{base}}(t_k) + \dot{q}_\phi(t_k), \\
\ddot{\phi}_{\text{old}}(t_k) &= \ddot{\phi}_{\text{base}}(t_k) + \ddot{q}_\phi(t_k),
\end{aligned}$$

i.e. the new total angle and its first two derivatives match the old motion at the start of the segment. At $t = t_k + T_{\text{tr}}$ (i.e. $s = 1$), we enforce

$$q_\phi(t_k + T_{\text{tr}}) = 0, \qquad \dot{q}_\phi(t_k + T_{\text{tr}}) = 0, \qquad \ddot{q}_\phi(t_k + T_{\text{tr}}) = 0,$$

so that after the transition the motion is purely given by the new base sinusoid with no residual correction.

These six boundary conditions determine the six coefficients $b_0, \ldots, b_5$ in closed form. An analogous construction is used for the pitch motion $\psi(t)$, so that both $\phi(t)$ and $\psi(t)$ and their first two time derivatives remain continuous as the RL policy updates the wing–beat parameters from one segment to the next.

In summary, at each environment step PPO chooses a new set of sinusoidal parameters, the environment smoothly transitions to these parameters over a short time window, integrates the ODEs over the current segment, and computes a reward based on how well the resulting motion matches a local linear approximation to the desired hover trajectory. PPO then adjusts its policy to maximize the cumulative sum of these rewards over the entire time interval $[t_0, t_f]$.

6.3.3 Rollout Strategy

The `rollout_and_collect` utility records simulated trajectories by stitching together segments of the ODE solution, ensuring no duplicate timestamps across intervals. This produces continuous time series of:

$$t(t), \quad y(t), \quad z(t), \quad \phi(t), \quad \psi(t).$$

These are used for visualization and validation of learned policies.

6.4 Reward Function Structure and Rationale

Unlike traditional robotic systems, where position tracking or torque minimization can be evaluated instantaneously, the effects of wing kinematics on body translation are perceptually delayed and integrated over time. A small change in pitch or flapping phase does not immediately translate into a measurable displacement at the center of mass; instead, the aerodynamic forces accumulate over a portion of the wingbeat cycle. This motivates a reward function that evaluates motion trends over an interval rather than penalizing instantaneous state error.

6.4.1 Segment-Based Evaluation

At each RL step, the simulator integrates the equations of motion over a time window $[t_k, t_{k+1}]$ determined by the environment timestep. Within this window, both the simulated trajectory

$$y_{\text{sim}}(t), \quad z_{\text{sim}}(t),$$

and the target trajectory

$$y_{\text{tgt}}(t), \quad z_{\text{tgt}}(t),$$

are sampled at a set of times $\{t_i\}$. Instead of measuring pointwise error at each t_i, the environment fits a linear model to each trajectory.

6.4.2 Least-Squares Trend Extraction

Let $\{t_j\}_{j=1}^{M}$ be the time samples in the segment $[t_k, t_{k+1}]$, with corresponding simulated trajectories $y_j = y(t_j)$, $z_j = z(t_j)$ and reference values y_j^{ref}, z_j^{ref}. For any quantity q_j defined on these points, we fit a line

$$q_j \approx a_0 + a_1\, t_j$$

by minimizing the weighted squared error

$$\min_{a_0, a_1} \sum_{j=1}^{M} w_j \big(q_j - (a_0 + a_1 t_j)\big)^2,$$

where w_j are (possibly uniform) weights. This leads to the normal equations

$$\mathbf{F}\,\mathbf{a} = \mathbf{b},$$

with

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \sum_j w_j & \sum_j w_j(t_j - \bar{t}) \\ \sum_j w_j(t_j - \bar{t}) & \sum_j w_j(t_j - \bar{t})^2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \sum_j w_j q_j \\ \sum_j w_j(t_j - \bar{t}) q_j \end{bmatrix},$$

and $\bar{t}$ is a weighted mean time. Solving $\mathbf{Fa} = \mathbf{b}$ yields (a_0, a_1).

The use of least-squares provides several benefits:

- It smooths out high-frequency aerodynamic oscillations induced by flapping motion.
- It captures the net effect of a wingbeat rather than reacting to instantaneous fluctuations.
- It stabilizes learning by avoiding noisy gradients caused by short-term aerodynamic transients.

Thus, the agent is rewarded for producing the correct direction and magnitude of motion over each interval.

6.4.3 Penalty on Intercept and Slope Differences

Let $(a_0^{(y)}, a_1^{(y)})$ denote the simulated intercept and slope for $y(t)$, and $(\hat{a}_0^{(y)}, \hat{a}_1^{(y)})$ the target. The reward function penalizes deviation:

$$R = -\left[w_0 \left((a_0^{(y)} - \hat{a}_0^{(y)})^2 + (a_0^{(z)} - \hat{a}_0^{(z)})^2 \right) + w_1 \left((a_1^{(y)} - \hat{a}_1^{(y)})^2 + (a_1^{(z)} - \hat{a}_1^{(z)})^2 \right) \right].$$

The intercept error measures how far the center-of-mass has drifted from the desired displacement over that interval, whereas the slope error measures mismatch in average velocity.

The weighting choice $w_1 \gg w_0$ reflects the physical reality that *the agent should learn to generate the correct aerodynamic forces first*, establishing the correct velocity trend, and only then refine precise positional tracking.

6.4.4 Noise Rejection

Because the aerodynamic forces oscillate rapidly at flapping frequency, an instantaneous reward would be dominated by high-frequency variations unrelated to meaningful trajectory tracking. The least-squares trend extraction acts as a low-pass filter. This is crucial because:

- It makes the reward more "RL-friendly" by smoothing the optimization landscape.
- PPO's policy updates rely on stable reward gradients.
- It allows the agent to exploit the underlying physics rather than overfitting to momentary fluctuations.

Without this smoothing, the reward would fluctuate violently due to the nonlinearities in Γ, drag, and rotation-induced velocity terms.

6.4.5 Scalability and Generalization

This reward design naturally extends to more complex tasks:

- Multi-segment trajectories can be tracked by minimizing changes in slope over larger windows.
- More complex reward structures (e.g., curvature or acceleration matching) can be built atop this approach.
- The trend-based method is invariant to small time shifts, making the agent robust to slight timing errors in its flapping.

Thus the reward function plays a dual role:

1. It enforces physically meaningful behavior aligned with how flapping wings generate motion.
2. It stabilizes the RL optimization problem, allowing PPO to learn efficiently.

6.5 Training Procedure and PPO Hyperparameters

Training is performed using Stable-Baselines3 PPO. PPO is chosen due to:

- Its robustness to noisy, nonlinear environments.
- Its ability to handle continuous action spaces.
- Its sample efficiency relative to other policy-gradient methods.

Key hyperparameters include:

$$\text{batch\_size} = 64, \qquad n_{\text{steps}} = 1280, \qquad \gamma = 0.99, \qquad \text{learning rate} = 3 \times 10^{-4},$$

which balance exploration with stable policy updates.

Periodic rollouts after training allow visualization of the learned wing kinematics and the resulting center-of-mass motion.

7 Apparatuses

7.1 Computer System and Code Editor

All simulations, reinforcement learning training runs, and numerical experiments described in this study were conducted on a 13-inch MacBook Pro (2022) equipped with an Apple M2 chip and 16 GB of unified memory, running macOS Sonoma 14.5. The Apple M2 architecture provides substantial floating-point throughput and efficient memory access, which is beneficial for the repeated numerical integrations and symbolic evaluations that occur within each RL episode. Although the system does not include a discrete GPU, the PPO algorithm and the `solve_ivp` integrations used in this project are CPU-bound and therefore executed efficiently on the M2's performance cores.

Code development was performed primarily in the Spyder IDE, chosen for its MATLAB-like environment, variable explorer, and integrated debugging tools. Spyder's real-time object inspection was especially useful during development of the aerodynamic model and wing-kinematic routines, where symbolic expressions and intermediate numerical arrays required close monitoring. However, all large-scale PPO training runs and rollout evaluations were executed from the macOS terminal. Running the code in a terminal environment reduced overhead, ensured deterministic library loading, and enabled long training sessions without interruption from IDE-related memory usage or interface delays.

This combination of Spyder for editing and terminal execution offered a flexible workflow: interactive debugging and iterative refinement during development, followed by stable command-line execution for extended numerical simulations. Despite being performed on a laptop system without specialized hardware accelerators, the computations remained tractable due to (i) the reduced-order aerodynamic model, (ii) efficient vectorized NumPy/SciPy routines, and (iii) PPO's stable on-policy training that avoids excessively large batch sizes. Training stability and simulation speed were further enhanced by compiling symbolic expressions into optimized NumPy functions via `sympy.lambdify`, enabling the environment to evaluate aerodynamic forces and inertial terms in real time during PPO rollouts.

Overall, the computational setup was sufficient for the reinforcement-learning-based identification of wing-beating kinematics, and the development workflow supported both rapid prototyping and long-duration numerical experimentation.

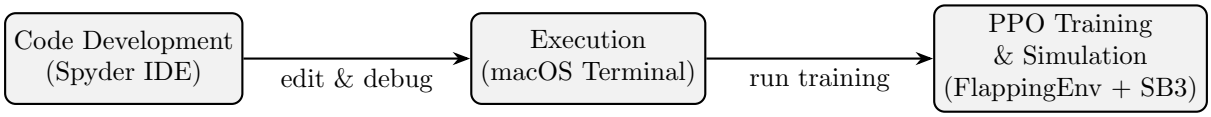


Figure 9: Workflow used for code development, debugging, and PPO training.

7.2 Software and Computational Libraries

The implementation of the flapping-wing reinforcement learning system relies on a collection of scientific computing and machine-learning libraries within the Python ecosystem. Each library was selected based on numerical stability, efficiency, and compatibility with symbolic modeling and RL frameworks.

NumPy. NumPy provides the foundational array structures and vectorized operations used throughout the aerodynamic model, wing kinematics, and environment integration. All high-frequency numerical computations, including velocity transformations, evaluation of symbolic expressions, and force aggregation, depend on NumPy's optimized linear algebra routines.

SciPy. SciPy is used primarily for numerical integration, including the Runge–Kutta ODE solver `solve_ivp` and the Gauss–Legendre quadrature routine `quad`. These tools enable stable and accurate time integration of the nonlinear dynamics and spanwise aerodynamic forces.

SymPy. SymPy is used to construct and symbolically simplify expressions for aerodynamic forces, inertial terms, and reference-frame transformations. After symbolic derivation, these expressions are compiled into efficient numerical functions using `lambdify`, allowing real-time evaluation during PPO rollouts.

Gymnasium / OpenAI Gym API. The flapping-wing environment (`FlappingEnv`) adheres to the Gym API, enabling interoperability with reinforcement learning libraries. The step-based architecture naturally integrates with PPO's collection of state–action–reward sequences.

Stable-Baselines3 (SB3). PPO training is performed using the Stable-Baselines3 implementation. SB3 provides a reliable, well-tested PPO optimizer that supports continuous action spaces, making it well-suited for learning wing-kinematic parameters. Its implementation also ensures efficient batching and stable policy updates, which are essential for simulation-heavy environments like this one.

Matplotlib. Matplotlib is used for visualization of rollout trajectories, wing kinematic histories, and evaluation diagnostics. All trajectory plots and post-training validation figures are generated using this library.

Together, these tools form a computational stack that supports symbolic derivation, numerical integration, real-time aerodynamic evaluation, and reinforcement learning optimization within a single, cohesive framework.

8 Results

8.1 Preliminary Results

During the initial reinforcement learning runs, a very simple reward function was used that penalized deviations from the initial y and z positions in order to encourage hovering behavior. The environment was configured with a maximum of 200 steps per episode and a total training horizon of 10,000 timesteps, with 512 timesteps collected per PPO rollout. In this first attempt, the action space was intentionally chosen to be large in order to allow the policy maximum flexibility in selecting wing–beat parameters.

The results of this first run are shown in Figures 10–11. The system quickly diverged from the desired hover condition, and the wing–beat parameters showed little meaningful adaptation over time.

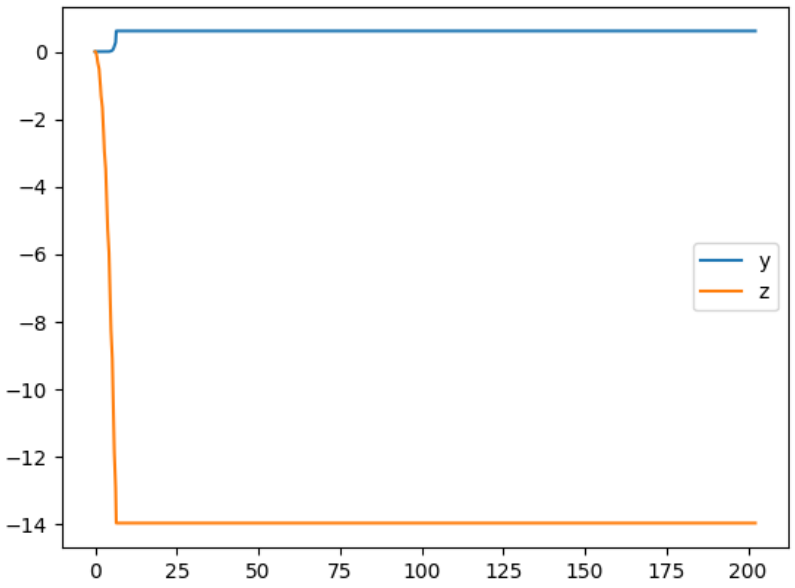


Figure 10: Body position over time for the initial reinforcement learning run. The horizontal and vertical axes represent dimensionless time and position, respectively.

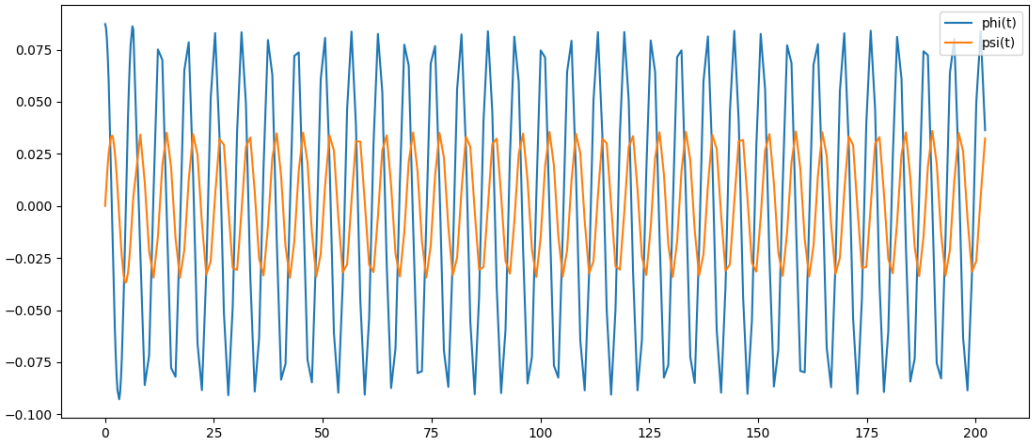


Figure 11: Wing kinematic angles over time for the initial reinforcement learning run. The horizontal and vertical axes represent dimensionless time and angle, respectively.

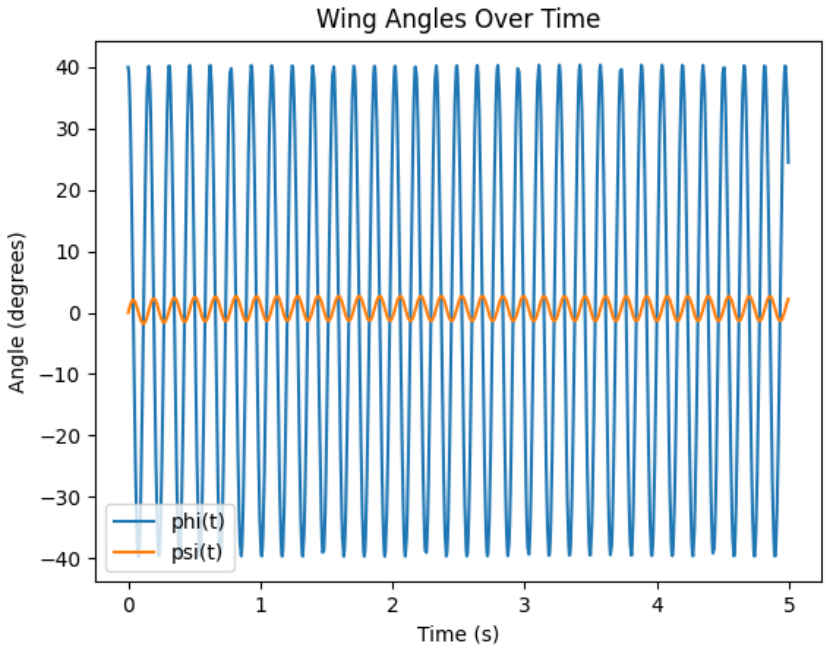


Figure 12: Trajectory of the body center of mass for the initial reinforcement learning run.

The large initial action space made it difficult for the policy to identify parameter combinations that produced near–equilibrium behavior. Examination of the PPO training logs revealed that both the episode reward mean and the action distribution standard deviation remained nearly constant across iterations, indicating that the policy was not effectively exploring or improving. This behavior suggested that the optimization problem was ill–conditioned due to the scale of the action space and the simplicity of the reward definition.

To address this, the action bounds were reduced and the reward function was modified. Several intermediate reinforcement learning runs were conducted using different reward scalings and parameter limits. Representative results from these additional runs are shown in Figures 13–15. In these cases, the policy was able to reduce drift in the y direction, but significant downward motion in z remained.

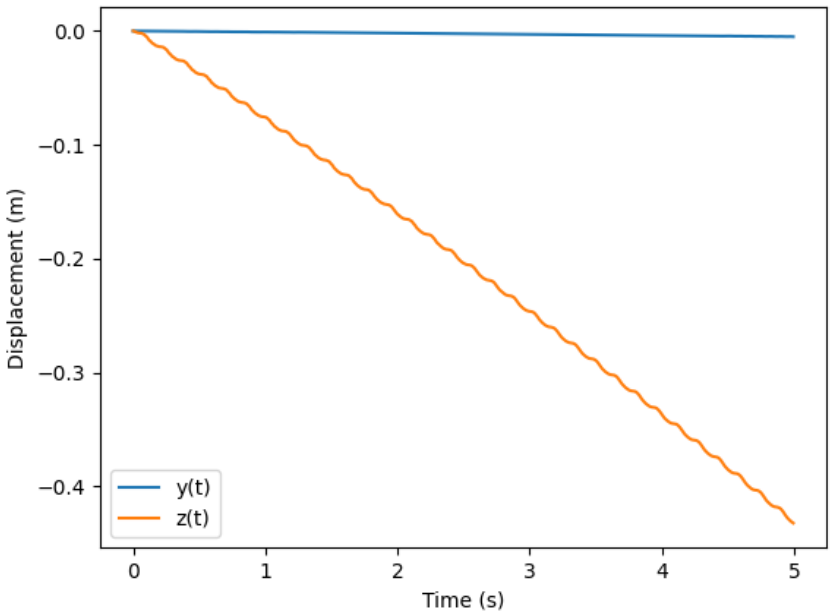


Figure 13: Intermediate reinforcement learning result showing partial stabilization in y.

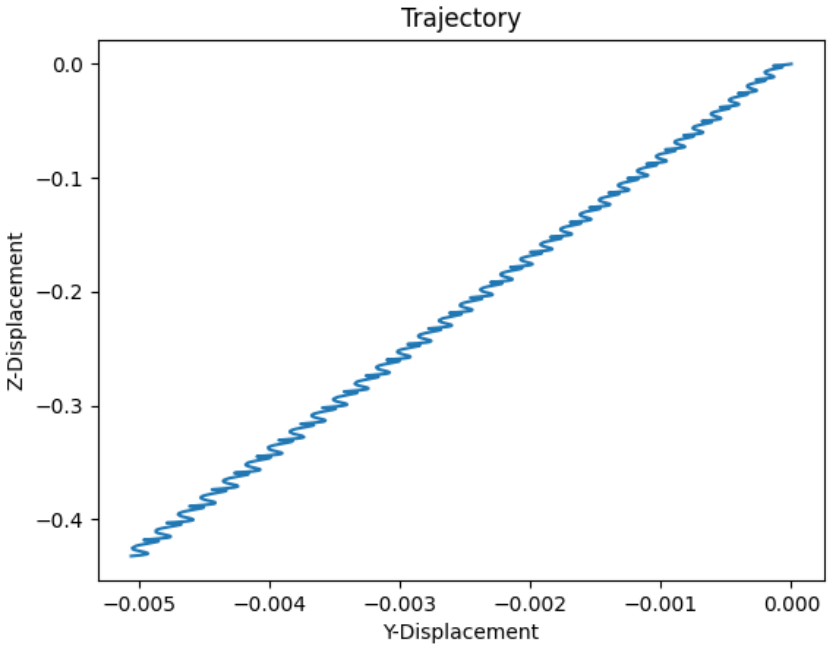


Figure 14: Body position over time for an intermediate reinforcement learning run.

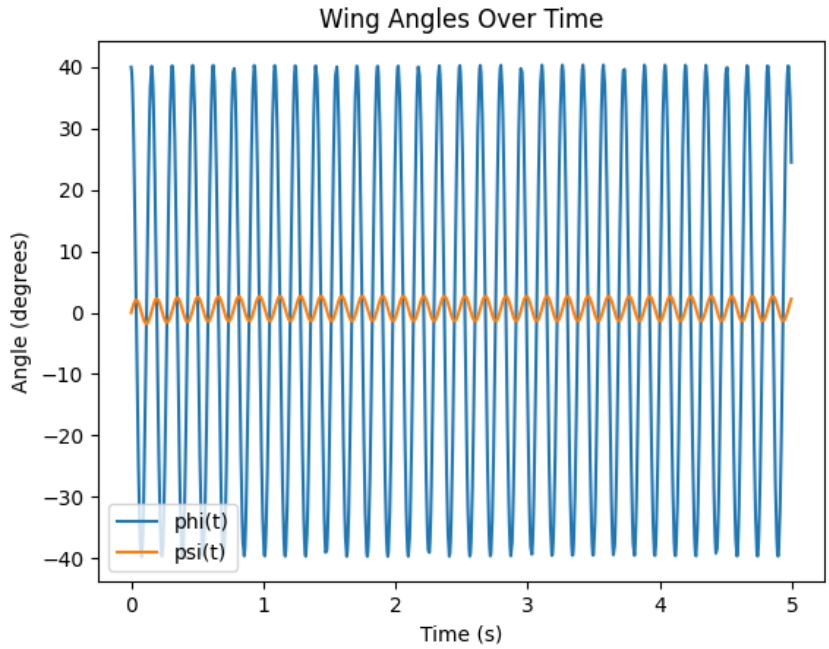


Figure 15: Wing kinematics corresponding to the intermediate reinforcement learning run.

A subsequent experiment focused on scaling the reward function in order to improve the numerical conditioning of the learning signal. With this rescaled reward, the PPO policy finally began to exhibit consistent learning behavior. Figures **??**–**??** show the wing kinematics, body motion, and trajectory for this run.

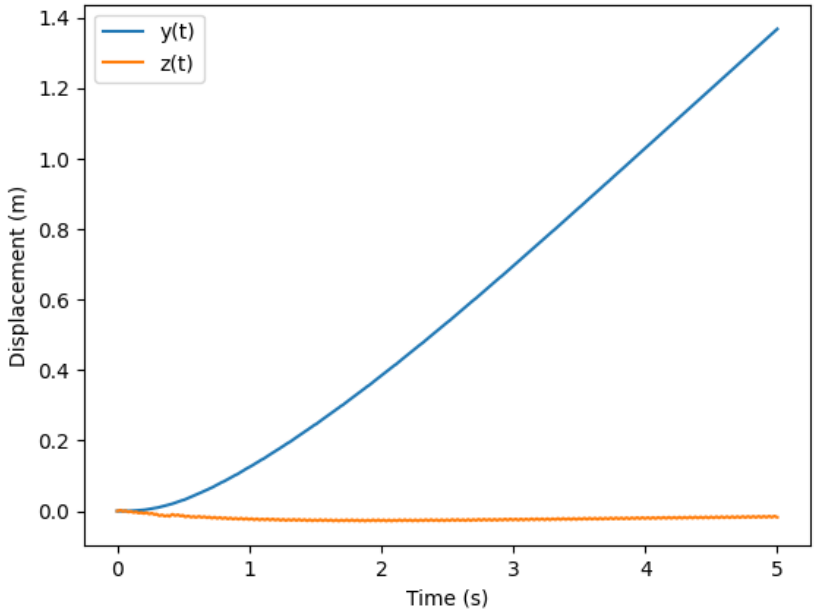


Figure 16: y and z displacements over time for the refined run.

And then another run with more time steps over the entire environment.

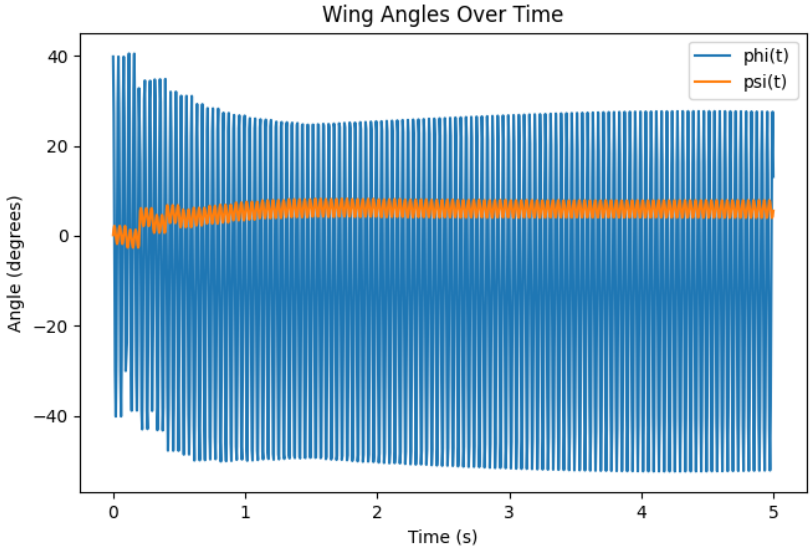


Figure 17: Wing kinematics of run with more timesteps.

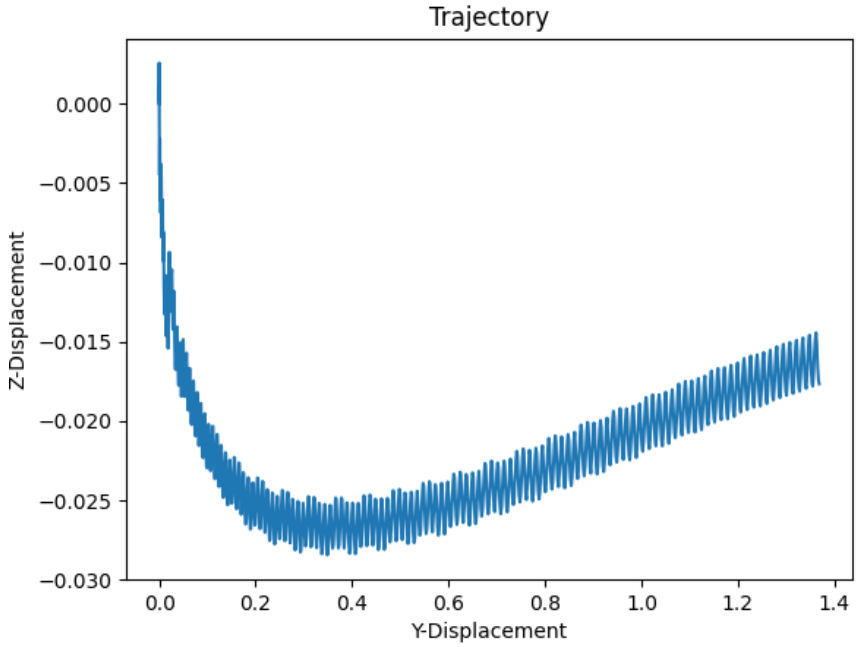


Figure 18: Trajectory from run with more timesteps.

During this rescaled reward run, the episode reward increased by over 25%, indicating that the policy was successfully identifying parameter changes that improved performance. After confirming that learning was occurring, the reward function was reverted to the originally intended formulation, and a much longer training run was initiated in order to assess whether the policy could continue improving over extended training.

The early iterations of this longer run are shown below, followed by the final iterations after approximately 17 hours of training:

```
[initial PPO training log excerpts]
---------------------------------
| rollout/           |           |
|    ep_len_mean     | 200       |
|    ep_rew_mean     | -1.75e+04 |
| time/              |           |
|    fps             | 1         |
|    iterations      | 1         |
|    time_elapsed    | 334       |
|    total_timesteps | 512       |
---------------------------------
----------------------------------------
| rollout/                |            |
|    ep_len_mean          | 200        |
|    ep_rew_mean          | -2.08e+04  |
| time/                   |            |
|    fps                  | 1          |
|    iterations           | 2          |
|    time_elapsed         | 643        |
|    total_timesteps      | 1024       |
| train/                  |            |
```

```
|    approx_kl            | 0.006990889 |
|    clip_fraction        | 0.0482      |
|    clip_range           | 0.2         |
|    entropy_loss         | -9.93       |
|    explained_variance   | -0.000166   |
|    learning_rate        | 0.0003      |
|    loss                 | 1.3e+06     |
|    n_updates            | 10          |
|    policy_gradient_loss | -0.0113     |
|    std                  | 0.998       |
|    value_loss           | 3.46e+06    |
-----------------------------------------
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 200         |
|    ep_rew_mean          | -1.9e+04    |
| time/                   |             |
|    fps                  | 1           |
|    iterations           | 3           |
|    time_elapsed         | 934         |
|    total_timesteps      | 1536        |
| train/                  |             |
|    approx_kl            | 0.005791565 |
|    clip_fraction        | 0.0158      |
|    clip_range           | 0.2         |
|    entropy_loss         | -9.91       |
|    explained_variance   | 0.00179     |
|    learning_rate        | 0.0003      |
|    loss                 | 1.87e+06    |
|    n_updates            | 20          |
|    policy_gradient_loss | -0.00826    |
|    std                  | 0.997       |
|    value_loss           | 3.71e+06    |
-----------------------------------------

[final PPO training log excerpts]
-----------------------------------------
| rollout/                |              |
|    ep_len_mean          | 200          |
|    ep_rew_mean          | -3.51e+03    |
| time/                   |              |
|    fps                  | 1            |
|    iterations           | 194          |
|    time_elapsed         | 54709        |
|    total_timesteps      | 99328        |
| train/                  |              |
|    approx_kl            | 0.00011171156 |
|    clip_fraction        | 0            |
|    clip_range           | 0.2          |
|    entropy_loss         | -8.42        |
|    explained_variance   | 0.0662       |
|    learning_rate        | 0.0003       |
|    loss                 | 1e+05        |
|    n_updates            | 1930         |
```

```
|    policy_gradient_loss | -0.000971     |
|    std                  | 0.837         |
|    value_loss           | 1.83e+05      |
-----------------------------------------
-----------------------------------------
| rollout/                |               |
|    ep_len_mean          | 200           |
|    ep_rew_mean          | -3.51e+03     |
| time/                   |               |
|    fps                  | 1             |
|    iterations           | 195           |
|    time_elapsed         | 54969         |
|    total_timesteps      | 99840         |
| train/                  |               |
|    approx_kl            | 0.011059868   |
|    clip_fraction        | 0.0842        |
|    clip_range           | 0.2           |
|    entropy_loss         | -8.42         |
|    explained_variance   | 0.0129        |
|    learning_rate        | 0.0003        |
|    loss                 | 2.11e+05      |
|    n_updates            | 1940          |
|    policy_gradient_loss | -0.0149       |
|    std                  | 0.836         |
|    value_loss           | 3.95e+05      |
-----------------------------------------
-----------------------------------------
| rollout/                |               |
|    ep_len_mean          | 200           |
|    ep_rew_mean          | -3.44e+03     |
| time/                   |               |
|    fps                  | 1             |
|    iterations           | 196           |
|    time_elapsed         | 55229         |
|    total_timesteps      | 100352        |
| train/                  |               |
|    approx_kl            | 0.010119354   |
|    clip_fraction        | 0.138         |
|    clip_range           | 0.2           |
|    entropy_loss         | -8.41         |
|    explained_variance   | 5.13e-05      |
|    learning_rate        | 0.0003        |
|    loss                 | 1.04e+05      |
|    n_updates            | 1950          |
|    policy_gradient_loss | -0.0154       |
|    std                  | 0.835         |
|    value_loss           | 1.77e+05      |
-----------------------------------------
```

Across this extended run, the episode reward mean steadily increased by more than 75% over approximately 196 iterations, while the action distribution standard deviation gradually decreased. This behavior indicates that the policy was learning to consistently select wing–beat parameters that reduce body drift. The final results of this run, shown in Figures 19–21, demonstrate simultaneous improvement in both y and z motion.

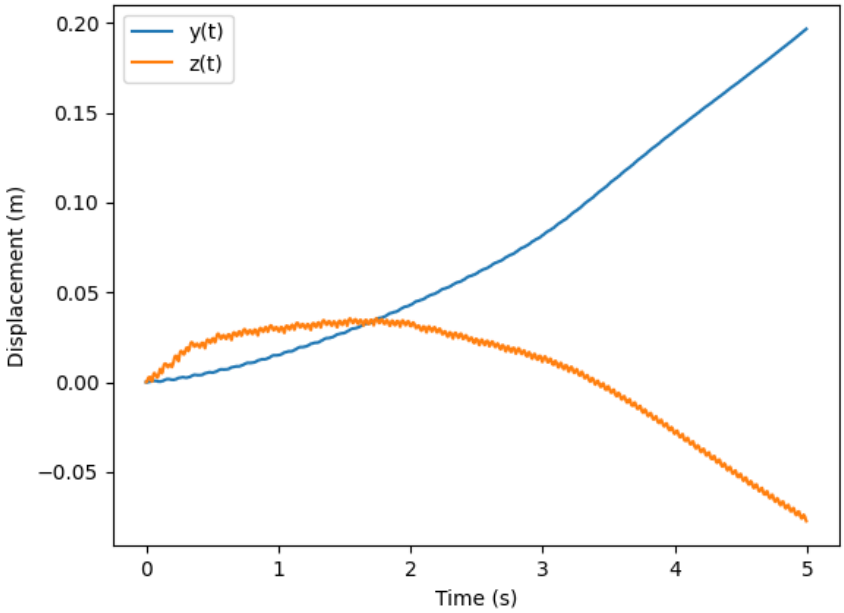


Figure 19: Body positions over the time interval

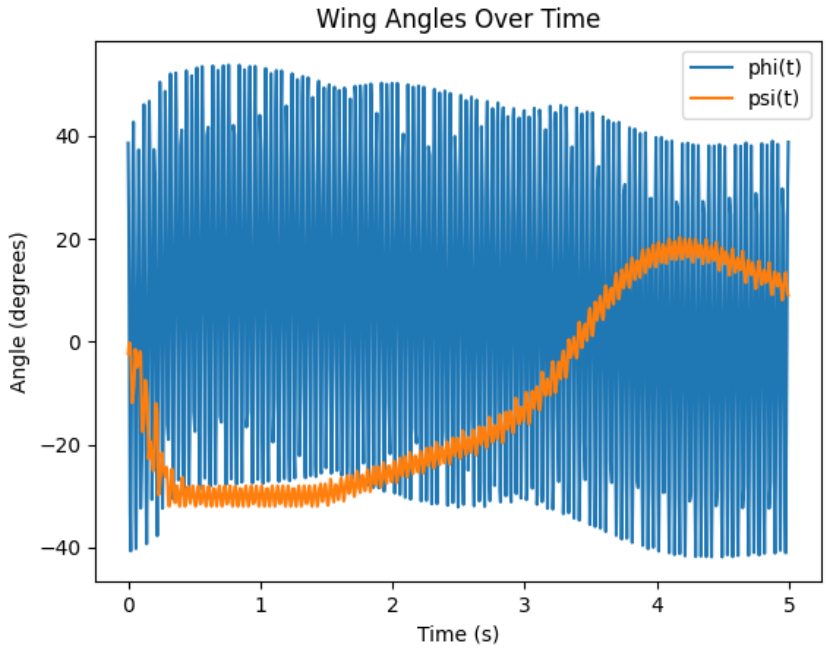


Figure 20: Wing kinematics over the time interval

While true hovering was not achieved within the available training time, these results suggest that the reinforcement learning framework is capable of discovering wing–beat parameter trends that reduce drift in both spatial directions. Continued training and further refinement of the reward structure are expected to

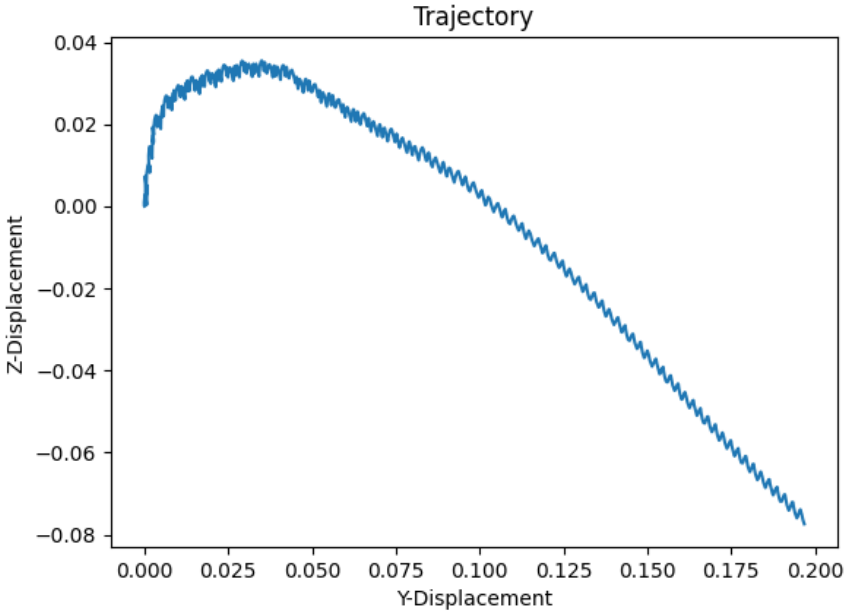


Figure 21

improve convergence toward sustained hover.

8.2 Final Results

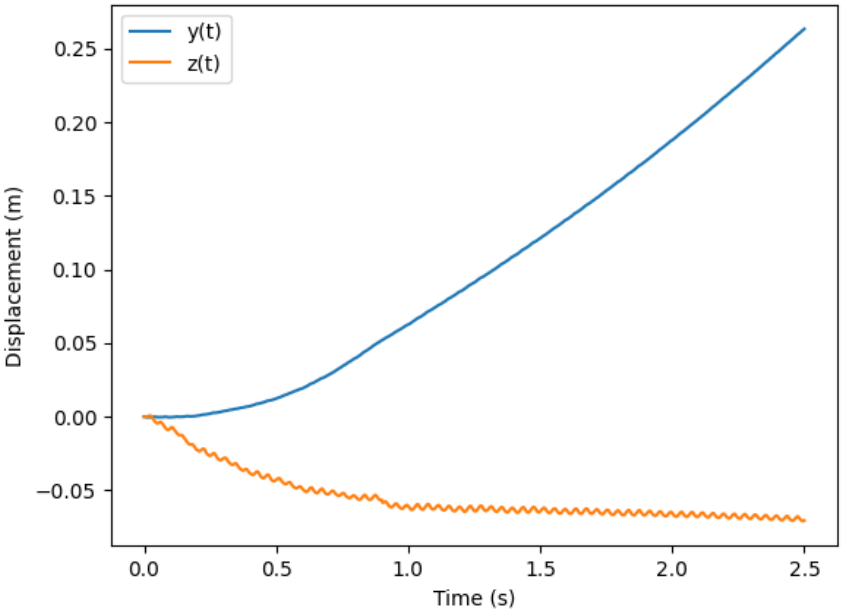


Figure 22: Y and Z displacements over a 2 second time span.

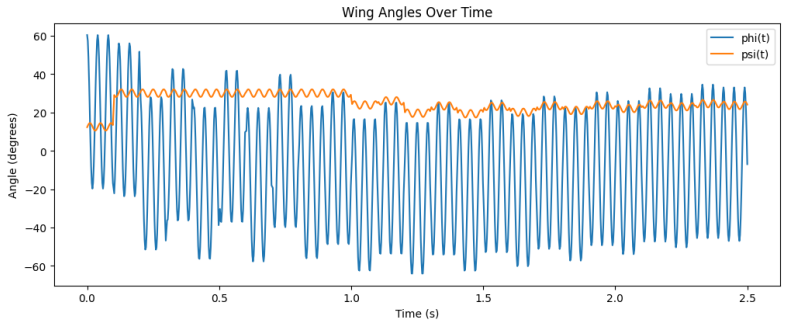


Figure 23: The wing kinematics over the time interval that the policy learned. Reward function likely indirectly over-punishes large amplitudes and high frequencies, which have been shown

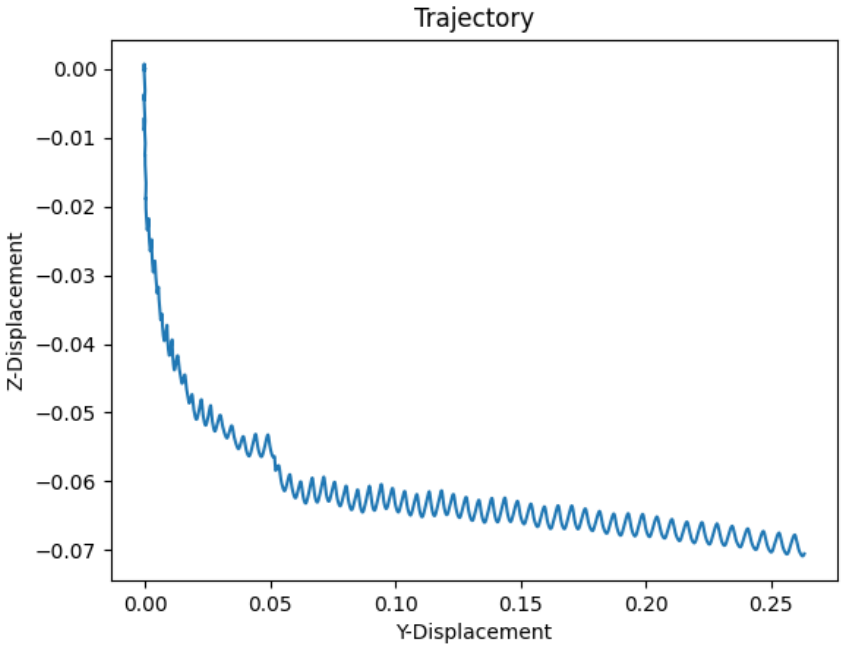


Figure 24: A outlook on what the 2.5 second trajectory looks like for the policies' chosen wing kinematics. As shown, the policy approaches hovering, but needs to explore more actions to discover true hover.

Although this run still did not result in the desired hover behavior, it is approach pretty close as shown by the drastic improvement in reward for the time steps:

```
[initial PPO training log excerpts]
----------------------------------
| rollout/           |           |
|    ep_len_mean     | 25        |
|    ep_rew_mean     | -1.73e+03 |
| time/              |           |
|    fps             | 0         |
|    iterations      | 1         |
|    time_elapsed    | 1615      |
|    total_timesteps | 1280      |
```

```
----------------------------------
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 25          |
|    ep_rew_mean          | -1.84e+03   |
| time/                   |             |
|    fps                  | 0           |
|    iterations           | 2           |
|    time_elapsed         | 3242        |
|    total_timesteps      | 2560        |
| train/                  |             |
|    approx_kl            | 0.00868952  |
|    clip_fraction        | 0.0938      |
|    clip_range           | 0.2         |
|    entropy_loss         | -9.92       |
|    explained_variance   | 2.37e-05    |
|    learning_rate        | 0.0003      |
|    loss                 | 6.56e+05    |
|    n_updates            | 10          |
|    policy_gradient_loss | -0.0159     |
|    std                  | 0.997       |
|    value_loss           | 9e+05       |
-----------------------------------------
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 25          |
|    ep_rew_mean          | -1.73e+03   |
| time/                   |             |
|    fps                  | 0           |
|    iterations           | 3           |
|    time_elapsed         | 4884        |
|    total_timesteps      | 3840        |
| train/                  |             |
|    approx_kl            | 0.006924638 |
|    clip_fraction        | 0.0556      |
|    clip_range           | 0.2         |
|    entropy_loss         | -9.91       |
|    explained_variance   | -0.000117   |
|    learning_rate        | 0.0003      |
|    loss                 | 4.61e+05    |
|    n_updates            | 20          |
|    policy_gradient_loss | -0.0122     |
|    std                  | 0.996       |
|    value_loss           | 1e+06       |
-----------------------------------------

[Intermediate PPO training log excerpts]
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 25          |
|    ep_rew_mean          | -130        |
| time/                   |             |
|    fps                  | 0           |
|    iterations           | 155         |
```

```
|    time_elapsed         | 540348      |
|    total_timesteps      | 198400      |
| train/                  |             |
|    approx_kl            | 0.006647116 |
|    clip_fraction        | 0.0748      |
|    clip_range           | 0.2         |
|    entropy_loss         | -8.58       |
|    explained_variance   | 0.0242      |
|    learning_rate        | 0.0003      |
|    loss                 | 6.97e+03    |
|    n_updates            | 1540        |
|    policy_gradient_loss | -0.00826    |
|    std                  | 0.881       |
|    value_loss           | 4.13e+04    |
-----------------------------------------
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 25          |
|    ep_rew_mean          | -150        |
| time/                   |             |
|    fps                  | 0           |
|    iterations           | 156         |
|    time_elapsed         | 541909      |
|    total_timesteps      | 199680      |
| train/                  |             |
|    approx_kl            | 0.0064748637 |
|    clip_fraction        | 0.0619      |
|    clip_range           | 0.2         |
|    entropy_loss         | -8.58       |
|    explained_variance   | 0.0578      |
|    learning_rate        | 0.0003      |
|    loss                 | 9.17e+03    |
|    n_updates            | 1550        |
|    policy_gradient_loss | -0.00811    |
|    std                  | 0.882       |
|    value_loss           | 1.51e+04    |
-----------------------------------------
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 25          |
|    ep_rew_mean          | -160        |
| time/                   |             |
|    fps                  | 0           |
|    iterations           | 157         |
|    time_elapsed         | 543487      |
|    total_timesteps      | 200960      |
| train/                  |             |
|    approx_kl            | 0.006274433 |
|    clip_fraction        | 0.0381      |
|    clip_range           | 0.2         |
|    entropy_loss         | -8.57       |
|    explained_variance   | 0.0487      |
|    learning_rate        | 0.0003      |
|    loss                 | 1.85e+04    |
```

```
|    n_updates            | 1560        |
|    policy_gradient_loss | -0.00675    |
|    std                  | 0.881       |
|    value_loss           | 4.97e+04    |
-----------------------------------------

[final PPO training log excerpts]
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 25          |
|    ep_rew_mean          | -100        |
| time/                   |             |
|    fps                  | 0           |
|    iterations           | 18          |
|    time_elapsed         | 28854       |
|    total_timesteps      | 23040       |
| train/                  |             |
|    approx_kl            | 0.0081445305 |
|    clip_fraction        | 0.0932      |
|    clip_range           | 0.2         |
|    entropy_loss         | -8.61       |
|    explained_variance   | -0.0113     |
|    learning_rate        | 0.0003      |
|    loss                 | 1.47e+04    |
|    n_updates            | 1740        |
|    policy_gradient_loss | -0.00737    |
|    std                  | 0.889       |
|    value_loss           | 1.64e+04    |
-----------------------------------------
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 25          |
|    ep_rew_mean          | -130        |
| time/                   |             |
|    fps                  | 0           |
|    iterations           | 19          |
|    time_elapsed         | 30416       |
|    total_timesteps      | 24320       |
| train/                  |             |
|    approx_kl            | 0.0077872844 |
|    clip_fraction        | 0.0603      |
|    clip_range           | 0.2         |
|    entropy_loss         | -8.6        |
|    explained_variance   | -0.00247    |
|    learning_rate        | 0.0003      |
|    loss                 | 1.94e+04    |
|    n_updates            | 1750        |
|    policy_gradient_loss | -0.00926    |
|    std                  | 0.886       |
|    value_loss           | 1.91e+04    |
-----------------------------------------
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 25          |
```

```
|    ep_rew_mean          | -70         |
| time/                   |             |
|    fps                  | 0           |
|    iterations           | 20          |
|    time_elapsed         | 31998       |
|    total_timesteps      | 25600       |
| train/                  |             |
|    approx_kl            | 0.008912076 |
|    clip_fraction        | 0.0765      |
|    clip_range           | 0.2         |
|    entropy_loss         | -8.58       |
|    explained_variance   | 0.0276      |
|    learning_rate        | 0.0003      |
|    loss                 | 8.57e+03    |
|    n_updates            | 1760        |
|    policy_gradient_loss | -0.00755    |
|    std                  | 0.885       |
|    value_loss           | 2.43e+04    |
-----------------------------------------
```

9 Discussion of Results

The results obtained from training the PPO agent did not converge to a hovering trajectory. Instead, the learned policies predominantly shifted the mean wing angle while leaving the flapping amplitude and frequency parameters nearly unchanged throughout training. This behavior is consistent with the structure of the reward function and its interaction with the action parameterization.

Although the action vector supplied to the agent includes parameters that control flapping amplitude, flapping frequency, and mean pitching angles, the reward function is highly sensitive to only a subset of these variables. In particular, the reward penalizes deviations of the body's center of mass from the desired target location at each timestep. Small changes to the mean wing angles can produce an immediate and consistent change in the net aerodynamic force direction. In contrast, changes to amplitude or frequency require coordinated, time-dependent interactions between both wings before producing measurable effects on the vertical force. As a result, the reward landscape is much steeper with respect to mean-angle adjustments than with respect to oscillatory parameters such as amplitude and frequency. The PPO update, which relies on the gradient of the expected return, therefore receives a much stronger learning signal in the directions associated with mean offsets than in those associated with oscillatory control.

This imbalance effectively biases the agent toward exploiting the simplest available mechanism for reducing the instantaneous position error: adjusting the baseline wing orientation. Because amplitude and frequency produce weaker and noisier reward gradients over short episode lengths, the agent gains little useful information about how variations in these parameters affect the long-term hovering performance. Consequently, the policy collapses onto a narrow region of the action space in which only the mean wing angles are significantly varied, while the amplitude and frequency parameters remain near their initial values. This phenomenon is further reinforced by PPO's clipping objective, which prevents large exploratory updates in directions where the reward gradient is weak or dominated by variance.

In summary, the reward function as currently designed does not sufficiently incentivize exploration of the full action space, particularly with respect to parameters that influence force production only indirectly or over extended timescales. As a result, the learned policies cannot discover flapping motions capable of generating sustained vertical lift, and the system fails to converge to a hovering solution. A more effective reward structure would need to provide clear credit assignment for improvements in vertical force production over an entire flapping cycle, or explicitly reward aerodynamic efficiency and lift generation associated with amplitude and frequency modulation. Such modifications would likely improve the agent's ability to explore and exploit the full range of physically relevant wing kinematics.

10 Conclusions

This project represents my first experience applying Reinforcement Learning to a dynamic, physics-driven control problem, and it has been a significant learning curve. Coming from a background in mechanical and aerospace engineering, much of my previous work has been rooted in deterministic modeling, rigid-body dynamics, and numerical simulation. Integrating computer science concepts—particularly policy-based learning, stochastic optimization, and reward shaping—required me to step outside my usual analytical framework and adopt new ways of thinking about control and system behavior. In that sense, the project has been as much an educational journey as a technical one.

Although the PPO agent did not ultimately achieve stable hover, the partial progress it exhibited demonstrates that the learning pipeline is capable of capturing nontrivial aerodynamic relationships. The agent began to exploit consistent trends in the reward landscape, even if those trends were not aligned with the physical behaviors required for sustained lift. It is clear from the results and from the diagnostic analyses that the limitations arose primarily from the formulation of the reward function and the structure of the action space, rather than from a fundamental inability of reinforcement learning to model flapping flight. With additional time to redesign the reward shaping, balance the action parameterization, and tune the PPO hyperparameters, I am confident that substantially better performance could be achieved. Such improvements would not only enable hover learning but would also open the door to more advanced tasks, such as stabilization and trajectory tracking.

Despite its imperfections, this work establishes a foundation for future research at the intersection of reinforcement learning and bio-inspired flight. Biological flyers generate lift and maneuver through complex,

time-varying sensorimotor processes that are difficult to express analytically but are well suited to learning-based approaches. Even the simplified model used here begins to illustrate how an artificial agent can infer wingbeat strategies without explicitly programmed aerodynamic rules. With further development, these methods could eventually approximate the neural control strategies insects and birds use to coordinate flapping motion, adapt to disturbances, and follow dynamic trajectories. As such, this project is a positive step toward a broader exploration of how modern learning techniques can deepen our understanding of natural flight and contribute to the design of agile, autonomous micro air vehicles.

11 Recommendations

The failure of the current PPO implementation to generate flapping kinematics capable of sustained hover highlights several avenues for improvement in the learning architecture. The following recommendations outline potential modifications that may enable more effective exploration of the action space and better credit assignment for aerodynamic performance.

Reward Function Redesign

A central limitation of the current training setup is that the reward signal is dominated by instantaneous position error. Because this error responds strongly to small changes in mean wing angle but only weakly to amplitude or frequency changes, the agent learns to exploit the variable with the steepest and least ambiguous reward gradient. To counteract this imbalance, future work should consider:

- **Shaping Terms for Aerodynamic Performance:** Adding explicit rewards for generating upward force, maintaining symmetry, or matching desired force–phase profiles may encourage exploration of amplitude and frequency parameters.
- **Penalizing Policy Stagnation:** Introducing entropy bonuses or anti-collapse penalties can help prevent premature convergence onto mean-angle-only solutions.

PPO Hyperparameter Modifications

Even with improved rewards and action structures, PPO may require hyperparameter adjustments to adequately explore the aerodynamic control space:

- **Increase Entropy Coefficient:** Strengthening the entropy bonus encourages broader exploration and reduces premature fixation on mean-angle strategies.
- **Reduce Clipping Range:** A tighter clipping threshold can prevent destructive updates when reward gradients become noisy or uninformative.
- **Increase Rollout Length:** Larger values of n_{steps} provide the agent with longer temporal dependencies, enabling it to observe the delayed aerodynamic effects of amplitude or frequency changes.
- **Use Smaller Learning Rates:** Reducing the policy learning rate can help stabilize training when gradients vary strongly across action dimensions.

Overall, a combination of reward shaping, improved action representation, carefully tuned hyperparameters, and potentially more expressive policy architectures offers a promising path toward learning wing kinematics capable of stable hover.

12 References

[1] A. Andersen, U. Pesavento, and Z. Jane Wang. "Analysis of transitions between fluttering, tumbling and steady descent of falling cards". In: *Journal of Fluid Mechanics* 541 (2005), pp. 91–104. DOI: 10.1017/S0022112005005847. URL: https://doi.org/10.1017/S0022112005005847.

[2] A. Andersen, U. Pesavento, and Z. Jane Wang. "Unsteady aerodynamics of fluttering and tumbling plates". In: *Journal of Fluid Mechanics* 541 (2005), pp. 65–90. DOI: 10.1017/S002211200500594X. URL: https://doi.org/10.1017/S002211200500594X.

[3] Farama Foundation. *Gymnasium Documentation*. Accessed: 2025-12-14. 2025. URL: https://gymnasium.farama.org/index.html.

[4] NumPy Developers. *NumPy Reference — NumPy v1.x Documentation*. Accessed: 2025-12-14. 2025. URL: https://numpy.org/doc/stable/reference/index.html#reference.

[5] Stable-Baselines3 Contributors. *Stable-Baselines3 Documentation*. Accessed: 2025-12-14. 2025. URL: https://stable-baselines3.readthedocs.io/en/master/.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd. Cambridge, MA: MIT Press, 2018. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[7] SymPy Development Team. *SymPy Documentation: Reference Guide*. Accessed: 2025-12-14. 2025. URL: https://docs.sympy.org/latest/reference/index.html#reference.

[8] Z. Jane Wang. "Dissecting Insect Flight". In: *Annual Review of Fluid Mechanics* 37 (2005), pp. 183–210. DOI: 10.1146/annurev.fluid.36.050802.121940.

[9] Z. Jane Wang. "Vortex shedding and lift in flapping flight". In: *Journal of Fluid Mechanics* 410 (2000), pp. 323–341. DOI: 10.1017/S0022112099008182. URL: https://dragonfly.tam.cornell.edu/publications/2000_JFM_Wang_FlpS.pdf.

A Code

A.1 Simulation Code

A.1.1 main.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: albertaddo
"""

import sympy as sp
import numpy as np
from numpy import pi
from funcs import *
from scipy.integrate import solve_ivp
from scipy.integrate import quad
import matplotlib.pyplot as plt

plt.close('all')

# Defining on dimensionless constants
ndc = {"CT": 2.5, "CR": np.pi, "A_CD": 1.4, "B_CD": 1.0, "Pi1": 0.23, "Pi2": 4.5,
       "pi2": 0.37, "pi1": sp.symbols('pi1', real = True, positive = True), "pi3":
           0.2, "pi4": 0.25}
l1 = 0.006
g = 9.81

knd = .3
cnd = .2
l0 = 3


t = sp.symbols('t', real = True, positive = True)
phi = sp.Function('phi', real = True)(t)
y = sp.Function('y', real = True)(t)
z = sp.Function('z', real = True)(t)
psi = sp.Function('psi', real = True)(t)

theta = pi/2
thetad = 0
thetadd = 0
phid = sp.diff(phi, t)
phidd = sp.diff(phid, t)
psid = sp.diff(psi, t)
psidd = sp.diff(psid, t)
yd = sp.diff(y, t)
ydd = sp.diff(yd, t)
zd = sp.diff(z, t)
zdd = sp.diff(zd, t)

b1 = sp.Matrix([[1],[0],[0]])
b2 = sp.Matrix([[0],[1],[0]])
c1 = sp.Matrix([[1],[0],[0]])
c2 = sp.Matrix([[0],[1],[0]])
d1 = sp.Matrix([[1],[0],[0]])
d2 = sp.Matrix([[0],[1],[0]])
d3 = sp.Matrix([[0],[0],[1]])
```

```
ey = sp.Matrix([[0],[1],[0]])
ez = sp.Matrix([[0],[0],[1]])

omega_I_B = sp.diff(theta,t) * b1
omega_I_C = transBC(phi, omega_I_B) - phid * c2
omega_I_D = transCD(psi, omega_I_C) + psid*d1

term1_C = ndc["pi1"] * (omega_I_C.cross(c1))
# print(term1_C)
term2_B = omega_I_B.cross(ndc["pi2"]*b2 + ndc["pi3"]*b1)
term3_I = yd*ey+zd*ez
# term3_I = sp.Matrix([[0],[yd],[zd]])

term1_D = transCD(psi,term1_C)
term2_D = transCD(psi, transBC(phi, term2_B))
term3_D = transCD(psi, transBC(phi, transIB(theta, term3_I)))
v_D_dw0 = sp.simplify(term1_D + term2_D + term3_D)
# print(v_D_dw0)

v_D_dw0_simplified = sp.simplify(sp.trigsimp(v_D_dw0))
# print(v_D_dw0_simplified)


# Compute projections of v_D_dw0 on d2 and d3 in D-frame
v_dot_d2 = v_D_dw0_simplified[1]  # component along d2
v_dot_d3 = v_D_dw0_simplified[2]  # component along d3
# print(v_dot_d2, "|||", v_dot_d3)

# U_vec_dw(s) in D-frame
U_vec_dw = sp.simplify(v_dot_d2 * d2 + v_dot_d3 * d3)

# Matrix R
R = sp.Matrix([
    [1, 0,  0],
    [0, 0, -1],
    [0, 1,  0]
])

I3 = sp.eye(3)

U_mag = sp.sqrt(U_vec_dw.dot(U_vec_dw))+1e-9

U_mag_func = sp.lambdify((t, y, z, yd, zd, phi, phid, phidd, ndc["pi1"]), U_mag, "
    numpy")

Gamma = -(ndc["CT"]*U_vec_dw[1]*U_vec_dw[2])/U_mag+((ndc["CR"]*ndc["pi2"])/2)*
    omega_I_D[0]


CD = ndc["A_CD"] - ndc["B_CD"]*((U_vec_dw[1]**2 - (U_vec_dw[2]**2))/U_mag**2)

faero = transIB(theta,transBC(phi,transCD(psi, ndc["Pi2"]*(Gamma*R-CD*U_mag*sp.eye
    (3)/2)*U_vec_dw, 'D'),'C'),'B')
f_sy = sp.lambdify((t, y, z, yd, zd, phi, phid, phidd, psi, psid, psidd, ndc["pi1"
    ]), faero[1], "numpy")
f_sz = sp.lambdify((t, y, z, yd, zd, phi, phid, phidd, psi, psid, psidd, ndc["pi1"
    ]), faero[2], "numpy")

#Equations of motion
```

```
# ddy = faero[1];
# ddz = -ndc["Pi1"]*(.5*(-sp.sin(phi)*phid**2+sp.cos(phi)*phidd**2)-\
#                    ndc["pi4"]*(-sp.sin(theta)*thetad**2+sp.cos(theta)*thetadd
    **2))-1+faero[2]
ddzna = -ndc["Pi1"]*(.5*(-sp.cos(theta)*sp.sin(phi)*phid**2+sp.cos(theta)*sp.cos(
    phi)*phidd)+\
                     ndc["pi4"]*(-sp.sin(theta)*thetad**2+sp.cos(theta)*thetadd))-1
ddyna = -ndc["Pi1"]*(-ndc["pi4"]*thetadd*sp.sin(theta)-ndc["pi4"]*thetad**2*sp.cos
    (theta)-\
                      .5*(phidd*sp.sin(phi)+phid**2*sp.cos(phi)))


# After defining symbolic ddy and ddz:
ddzna_simplified = sp.simplify(ddzna)
ddyna_simplified = sp.simplify(ddyna)

# Convert to numeric callable functions
vars_ = (t, y, z, yd, zd, phi, phid, phidd)
ddzna_func = sp.lambdify(vars_, ddzna_simplified, modules='numpy')
ddyna_func = sp.lambdify(vars_, ddyna_simplified, modules='numpy')


# phi(t) input
A1 = np.deg2rad(60)
# w1 = (2 * np.pi * 1802.5/60)/np.sqrt(g/l1)
w1 = (2 * np.pi * 1695/60)/np.sqrt(g/l1)
psimax = np.deg2rad(10.05)*2
psimin = np.deg2rad(-10.05)*2
A_psi = (psimax-psimin)/2
phic = (psimax+psimin)/2


def phi_input(t):
    return A1 * np.cos(w1*t) + np.deg2rad(-13.5);

def phi_d_input(t):
    return -A1 * (w1*np.sin(w1*t))

def phi_dd_input(t):
    return -A1 * w1**2 * np.cos(w1*t)

def psi_input(t):
    return -A_psi * np.sin(w1*t) + phic;

def psi_d_input(t):
    return -A_psi * (w1*np.cos(w1*t))

def psi_dd_input(t):
    return A_psi * w1**2 * np.sin(w1*t)
# ----------------------

# RHS for solver
def rhs(t, Y):
    y, z, dy, dz = Y
    ph, phd, phdd = phi_input(t), phi_d_input(t), phi_dd_input(t)
    ps, psd, psdd = psi_input(t), psi_d_input(t), psi_dd_input(t)
    ddy = ddyna_func(t, y, z, dy, dz, ph, phd, phdd) + \
          quad(lambda s: f_sy(t, y, z, dy, dz, ph, phd, phdd, ps, psd, psdd, s),
              0, 1)[0]
```

```
    ddz = ddzna_func(t, y, z, dy, dz, ph, phd, phdd) + \
          quad(lambda s: f_sz(t, y, z, dy, dz, ph, phd, phdd, ps, psd, psdd, s),
              0, 1)[0]

    # print("Z-Vel: ", dz*np.sqrt(g*l1))
    # print("Y-Vel: ", dy*np.sqrt(g*l1))
    # print("Z-Acc: ", ddy*g)
    # print("Y-Acc: ", ddz*g)
    # # print("Aero Force Acceleration: ",quad(lambda s: f_sz(t, y, z, dy, dz, ph,
        phd, phdd, s), 0, 1)[0]*g)
    # print("t: ", t*np.sqrt(l1/g))
    # print(ddy,ddz)
    # zdd = ddz_func(t, y, z, yd, zd, ph, phd, phdd)

    return [float(dy), float(dz), ddy, ddz]

def rhs2(t, Y):
    z, zd = Y
    ph = phi_input(t)
    phd = phi_d_input(t)
    phdd = phi_dd_input(t)
    zdd = ddzna_func(t, y, z, yd, zd, ph, phd, phdd)
    # zdd=-1
    # if not np.isfinite(zdd) or abs(zdd)>1e6: zdd=0
    # if np.isnan(ydd) or np.isnan(zdd):
    #     ydd, zdd = 0, 0  # prevent propagation of NaN
    return [zd, zdd]

def rhs3(t, X):
    x,xd = X;
    xdd = -cnd*xd-knd*(x-1)
    return [xd, xdd]
# Integrate

X0 = [6/10, 0]

# Y0 = [0, 0, -.36/np.sqrt(g*l1), 0]
Y0 = [0, 0, 0, 0]
t_span = (0,5/np.sqrt(l1/g))
t_eval = np.linspace(*t_span, 2000)

Z0 = [Y0[1],Y0[3]]
sol = solve_ivp(rhs,
                t_span,
                Y0,
                t_eval=t_eval,
                method='RK45',
                rtol=1e-4,
                atol=1e-6)
                # max_step = 1e-6)

na_sol = solve_ivp(rhs2,
                   t_span,
                   Z0,
                   t_eval=t_eval,
                   method='RK45',
                   rtol=1e-6,
                   atol=1e-8)
                 # max_step = 5e-5)
```

```
219
220
221 plt.figure(figsize=(8, 4))
222 plt.plot(sol.t*np.sqrt(l1/g), sol.y[0]*l1, label='y(t)')
223 plt.xlabel('Time [s]')
224 plt.ylabel('Y-Displacement [m]')
225 plt.title('Y-Position Over Time')
226 plt.grid(True)
227
228 # print(sol.y[1])
229 plt.figure(figsize=(8, 4))
230 plt.plot(sol.t*np.sqrt(l1/g), sol.y[1]*l1, label='z(t)')
231 # plt.plot(na_sol.t*np.sqrt(l1/g), na_sol.y[0]*l1, label = 'z(t) with No Aero
    Force')
232 plt.xlabel('Time [s]')
233 plt.ylabel('Z-Displacement [m]')
234 plt.title('Z-Position Over Time')
235 plt.legend()
236 plt.grid(True)
237
238 # print(sol.y[1])
239 plt.figure(figsize=(8, 4))
240 plt.plot(sol.y[0]*l1, sol.y[1]*l1)
241 plt.xlabel('Y [m]')
242 plt.ylabel('Z [m]')
243 plt.title('Trajectory')
244 plt.grid(True)
245 plt.xlim(-.5, .5)
246 plt.ylim(-.5,.5)
247
248 # plt.figure(figsize=(8, 4))
249 # plt.plot(sol.t*np.sqrt(l1/g), phi_input(sol.t), label='phi')
250 # plt.xlabel('Time [s]')
251 # plt.ylabel('$\phi (t)$ [radians]')
252 # plt.title('Wing Beat Angle Over Time')
253 # plt.grid(True)
254 # plt.show()
255
256 # plt.figure(figsize=(8, 4))
257 # plt.plot(sol.t*np.sqrt(l1/g), psi_input(sol.t), label='psi')
258 # plt.xlabel('Time [s]')
259 # plt.ylabel('$\psi (t)$ [radians]')
260 # plt.title('Wing Pitch Angle Over Time')
261 # plt.grid(True)
262 # plt.show()
263
264 # plt.figure(figsize=(8, 4))
265 # plt.plot(osc_sol.t*np.sqrt(l0/g), osc_sol.y[0]*l0, label='x')
266 # plt.xlabel('Time [s]')
267 # plt.ylabel('X(t)[meters]')
268 # plt.title('Simple Harmonic Oscillator')
269 # plt.grid(True)
270 # plt.show()
```

A.2 funcs.py

```
1 #!/usr/bin/env python3
```

```
# -*- coding: utf-8 -*-
"""
Albert Addo
Professor Jane Wang Advisor
Equation of Motion Validation Using One Blade Element
Helper Function Module
"""
import sympy as sp
import numpy as np

# Defining on dimensionless constants
ndc = {"CT": 2.5, "CR": 0.05, "A_CD": 1.3, "B_CD": 0.7, "Pi1": 0.4, "Pi2": 0.1,
       "pi2": 0.25, "pi1": 1/2}

# Define symbolic rotation angles
theta, phi, psi, z, y = sp.symbols('theta phi psi z y', real=True)

# Define DCMs
BNI = sp.Matrix([
    [1,              0,              0],
    [0,  sp.cos(theta),  sp.sin(theta)],
    [0, -sp.sin(theta),  sp.cos(theta)]
])

CNB = sp.Matrix([
    [ sp.cos(phi), 0, sp.sin(phi)],
    [0,            1,           0],
    [-sp.sin(phi), 0, sp.cos(phi)]
])

DNC = sp.Matrix([
    [0,  sp.cos(psi),  sp.sin(psi)],
    [1,           0,           0],
    [0, -sp.sin(psi),  sp.cos(psi)]
])

# Conducting tranformation operations
def transIB(theta, input_data=None, frame='I', direction=None, scalar=None):
    """
    Transform between Inertial (I) and Body (B) frames.
    Either:
      - Mode (a): transIB(scalar=s, frame='I', direction=1)  -> returns 3x1
      vector
      - Mode (b): transIB(vector, frame='I')  -> transforms vector to other
      frame
    """
    BNI = sp.Matrix([
        [1,              0,              0],
        [0,  sp.cos(theta),  sp.sin(theta)],
        [0, -sp.sin(theta),  sp.cos(theta)]
    ])
    if scalar is not None:
        vec = sp.zeros(3, 1)
        vec[direction - 1] = scalar
        input_data = vec

    if frame == 'I':   # I -> B
        return BNI * input_data
    elif frame == 'B': # B -> I
```

```
        return BNI.T * input_data
    else:
        raise ValueError("Frame must be 'I' or 'B'.")


def transBC(phi, input_data=None, frame='B', direction=None, scalar=None):
    """
    Transform between Body (B) and Wing (C) frames.
    """
    CNB = sp.Matrix([
        [ sp.cos(phi), 0, sp.sin(phi)],
        [0,            1,           0],
        [-sp.sin(phi), 0, sp.cos(phi)]
    ])
    if scalar is not None:
        vec = sp.zeros(3, 1)
        vec[direction - 1] = scalar
        input_data = vec

    if frame == 'B':   # B -> C
        return CNB * input_data
    elif frame == 'C': # C -> B
        return CNB.T * input_data
    else:
        raise ValueError("Frame must be 'B' or 'C'.")


def transCD(psi, input_data=None, frame='C', direction=None, scalar=None):
    """
    Transform between Wing (C) and D frames.
    """
    DNC = sp.Matrix([
        [0,  sp.cos(psi),  sp.sin(psi)],
        [1,           0,           0],
        [0, -sp.sin(psi),  sp.cos(psi)]
    ])

    if scalar is not None:
        vec = sp.zeros(3, 1)
        vec[direction - 1] = scalar
        input_data = vec

    if frame == 'C':   # C -> D
        return DNC * input_data
    elif frame == 'D': # D -> C
        return DNC.T * input_data
    else:
        raise ValueError("Frame must be 'C' or 'D'.")
```

A.3 PPO Code

A.3.1 train_ppo.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Albert Addo
Under Professor Jane Wang
```

```
Using Reinforcement Learning PPO Model to Identify Wing Beating Motion

This script begins or continues the PPO training.
"""

import numpy as np
from stable_baselines3 import PPO
from rollout_utils import rollout_and_collect
from flapping_env import FlappingEnv
import matplotlib.pyplot as plt
import os

def build_dummy_target(l1=0.006, g=9.81):
    t0 = 0.0
    t1 = 2.5 / np.sqrt(l1 / g)
    t_eval = np.linspace(t0, t1, 1000)
    # example: hover at origin
    y_ref = np.zeros_like(t_eval)
    z_ref = np.zeros_like(t_eval)
    return t_eval, y_ref, z_ref

if __name__ == "__main__":
    target_times, target_y, target_z = build_dummy_target()

    env = FlappingEnv(
        target_times=target_times,
        target_y=target_y,
        target_z=target_z,
        dt=None,              # default: (tf - t0)/max_steps
        max_steps=25,
        transition_duration=0.002
    )

    MODEL_PATH = "hope_this_works"

    if os.path.exists(MODEL_PATH + ".zip"):
        print("Loading existing model...")
        model = PPO.load(MODEL_PATH, env=env)
    else:
        print("Creating new model...")
        model = PPO(
            "MlpPolicy",
            env,
            verbose=1,
            batch_size=64,
            n_steps=64*20,
            gamma=0.99,
            learning_rate=0.0003
        )

    model.learn(total_timesteps=25_600)
    model.save(MODEL_PATH)



t_all, y_all, z_all, phi_all, psi_all = rollout_and_collect(env, model)

# plot or print results

```

```
plt.figure()
plt.plot(t_all*np.sqrt(env.l1/env.g), y_all*env.l1, label="y(t)")
plt.plot(t_all*np.sqrt(env.l1/env.g), z_all*env.l1, label="z(t)")
plt.xlabel("Time (s)")
plt.ylabel("Displacement (m)")
plt.legend()

plt.figure()
plt.plot(t_all*np.sqrt(env.l1/env.g), phi_all*180/np.pi, label="phi(t)")
plt.plot(t_all*np.sqrt(env.l1/env.g), psi_all*180/np.pi, label="psi(t)")
plt.ylabel("Angle (degrees)")
plt.xlabel("Time (s)")
plt.title("Wing Angles Over Time")
plt.legend()

plt.figure()
plt.plot(y_all*env.l1, z_all*env.l1)
plt.title("Trajectory")
plt.xlabel("Y-Displacement")
plt.ylabel("Z-Displacement")
plt.show()
```

A.3.2 flapping_env.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Albert Addo
Professor Jane Wang
Using Reinforcement Learning PPO Model to Identify Wing Beating Motion

This Module Holds the Flapping Environment Class
"""

import gym
import numpy as np
from numpy import pi
import sympy as sp
from scipy.integrate import solve_ivp, quad
from wing import Wings


# ---------- linear least-squares helper (like your CO2 code, but linear) ----------

def linear_least_squares(x, y, std=None):
    """
    Weighted linear least-squares fit:
        y     a0 + a1 * x

    Robust to degenerate cases:
      - If all x are (effectively) equal or the normal matrix is singular,
        falls back to a constant fit: a1 = 0, a0 = weighted mean(y).
    """
    x = np.asarray(x, dtype=float)
    y = np.asarray(y, dtype=float)

    if std is None:
```

```
        w = np.ones_like(y)
    else:
        std = np.asarray(std, dtype=float)
        w = 1.0 / (std * std)

    # If lengths don't match or too few points, just return a flat fit
    if x.size != y.size or x.size < 2:
        a0 = np.average(y) if y.size > 0 else 0.0
        a1 = 0.0
        return a0, a1

    # Shift x to improve conditioning (optional but nice)
    x_mean = np.average(x, weights=w)
    x_rel = x - x_mean

    S_w   = np.sum(w)
    S_wx  = np.sum(w * x_rel)
    S_wx2 = np.sum(w * x_rel * x_rel)
    S_wy  = np.sum(w * y)
    S_wxy = np.sum(w * x_rel * y)

    F = np.array([[S_w,  S_wx],
                  [S_wx, S_wx2]])
    b = np.array([S_wy, S_wxy])

    # Check for singular or near-singular matrix
    det = F[0, 0] * F[1, 1] - F[0, 1] * F[1, 0]
    if abs(det) < 1e-12:
        # Fall back: constant fit (slope 0), intercept = weighted mean of y
        a1 = 0.0
        a0 = S_wy / S_w if S_w > 0 else 0.0
        return a0, a1

    # Otherwise solve normally
    a0_rel, a1 = np.linalg.solve(F, b)
    # Adjust intercept back to original x (undo centering)
    a0 = a0_rel - a1 * x_mean

    return a0, a1


# ---------- transforms (same as before) ----------

def transIB(theta, input_data=None, frame='I', direction=None, scalar=None):
    BNI = sp.Matrix([
        [1,              0,              0],
        [0,  sp.cos(theta),  sp.sin(theta)],
        [0, -sp.sin(theta),  sp.cos(theta)]
    ])
    if scalar is not None:
        vec = sp.zeros(3, 1)
        vec[direction - 1] = scalar
        input_data = vec

    if frame == 'I':
        return BNI * input_data
    elif frame == 'B':
        return BNI.T * input_data
    else:
```

```
        raise ValueError("Frame must be 'I' or 'B'.")


def transBC(phi, input_data=None, frame='B', direction=None, scalar=None):
    CNB = sp.Matrix([
        [ sp.cos(phi), 0, sp.sin(phi)],
        [0,            1,           0],
        [-sp.sin(phi), 0, sp.cos(phi)]
    ])
    if scalar is not None:
        vec = sp.zeros(3, 1)
        vec[direction - 1] = scalar
        input_data = vec

    if frame == 'B':
        return CNB * input_data
    elif frame == 'C':
        return CNB.T * input_data
    else:
        raise ValueError("Frame must be 'B' or 'C'.")


def transCD(psi, input_data=None, frame='C', direction=None, scalar=None):
    DNC = sp.Matrix([
        [0,  sp.cos(psi),  sp.sin(psi)],
        [1,           0,           0],
        [0, -sp.sin(psi),  sp.cos(psi)]
    ])
    if scalar is not None:
        vec = sp.zeros(3, 1)
        vec[direction - 1] = scalar
        input_data = vec

    if frame == 'C':
        return DNC * input_data
    elif frame == 'D':
        return DNC.T * input_data
    else:
        raise ValueError("Frame must be 'C' or 'D'.")


class FlappingEnv(gym.Env):
    """
    Time-stepping RL environment:

    - Global reference trajectory: target_times, target_y, target_z.
    - Episode runs from t0 = target_times[0] to tf = target_times[-1].
    - Each env.step(action) advances from t_curr to t_curr + dt (or tf).
    - Action = [Aphi, w, phi_c, delta_phi, Apsi, psi_c, delta_psi].
    - Wings.set_parameters_with_transition enforces continuity and
      smooth transitions in   ,    between steps.
    - Reward is computed from linear least-squares fits of y,z over
      the interval vs the target y,z over the same interval.
    """

    metadata = {"render.modes": []}

    def __init__(self,
                 target_times,
```

```
                target_y,
                target_z,
                dt=None,
                max_steps=200,
                transition_duration=0.002):
        super().__init__()

        self.target_times = np.asarray(target_times, dtype=float)
        self.target_y = np.asarray(target_y, dtype=float)
        self.target_z = np.asarray(target_z, dtype=float)

        self.l1 = 0.006
        self.g = 9.81

        self.ndc = {
            "CT": 2.5,
            "CR": np.pi,
            "A_CD": 1.4,
            "B_CD": 1.0,
            "Pi1": 0.23,
            "Pi2": 4.5,
            "pi2": 0.37,
            "pi1": sp.symbols('pi1', real=True, positive=True),
            "pi3": 0.2,
            "pi4": 0.25
        }

        self._build_aero_model()

        # Episode time settings
        self.t0 = float(self.target_times[0])
        self.tf = float(self.target_times[-1])

        if dt is None:
            # default: split into max_steps equal segments
            self.dt = (self.tf - self.t0) / max_steps
        else:
            self.dt = float(dt)

        self.max_steps = max_steps
        self.transition_duration = transition_duration

        # Action: [Aphi, w, phi_c, delta_phi, Apsi, psi_c, delta_psi]
        self.action_space = gym.spaces.Box(
            low=np.array([
                np.deg2rad(40),
                (2 * np.pi * 1500/60)/np.sqrt(self.g/self.l1),
                np.deg2rad(-45),
                -np.pi/2,
                np.deg2rad(2),
                np.deg2rad(-30),
                -np.pi/2
            ], dtype=np.float32),
            high=np.array([
                np.deg2rad(90),
                (2 * np.pi * 2400/60)/np.sqrt(self.g/self.l1),
                np.deg2rad(45),
                np.pi/2,
                np.deg2rad(30),
```

```
                np.deg2rad(30),
                np.pi/2
            ], dtype=np.float32),
            dtype=np.float32
        )



        # Observation: [y, z, dy, dz, t_normalized]
        self.observation_space = gym.spaces.Box(
            low=-np.inf, high=np.inf, shape=(5,), dtype=np.float32
        )

        self.wings = Wings()

        self._first_action = True
        self.t_curr = self.t0
        self.state = np.zeros(4)
        self.step_count = 0

    def _build_aero_model(self):
        ndc = self.ndc

        t = sp.symbols('t', real=True, positive=True)
        phi = sp.Function('phi', real=True)(t)
        y = sp.Function('y', real=True)(t)
        z = sp.Function('z', real=True)(t)
        psi = sp.Function('psi', real=True)(t)

        theta = 0
        thetad = 0
        thetadd = 0
        phid = sp.diff(phi, t)
        phidd = sp.diff(phid, t)
        psid = sp.diff(psi, t)
        psidd = sp.diff(psid, t)
        yd = sp.diff(y, t)
        ydd = sp.diff(yd, t)
        zd = sp.diff(z, t)
        zdd = sp.diff(zd, t)

        b1 = sp.Matrix([[1], [0], [0]])
        b2 = sp.Matrix([[0], [1], [0]])
        c1 = sp.Matrix([[1], [0], [0]])
        c2 = sp.Matrix([[0], [1], [0]])
        d1 = sp.Matrix([[1], [0], [0]])
        d2 = sp.Matrix([[0], [1], [0]])
        d3 = sp.Matrix([[0], [0], [1]])
        ey = sp.Matrix([[0], [1], [0]])
        ez = sp.Matrix([[0], [0], [1]])

        omega_I_B = sp.diff(theta, t) * b1
        omega_I_C = transBC(phi, omega_I_B, frame='B') - phid * c2
        omega_I_D = transCD(psi, omega_I_C, frame='C') + psid * d1

        term1_C = ndc["pi1"] * (omega_I_C.cross(c1))
        term2_B = omega_I_B.cross(ndc["pi2"] * b2 + ndc["pi3"] * b1)
        term3_I = yd * ey + zd * ez

```

```
        term1_D = transCD(psi, term1_C, frame='C')
        term2_D = transCD(psi, transBC(phi, term2_B, frame='B'), frame='C')
        term3_D = transCD(psi, transBC(phi, transIB(theta, term3_I, frame='I'),
            frame='B'), frame='C')

        v_D_dw0 = sp.simplify(term1_D + term2_D + term3_D)
        v_D_dw0_simplified = sp.simplify(sp.trigsimp(v_D_dw0))

        v_dot_d2 = v_D_dw0_simplified[1]
        v_dot_d3 = v_D_dw0_simplified[2]

        U_vec_dw = sp.simplify(v_dot_d2 * d2 + v_dot_d3 * d3)

        R = sp.Matrix([
            [1, 0, 0],
            [0, 0, -1],
            [0, 1, 0]
        ])

        U_mag = sp.sqrt(U_vec_dw.dot(U_vec_dw)) + 1e-9

        Gamma = -(ndc["CT"] * U_vec_dw[1] * U_vec_dw[2]) / U_mag + \
            ((ndc["CR"] * ndc["pi2"]) / 2) * omega_I_D[0]

        CD = ndc["A_CD"] - ndc["B_CD"] * ((U_vec_dw[1]**2 - (U_vec_dw[2]**2)) /
            U_mag**2)

        faero = transIB(
            theta,
            transBC(
                phi,
                transCD(
                    psi,
                    ndc["Pi2"] * (Gamma * R - CD * U_mag * sp.eye(3) / 2) *
                        U_vec_dw,
                    frame='C'
                ),
                frame='B'
            ),
            frame='I'
        )

        f_sy = sp.lambdify(
            (t, y, z, yd, zd, phi, phid, phidd, psi, psid, psidd, ndc["pi1"]),
            faero[1],
            "numpy"
        )
        f_sz = sp.lambdify(
            (t, y, z, yd, zd, phi, phid, phidd, psi, psid, psidd, ndc["pi1"]),
            faero[2],
            "numpy"
        )

        ddzna = -ndc["Pi1"] * (
            0.5 * (
                -sp.cos(theta) * sp.sin(phi) * phid**2 +
                sp.cos(theta) * sp.cos(phi) * phidd
            )
            + ndc["pi4"] * (
```

```
                -sp.sin(theta) * thetad**2 +
                sp.cos(theta) * thetadd
            )
        ) - 1

        ddyna = -ndc["Pi1"] * (
            -ndc["pi4"] * thetadd * sp.sin(theta)
            - ndc["pi4"] * thetad**2 * sp.cos(theta)
            - 0.5 * (phidd * sp.sin(phi) + phid**2 * sp.cos(phi))
        )

        ddzna_simplified = sp.simplify(ddzna)
        ddyna_simplified = sp.simplify(ddyna)

        vars_ = (t, y, z, yd, zd, phi, phid, phidd)
        self.ddzna_func = sp.lambdify(vars_, ddzna_simplified, modules='numpy')
        self.ddyna_func = sp.lambdify(vars_, ddyna_simplified, modules='numpy')

        self.f_sy = f_sy
        self.f_sz = f_sz

    # ---------- Gym API ----------

    def reset(self):
        self.wings = Wings()
        self._first_action = True
        self.t_curr = self.t0
        self.state = np.array([0.0, 0.0, 0.0, 0.0])  # y, z, dy, dz
        self.step_count = 0

        return self._get_obs()

    def _get_obs(self):
        y, z, dy, dz = self.state
        t_norm = (self.t_curr - self.t0) / (self.tf - self.t0 + 1e-9)
        return np.array([y, z, dy, dz, t_norm], dtype=np.float32)

    def step(self, action):
        action = np.array(action, dtype=float)
        Aphi, w, phi_c, delta_phi, Apsi, psi_c, delta_psi = action

        t_start = self.t_curr
        t_end = min(self.t_curr + self.dt, self.tf)

        # Set / transition wing parameters
        if self._first_action:
            # First step: no transition, just set base at t_start
            self.wings.set_parameters_with_transition(
                t_k=t_start,
                Aphi=Aphi, w=w, phi_c=phi_c, delta_phi=delta_phi,
                Apsi=Apsi, psi_c=psi_c, delta_psi=delta_psi,
                T_tr=0.0
            )
            self._first_action = False
        else:
            # Subsequent steps: transition smoothly starting at t_start
            self.wings.set_parameters_with_transition(
                t_k=t_start,
                Aphi=Aphi, w=w, phi_c=phi_c, delta_phi=delta_phi,
```

```
                Apsi=Apsi, psi_c=psi_c, delta_psi=delta_psi,
                T_tr=self.transition_duration
            )

        # Integrate over [t_start, t_end] starting from current state
        Y0 = self.state.copy()
        t_span = (t_start, t_end)

        # choose target times in this interval for fitting
        mask = (self.target_times >= t_start) & (self.target_times <= t_end)
        t_eval = self.target_times[mask]
        # ensure at least 2 points
        if t_eval.size < 2:
            t_eval = np.linspace(t_start, t_end, 5)

        sol = solve_ivp(
            fun=self._rhs_wrapper,
            t_span=t_span,
            y0=Y0,
            t_eval=t_eval,
            method='RK45',
            rtol=1e-4,
            atol=1e-6,
        )

        t_seg = np.asarray(sol.t, dtype=float)
        sol_y = np.asarray(sol.y, dtype=float)

        # If the solver basically only gave us one point, treat this as a "failed"
            step
        if t_seg.size < 2 or sol_y.size == 0:
            reward = -1000.0

            # Keep state unchanged
            self.state = Y0
            self.t_curr = t_end
            self.step_count += 1

            obs = self._get_obs()
            done = (self.t_curr >= self.tf - 1e-9) or (self.step_count >= self.
                max_steps)

            # fabricate a tiny segment with constant y,z for logging
            t_fail = np.array([t_start, t_end], dtype=float)
            y_fail = np.array([Y0[0], Y0[0]])
            z_fail = np.array([Y0[1], Y0[1]])
            phi_fail = np.array([self.wings.phi_input(t_start),
                                 self.wings.phi_input(t_end)])
            psi_fail = np.array([self.wings.psi_input(t_start),
                                 self.wings.psi_input(t_end)])

            info = {
                "t_eval": t_fail,
                "y_sim": y_fail,
                "z_sim": z_fail,
                "y_tgt_seg": np.interp(t_fail, self.target_times, self.target_y),
                "z_tgt_seg": np.interp(t_fail, self.target_times, self.target_z),
                "phi_seg": phi_fail,
                "psi_seg": psi_fail,
```

```
                "failure": True,
            }
            return obs, float(reward), done, info


        # Ensure sol_y is 2D: shape (n_states, n_times)
        if sol_y.ndim == 1:
            # We know state dimension is 4 (y,z,dy,dz), so reshape accordingly
            n_states = 4
            sol_y = sol_y.reshape(n_states, -1)

        y_sim = sol_y[0, :]
        z_sim = sol_y[1, :]

        # if you also store dy, dz:
        # dy_sim = sol_y[2, :]
        # dz_sim = sol_y[3, :]

        t_seg = np.asarray(sol.t, dtype=float)

        # Align target on the same t_eval grid
        # If we used target_times slice, target segment is easy:
        if np.array_equal(t_eval, self.target_times[mask]):
            y_tgt_seg = self.target_y[mask]
            z_tgt_seg = self.target_z[mask]
        else:
            # fallback: interpolate target onto t_eval
            y_tgt_seg = np.interp(t_seg, self.target_times, self.target_y)
            z_tgt_seg = np.interp(t_seg, self.target_times, self.target_z)

        # --- least squares fits on this interval ---
        a0_y_sim, a1_y_sim = linear_least_squares(t_seg, y_sim)
        a0_z_sim, a1_z_sim = linear_least_squares(t_seg, z_sim)

        a0_y_tgt, a1_y_tgt = linear_least_squares(t_seg, y_tgt_seg)
        a0_z_tgt, a1_z_tgt = linear_least_squares(t_seg, z_tgt_seg)

        # -------- PLACEHOLDER REWARD FUNCTION (FITS-BASED) ----------
        # You can tune the weights to care more about slope vs intercept.
        w_intercept = 1.0
        w_slope = 10.0
        scale = 1e-13

        dy_int = a0_y_sim - a0_y_tgt
        dy_slp = a1_y_sim - a1_y_tgt
        dz_int = a0_z_sim - a0_z_tgt
        dz_slp = a1_z_sim - a1_z_tgt

        reward = -(
            w_intercept * (dy_int**2 + dz_int**2) +
            w_slope * (dy_slp**2 + dz_slp**2)
        )

        reward *= scale
        # ------------------------------------------------------------

        # Update state/time
        self.state = sol_y[:, -1]  # [y, z, dy, dz] at t_end
```

```
        self.t_curr = t_end
        self.step_count += 1

        done = (self.t_curr >= self.tf - 1e-9) or (self.step_count >= self.
            max_steps)

        obs = self._get_obs()

        # sample phi(t), psi(t) over this interval on the SAME time grid as y_sim/
            z_sim
        phi_seg = np.array([self.wings.phi_input(t) for t in t_seg])
        psi_seg = np.array([self.wings.psi_input(t) for t in t_seg])

        info = {
            "t_eval": t_seg,          # <-- use t_seg, not t_eval
            "y_sim": y_sim,
            "z_sim": z_sim,
            "y_tgt_seg": y_tgt_seg,
            "z_tgt_seg": z_tgt_seg,
            "fit_y_sim": (a0_y_sim, a1_y_sim),
            "fit_z_sim": (a0_z_sim, a1_z_sim),
            "fit_y_tgt": (a0_y_tgt, a1_y_tgt),
            "fit_z_tgt": (a0_z_tgt, a1_z_tgt),
            "phi_seg": phi_seg,
            "psi_seg": psi_seg,
            "action": action
        }

        return obs, float(reward), done, info


    def _rhs_wrapper(self, t, Y):
        return self.rhs(t, Y)

    def rhs(self, t, Y):
        y, z, dy, dz = Y

        ph = self.wings.phi_input(t)
        phd = self.wings.phi_d_input(t)
        phdd = self.wings.phi_dd_input(t)
        ps = self.wings.psi_input(t)
        psd = self.wings.psi_d_input(t)
        psdd = self.wings.psi_dd_input(t)

        ddy_na = self.ddyna_func(t, y, z, dy, dz, ph, phd, phdd)
        ddz_na = self.ddzna_func(t, y, z, dy, dz, ph, phd, phdd)

        f_sy = self.f_sy
        f_sz = self.f_sz

        ddy_aero = quad(
            lambda s: f_sy(t, y, z, dy, dz, ph, phd, phdd, ps, psd, psdd, s),
            0, 1
        )[0]
        ddz_aero = quad(
            lambda s: f_sz(t, y, z, dy, dz, ph, phd, phdd, ps, psd, psdd, s),
            0, 1
        )[0]

```

```
558             ddy = ddy_na + ddy_aero
559             ddz = ddz_na + ddz_aero
560 
561             return np.array([dy, dz, ddy, ddz], dtype=float)
```

A.3.3 wing.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Albert Addo
M. Eng. Project
Professor Jane Wang
Using Reinforcement Learning PPO Model to Identify Wing Beating Motion

This Module Holds the Wing Class which represents the Wing Kinematics.
"""

import numpy as np

class Wings:
    """
    Wing kinematics for   (t) and   (t) with smooth transitions between
    different sinusoid parameter sets.

    Base sinusoids:
        phi_base(t) = phi_c + Aphi * cos(w * t + delta_phi)
        psi_base(t) = psi_c + Apsi * sin(w * t + delta_psi)

    Total signals during a transition:
        phi(t) = phi_base(t) + q_phi(t)
        psi(t) = psi_base(t) + q_psi(t)

    where q_phi and q_psi are quintic polynomials chosen so that   ,     ,
    and   ,     ,      are continuous when parameters change.
    """

    def __init__(self):
        # Base sinusoid parameters (current)
        self.Aphi = 0.0
        self.w = 1.0
        self.phic = 0.0
        self.delta_phi = 0.0

        self.Apsi = 0.0
        self.psic = 0.0
        self.delta_psi = 0.0

        self._has_base = False

        # Transition data for
        self._phi_tr_active = False
        self._phi_tr_t0 = 0.0
        self._phi_tr_T = 1.0
        self._phi_tr_b = np.zeros(6)

        # Transition data for
        self._psi_tr_active = False
        self._psi_tr_t0 = 0.0
        self._psi_tr_T = 1.0
        self._psi_tr_b = np.zeros(6)

    # ---------- base sinusoid ----------

```

```
    def _phi_base(self, t):
        return self.phic + self.Aphi * np.cos(self.w * t + self.delta_phi)

    def _phi_base_d(self, t):
        return -self.Aphi * self.w * np.sin(self.w * t + self.delta_phi)

    def _phi_base_dd(self, t):
        return -self.Aphi * self.w**2 * np.cos(self.w * t + self.delta_phi)

    def _psi_base(self, t):
        return self.psic + self.Apsi * np.sin(self.w * t + self.delta_psi)

    def _psi_base_d(self, t):
        return self.Apsi * self.w * np.cos(self.w * t + self.delta_psi)

    def _psi_base_dd(self, t):
        return -self.Apsi * self.w**2 * np.sin(self.w * t + self.delta_psi)

    # ---------- transition math ----------

    @staticmethod
    def _compute_transition_coeffs(phi_old, phid_old, phdd_old,
                                   phi_base0, phid_base0, phdd_base0,
                                   T_tr):
        D0 = phi_old - phi_base0
        D1 = phid_old - phid_base0
        D2 = phdd_old - phdd_base0

        b0 = D0
        b1 = T_tr * D1
        b2 = 0.5 * T_tr**2 * D2
        b3 = -10*D0 - 6*T_tr*D1 - 1.5*T_tr**2*D2
        b4 =  15*D0 + 8*T_tr*D1 + 1.5*T_tr**2*D2
        b5 =  -6*D0 - 3*T_tr*D1 - 0.5*T_tr**2*D2

        return np.array([b0, b1, b2, b3, b4, b5])

    @staticmethod
    def _q_and_derivs(t, t0, T, b):
        if T <= 0.0 or t <= t0 or t >= t0 + T:
            return 0.0, 0.0, 0.0

        s = (t - t0) / T
        s2 = s*s
        s3 = s2*s
        s4 = s3*s

        b0, b1, b2, b3, b4, b5 = b

        q = b0 + b1*s + b2*s2 + b3*s3 + b4*s4 + b5*s4*s

        q_s_prime = b1 + 2*b2*s + 3*b3*s2 + 4*b4*s3 + 5*b5*s4
        dqdt = q_s_prime / T

        q_s2 = 2*b2 + 6*b3*s + 12*b4*s2 + 20*b5*s3
        d2qdt2 = q_s2 / (T**2)

        return q, dqdt, d2qdt2
```

```
    def set_parameters_with_transition(self, t_k,
                                       Aphi, w, phi_c, delta_phi,
                                       Apsi, psi_c, delta_psi,
                                       T_tr):
        # First time: just set base, no transition
        if not self._has_base:
            self.Aphi = Aphi
            self.w = w
            self.phic = phi_c
            self.delta_phi = delta_phi

            self.Apsi = Apsi
            self.psic = psi_c
            self.delta_psi = delta_psi

            self._has_base = True
            self._phi_tr_active = False
            self._psi_tr_active = False
            return

        # Old full values at t_k
        phi_old = self.phi_input(t_k)
        phid_old = self.phi_d_input(t_k)
        phdd_old = self.phi_dd_input(t_k)

        psi_old = self.psi_input(t_k)
        psid_old = self.psi_d_input(t_k)
        psdd_old = self.psi_dd_input(t_k)

        # Update base params
        self.Aphi = Aphi
        self.w = w
        self.phic = phi_c
        self.delta_phi = delta_phi

        self.Apsi = Apsi
        self.psic = psi_c
        self.delta_psi = delta_psi

        # New base at t_k
        phi_base0 = self._phi_base(t_k)
        phid_base0 = self._phi_base_d(t_k)
        phdd_base0 = self._phi_base_dd(t_k)

        psi_base0 = self._psi_base(t_k)
        psid_base0 = self._psi_base_d(t_k)
        psdd_base0 = self._psi_base_dd(t_k)

        self._phi_tr_b = self._compute_transition_coeffs(
            phi_old, phid_old, phdd_old,
            phi_base0, phid_base0, phdd_base0,
            T_tr
        )
        self._psi_tr_b = self._compute_transition_coeffs(
            psi_old, psid_old, psdd_old,
            psi_base0, psid_base0, psdd_base0,
            T_tr
        )

```

```
176         self._phi_tr_t0 = t_k
177         self._phi_tr_T = T_tr
178         self._phi_tr_active = T_tr > 0.0
179 
180         self._psi_tr_t0 = t_k
181         self._psi_tr_T = T_tr
182         self._psi_tr_active = T_tr > 0.0
183 
184         self._has_base = True
185 
186     # ----   (t),     (t),     (t) ----
187 
188     def phi_input(self, t):
189         if not self._has_base:
190             return 0.0
191         q, _, _ = self._q_and_derivs(t, self._phi_tr_t0, self._phi_tr_T, self.
                _phi_tr_b) if self._phi_tr_active else (0.0, 0.0, 0.0)
192         return self._phi_base(t) + q
193 
194     def phi_d_input(self, t):
195         if not self._has_base:
196             return 0.0
197         _, dqdt, _ = self._q_and_derivs(t, self._phi_tr_t0, self._phi_tr_T, self.
                _phi_tr_b) if self._phi_tr_active else (0.0, 0.0, 0.0)
198         return self._phi_base_d(t) + dqdt
199 
200     def phi_dd_input(self, t):
201         if not self._has_base:
202             return 0.0
203         _, _, d2qdt2 = self._q_and_derivs(t, self._phi_tr_t0, self._phi_tr_T, self
                ._phi_tr_b) if self._phi_tr_active else (0.0, 0.0, 0.0)
204         return self._phi_base_dd(t) + d2qdt2
205 
206     # ----   (t),     (t),     (t) ----
207 
208     def psi_input(self, t):
209         if not self._has_base:
210             return 0.0
211         q, _, _ = self._q_and_derivs(t, self._psi_tr_t0, self._psi_tr_T, self.
                _psi_tr_b) if self._psi_tr_active else (0.0, 0.0, 0.0)
212         return self._psi_base(t) + q
213 
214     def psi_d_input(self, t):
215         if not self._has_base:
216             return 0.0
217         _, dqdt, _ = self._q_and_derivs(t, self._psi_tr_t0, self._psi_tr_T, self.
                _psi_tr_b) if self._psi_tr_active else (0.0, 0.0, 0.0)
218         return self._psi_base_d(t) + dqdt
219 
220     def psi_dd_input(self, t):
221         if not self._has_base:
222             return 0.0
223         _, _, d2qdt2 = self._q_and_derivs(t, self._psi_tr_t0, self._psi_tr_T, self
                ._psi_tr_b) if self._psi_tr_active else (0.0, 0.0, 0.0)
224         return self._psi_base_dd(t) + d2qdt2
```

A.3.4 rollout_utils.py

```python
import numpy as np

def rollout_and_collect(env, model=None, deterministic=True):
    obs = env.reset()

    t_all = []
    y_all = []
    z_all = []
    phi_all = []
    psi_all = []

    done = False
    step_idx = 0

    while not done:
        if model is None:
            action = env.action_space.sample()
        else:
            action, _ = model.predict(obs, deterministic=deterministic)

        obs, reward, done, info = env.step(action)

        t_seg  = info["t_eval"]
        y_seg  = info["y_sim"]
        z_seg  = info["z_sim"]
        phi_seg = info["phi_seg"]
        psi_seg = info["psi_seg"]


        # avoid duplicate times at boundaries
        if step_idx == 0:
            t_all.append(t_seg)
            y_all.append(y_seg)
            z_all.append(z_seg)
            phi_all.append(phi_seg)
            psi_all.append(psi_seg)
        else:
            t_all.append(t_seg[1:])
            y_all.append(y_seg[1:])
            z_all.append(z_seg[1:])
            phi_all.append(phi_seg[1:])
            psi_all.append(psi_seg[1:])

        step_idx += 1

    t_all = np.concatenate(t_all)
    y_all = np.concatenate(y_all)
    z_all = np.concatenate(z_all)
    phi_all = np.concatenate(phi_all)
    psi_all = np.concatenate(psi_all)

    return t_all, y_all, z_all, phi_all, psi_all
```